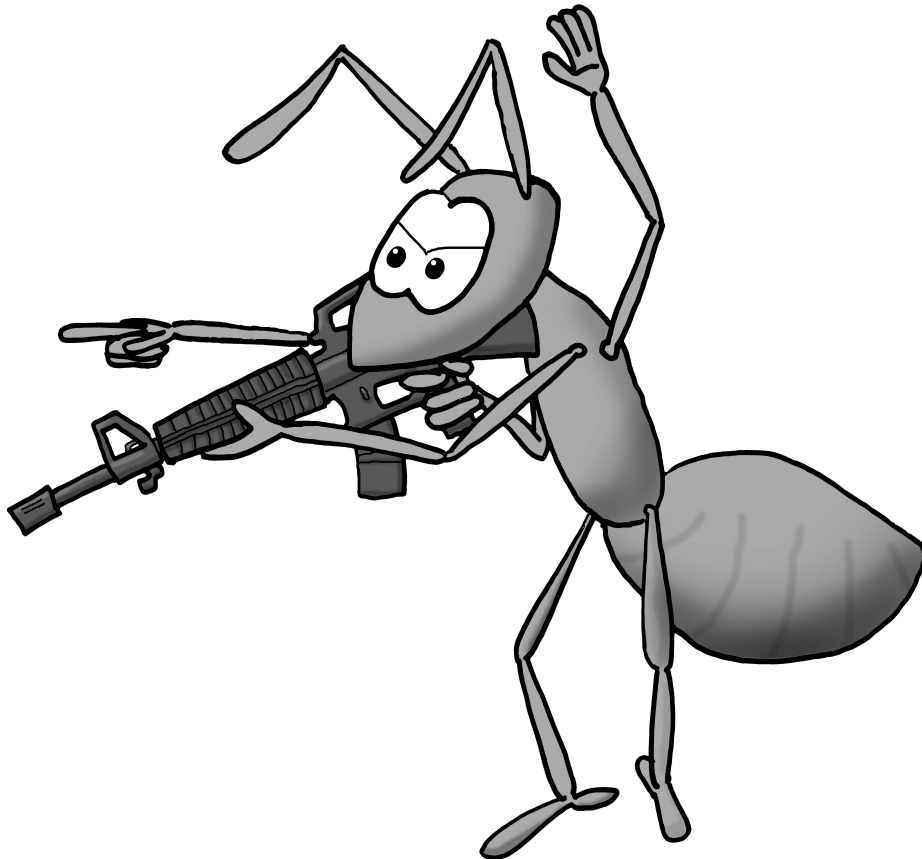


Language technology

AntL

Creating a language for MyreKrig



Spring 2004
SW4, project group s402a
Department of Computer Science
University of Aalborg



TITLE:

EN: Language technology

DK: Sprogteknologi

SUBTITLE:

AntLang:

Creating a language for MyreKrig

PROJECT PERIOD:

SW4,

February 2nd - May 28th 2004

PROJECT GROUP:

S402A

GROUP MEMBERS:

Brian G. Jørgensen,
qte@cs.auc.dk

Eckhart T. H. Pedersen,
corner@cs.auc.dk

Kristian S. Bødker,
boedker@cs.auc.dk

Peter Sørensen,
ptrs@cs.auc.dk

Tinus Nortsved
tinus@cs.auc.dk

SYNOPSIS:

This report documents the design of AntL, a programming language targeted for programmers developing programs for a programming game called MyreKrig, and the implementation of a compiler for the AntL language. The first part contains an analysis of what is required of the language. The design part covers theory and design of a programming language. The implementation part describes structure and details of selected parts of the compiler implementation. Finally the last part describes reflections on the language concept and the compiler implementation. One of the conclusions drawn, is that the language made through the semester supports the agent abstraction.

SUPERVISOR:

Thomas Vestdam
odin@cs.auc.dk

NUMBER OF COPIES: 7

NUMBER OF PAGES: 75

Preface

This report is written by the project group s402a, as part of the 4th semester of Software Engineering, spring 2004 at Department of Computer Science, Aalborg University. The outline for this semester is called "Language technology" (the danish title is Sprogteknologi), and focuses on the processes involved in designing and implementing a programming language.

This report documents the development of the AntL language and the implementation of a compiler for the language. A run-able version of compiler, its source code, documentation and example programs can be found on the accompanying CD which is described in appendix A.

Type setting

Throughout the report source code, pseudo code and syntax is written in `monospaced` font. In source code a backslash (\) indicates that the line breaks and continues on the next line.

Brian Jørgensen

Eckhart Pedersen

Kristian Bødker

Peter Sørensen

Tinus Norstved

Contents

1	Introduction	1
2	Problem Analysis	2
2.1	Programming games	2
2.2	Description of MyreKrig	5
2.3	MyreKrig as an Agent System	10
2.4	Problem statement	14
3	Language design	15
3.1	Paradigm	15
3.2	Language Features	15
3.3	Program Structure	21
3.4	Semantics	24
4	Compiler design	32
4.1	Compiler Theory	32
4.2	Tools	37
4.3	Compiler	41
4.4	Error Reporting and Handling	51
5	Implementation	52
5.1	Preprocessor	52
5.2	Contextual Analysis	55
5.3	Code generation	57
6	Reflections	62
6.1	Concept and Language Discussion	62
6.2	Compiler Discussion	63
7	Future development	64
7.1	Optional Runtime-checking	64
7.2	Variable Types	64
7.3	Debug Information	64
7.4	Execution Time Optimization	65
7.5	Memory Optimization	65
7.6	Inherit Roles from Super Roles	65
7.7	Editor	66
8	Software processes	68
8.1	Development Strategy	68
8.2	Quality Goals and Evaluation	68

8.3 Testing	69
9 Conclusion	73
Bibliography	74
Appendixes	75
A Accompanying CD	76
B Rambo.ant	77
C Rambo.ant in C	78
D Example.ant	83
E Syntax	87
F Null vs Rambo	105
G Null vs Rambo3	107

INTRODUCTION

Programming games are games in which the participants don't compete directly, but where the goal is to write programs/algorithms that compete against each other. Some of these programming games offer a graphical interface to create the behavior/strategy, but in most often the behavior is written in a common programming language.

MyreKrig is an example of such a programming game, where the competitors create ant races that battle each other in a virtual environment. In MyreKrig ants are written in ANSI C and since the C language is a general purpose language and not a language developed specifically to make ants, the process of making new ant races can be quite difficult. In order to make ant development easier and allow the programmer to make more sophisticated ants, we want to create a new programming language designed specifically for ant development.

PROBLEM ANALYSIS

In this chapter we will discuss our motivation and basis for creating a new programming language for ants in MyreKrig. The chapter will begin with a description of programming games in general and will then move onto describing the programming game MyreKrig and its rules in detail. Thereafter we will give a brief introduction to the subject of agents and agent oriented programming. We will also show that MyreKrig can be considered an agent based game. In the conclusion to this chapter a detailed problem statement will be presented.

2.1 Programming games

The concept of programming games is not easily explained as it covers a wide variety of different games. A common feature of these games is that people, often from all over the world, create their own entry to the competition and then have it compete against the creations of other participants. The programmer¹ or a team of programmers create an autonomous entity. The main difference between programming games and a conventional game is that when the game begins the creator no longer has any control over the entity. The entity is placed in an environment with which it can interact to complete the game objectives. The environment can contain both friendly and enemy entities and various other objects. In the majority of programming games the entity is submitted as source code in some programming language, which defines the rules by which the entity interacts with the environment.

Games employ different victory conditions and thereby attract different parts of the computer science community. Some games reward small and time-efficient entities either by allowing competing entities to move around while the entity is deciding what to do, or by multiplying the final score of the entity with a score for its usage of execution time and memory. Other games only look at the final result and doesn't punish time or memory demanding entities (like a game of chess).

Participants have different motivation for competing in the programming games. Some participate for fun and mainly see it as a game where they can gain prestige, others use the games as part of their scientific research and as a way of testing their research in practice, and some are mostly interested in the price that some games reward the winner with [11].

2.1.1 The Games

The next section will contain an overview of the different game types. The games described have been chosen such that they should show the different approaches to the programming games genre.

Genre: Robot vs Robot, War

¹The principal participants in programming games.

Robocode Robocode [1] is a good example of a robot fighting game, where the different robots can navigate around a small arena, scan for enemies, aim for targets and fire their weapon. The game interface furthermore allows programmers to lock on targets by reading bearing and distance of its targets, and then turn the gun-turret to point in the direction of the target it has locked on.

Robocode is designed to introduce novice programmers to object oriented programming by using some of the features available in the java language.

Commercial game: Mindrover With Mindrover [4] CogniToy has tried to make programming games accessible to people with little or no programming experience. Instead of having to type in code, the users can select from a collection of control actions to specify how the robot should act in different situations.

In an attempt to be a commercial success they have developed a programming interface which they hope will appeal to the young and upcoming programmer. They have adopted the **event-based programming paradigm** and developed a graphical development tool to produce small programs using a drag-and-drop interface, where the programmer drag different functionality to the robot and then interconnect them with logical statements.

Genre: Robot vs Robot, Non-violent

Robot Auto Racing Simulator Compared with most of the other programming games [7], RARS (Robot Auto Racing Simulator) is a bit different. Whereas programmers in the other games often have to design a robot, which can fight against other robots, in RARS they have to develop the brain of a race-car driver and hence define how he steers the car through sharp corners etc. It is possible to push the other contestants of the track, but each race has a race director, and he alone decides whether this is allowed (and in general they do not permit this kind of action).

The game comes with a 3D-engine so the programmers can watch their creations every move on the track. Combined with the statistics provided both in-game and afterwards programmers can look into how to improve their driver. RARS is written in C++ and hence also use the object oriented programming paradigm.

Genre: Computer vs Human

A fundamental question asked by A. M. Turing (from now on refereed to as Turing) in the 1950s was: "Can machines think?" [14]. Turing devised a game from this question called "The imitation game" in which 3 people are involved; a man, a woman and an interrogator of either sex. The interrogator is placed in a separate room and his mission is to determine which of the two is the man and which is the woman. The interrogator may ask as many questions to the two persons as he finds necessary. The real problem in this game becomes apparent when the man or the woman is replaced by a computer. The question is whether the interrogator would be fooled as often as before.

Turing himself answered this question with a yes, as he thought that computers can think. However, another question arose: "If a computer could think, how could we tell?". The latter question is presently known as the Turing Test [11], which basically refers to the same game



Figure 2.1: Screenshot from a game with multiple participants.

as above, with the exception that one of the people being interrogated is a computer. The interrogator's mission is now to determine which is the human and which is a computer.

The Turing Test has been extended even further by Caltec [2] which hosts a game where the interrogator also could be a computer. There is no specific paradigm connected with these tests and they accept many different languages, with the only requirement that they can receive some input and give some output. It is up to the programmer to decide which paradigm he or she judges to be the best to emulate human behavior.

Genre: Mechanical robots

Why compete in a virtual world when it is possible to compete in the real world? This seems to be the credo for the programmers and engineers that compete in this type of game. The preferred game to play in this genre is soccer [5] [6]. The size of the robots that compete vary from human size down to RCX² size. Again there are no restrictions on which paradigm to use here. It is up to the programmers own discretion.

2.1.2 The Lessons Learned

There are practical uses for the techniques and insight gained through programming games, since these can be used in real world application. An application could be in programming small embedded system where you only have a limited amount of resources available yet another example could be the NASA Martian rovers Spirit and Opportunity [10] if it is possible to execute on smaller hardware it is possible to conserve weight yielding a bigger payload. General knowledge is gained about algorithms, reaction on input and decision making.

²LEGO mindstorm

2.2 Description of MyreKrig

MyreKrig is a game where different ant teams battle each other in a virtual world build upon a torus. Each team starts with a base and a random number of ants, and the only thing that is in this world apart from these ants and bases is food. The goal for each team is to be the one with the most points after a specific number of turns, where points are calculated as the sum of the number of ants and bases that a team has. A team can only grow by gathering food, which is distributed randomly on the torus. But a team can also lose some of its ants in battle since ants from different teams can kill each other. Summarized the goal for the competitors/programmers is to create an ant team that gathers a lot of food, that attacks enemies and is able to defend itself from the other teams. MyreKrig has been developed and is maintained by Aske Simon Christensen [3].

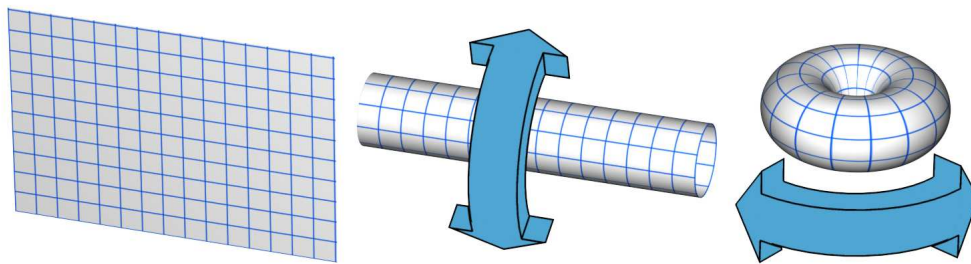


Figure 2.2: Torus on which the ants move around.

2.2.1 A game in MyreKrig

A game in MyreKrig consists of a number of battles, in which the size of the map and the placement of the teams is determined with a pseudo-random seed at the beginning of each battle. Each battle is then again divided into at most 20,000 turns, where all teams get a chance to move their ants during each turn. The order in which the teams get their chance is random, which makes it possible for a team to move twice in a row. Also new food is placed on the torus at the beginning of each turn.

The torus that the ants live in is divided into fields, like a chessboard, and ants can only see, and only move onto the surrounding fields and the field they are placed on. The fields that an ant has access to are numbered so when the programmer wants to move onto a field or obtain information about what there is on the field, then he has to use these numbers (see figure 2.3). The programmer can gain information about the environment in C by obtaining the value of the following:

- **field[i].NumAnts**
contains the number of ants on field i
- **field[i].Team**
contains the team number of ants on field i . The ant itself is team 0 and the other teams are numbered from 1 (total number of teams - 1). If there isn't any ants on field then it returns 0 (the same as if there were an ant from its own team!)

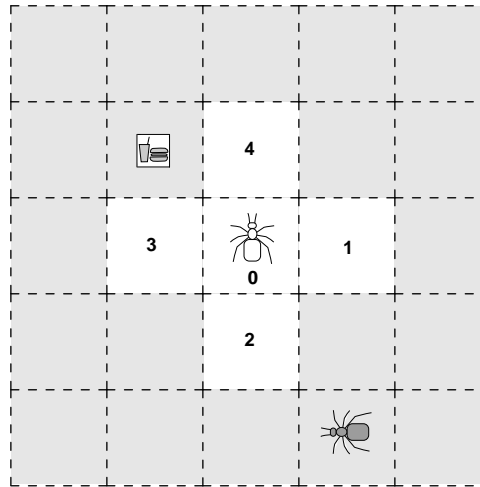


Figure 2.3: Checkered surface showing viewing field from an ant.

- **field[i].Base**
contains 1 if there is a base on field i and 0 if there isn't
- **field[i].NumFood**
contains the number of food laying on field i .

Apart from what an ant detects from the environment then it has access to what it has remembered from earlier turns, and what other ants (on the same field) remembers. This is the only way in which ants can communicate with each other and it will be explained in detail later in this section. When the ant has gathered all the info it needs from the environment it is able to perform one of the following actions:

- (0) Stay
- (1-4) Walk 1 field in one of the following directions: right, down, left or up
- (9-12) If there is food on the field where the ant is placed, it drags this food with it as defined above
- (16) If there is at least 25 ants and 50 pieces of food on the field, then it may build a new base. The cost of building a base is the same as the requirements.

The numbers at the beginning of each line represent what the ant should return as its next move e.g. it will move left if it returns 3, and build a base if it returns 16. The ant stays if it returns a number that doesn't match one of those defined above - or if it want to build a new base and the requirements isn't fulfilled. The numbers is the same as used when checking the environment (see figure 2.3), where drag is equal to direction + 8.

A very important factor that comes into play whenever the ant is going to make its move and that is how fights are resolved in MyreKrig. A fight begins whenever an ant from one team moves onto a field where there are an ant from another team, and they are resolved just like fights are resolved in chess. The ant moving onto the field always wins the fight, the only difference between chess and MyreKrig is that there can be multiple ants on a field, but the result is the same. An ant kills all ants and/or destroys the base if it moves onto a field where there are enemy ants and/or an enemy base.

Points

MyreKrig ends a battle based either on points or the number of turns, where a teams points is given by the following definition:

$$\text{points} = \text{number of ants} + \text{number of bases} \times 75$$

This is updated after each turn, due to new ants/bases and/or lost ants. A battle ends when one of the following conditions is true:

- A team has 75% of the **totalpoints**
- There has been 10000 turns and a team has 60% of the **totalpoints**
- The game times out (after 20000 turns)

Where **totalpoints** is the sum of the two teams with the most points, and the winner of that battle is then the team with the most points. The game ends after 100 battles (the default setting) after which it shows game statistics (see table 2.1).

Team	Battles	Won	Bases	Ants	Comb	Time	Vict	Perf	Pres
A5	183	43	1.0	5200	51.8%	425	23.5%	25.1%	17.6%
SkyNET	198	3	1.0	1512	59.5%	415	1.5%	1.6%	18.0%
Rambo	166	0	1.0	28	86.9%	0.0%	0.0%	0.0%	0.0%

Table 2.1: An example of the statistics that is shown after a game ends, taken from the last game on www.myrekrig.dk. (Note: "Size" and "Age" columns have been cut out)

2.2.2 An ant in C

To understand and to use the statistics it is necessary to know what an ant looks like in C. This subsection will describe the general structure of an ant using "Rambo", which is one of the ants included with MyreKrig.

There are four requirements when creating an ant; an ant has to and may only include "Myre.h", there has to be a struct representing its brain, a method that returns an int and an ant definition. The method returning an int is responsible for determining the ants next move as stated earlier. The antdefinition contains the name of the C file, the name that should be displayed in the programs output (with an optional color) and the name of the struct and the method. In Rambo's case (figure 2.4) then is the struct called rb and the method called RB.

Rambo's purpose is to stay put unless it detects an enemy and it is placed on the same field as its homebase. Rambo is a good example of a guard, since it doesn't stray around on the map looking for food or enemies - it stays near its base. The author (Jørn Holm) uses the "Base", "Team" and "NumAnts" to determine the ants behavior.

Brain struct

All variables declared in the brain will be initialized with 0 when the game starts, with the exception of the first "u_long"³ which will get a random value. After the game has begun these

³u_long: the first 32 bits

```

#include "Myre.h"
struct rb {
};
int RB(struct SquareData *f, struct rb *m) {
    int i = 0;
    int max = 0;
    int best = 0;
    for(i = 1; i < 5; i++){
        if(f[i].Base == 1)
            return i;
    }
    for(i = 1; i < 5; i++){
        if(f[i].Team > 0){
            if(f[i].NumAnts > max){
                max = f[i].NumAnts;
                best = i;
            }
        }
    }
    return best;
}

DefineAnt(Rambo, "Rambo#FF0000", RB, struct rb);

```

Figure 2.4: Rambo - by Jørn Holm (gekko@myrekrig.dk)

variables will be the only ones that keeps their value between turns. The brain struct serves as an ants memory and it is accessible by the ant itself and by all other ants on the same field. Rambo is a special ant since it has an empty brain struct, but in this subsection we will assume that Rambo has the following brain struct:

```

struct rb {
    int mostkills;
};

```

The brain struct is an array, of which the size is equal to the number of ants on the field where an ant is placed. A Rambo ant can access the "mostkills" brain variable within its body using either `m->mostkills` or `m[0].mostkills` ("m" is a pointer to the brain struct). If there are other ants on the same field, then will the ant be able to access the brain of other ants using `m[i].mostkills` where "i" is the brain of ant number "i". A brain variable is accessed like any other variable and communication among ants is therefore very limited, since they only can read or write in each others brain and this is the only way that ants can communicate since it isn't allowed to have global or static variables.

2.2.3 Official Tournament

MyreKrig has its own official tournament [3] where everyone can submit their ant-algorithm and hence become a team in the game. The author of MyreKrig runs the game from time to time with all the ants that are submitted to him and he has made an official ranking using the statistics that the game gives. An ants rank is decided by the following:

$$\text{Overall rating} = \frac{\text{Perf} \times \text{Pres}}{\text{Vict}} \quad (2.1)$$

If we apply this ranking system on table 2.1 then would SkyNET be the winner (table 2.2) even though it only won 3 battles (against the 43 wins of A5). A5 seems superior since it has the highest numbers in almost every category but the calculation shows otherwise, but the ranking system takes other factors into account, and those are **Brainsize** and **Time**. Brainsize is the amount of memory needed (in bytes) to store the variables declared in the brain, e.g. an int requires 4 bytes and a char requires 1 byte. Time is equal to an ant teams execution time. An

Team	Brainsize	Time	Vict	Perf	Pres	Overall
A5	16	425	23.5%	25.1%	17.6%	18.7%
SkyNET	1	415	1.5%	1.6%	18.0%	19.2%
Rambo	0	354	0.0%	0.0%	0.0%	0.0%

Table 2.2: The overall brainsize, time and overall performance of A5, SkyNET and Rambo

ants victory rating is equal to how many battles it has won out of how many it has participated in and the prestige and performance ratings is given by the following:

$$\begin{aligned} \text{(Performance rating) Perf} &= \frac{\text{Vict}}{\text{Time} \times \text{median}_{\text{Time}}} \\ \text{(Prestige rating) Pres} &= \frac{\text{Vict}}{\text{Brainsize} \times \text{median}_{\text{Brainsize}}} \end{aligned}$$

This explains why SkyNET is able to outperform A5 and as well show that the brainsize is important to have in mind when developing an ant.

2.3 MyreKrig as an Agent System

An agent is an abstract concept, usually used when designing software which must act upon its environment in an autonomous manner. This section will give a brief introduction to the topic of agents and to what extent the agent abstraction can be applied to MyreKrig. The conclusion of this section will discuss what benefits the agent abstraction can give when developing ants. This section is based on [12]

2.3.1 Agent Definition

An agent in relation to artificial intelligence is a concept which is not clearly defined, as it is an abstract concept. The definition we have chosen to use in this report is the following:

”An agent is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **effectors**” -[12].

An example of such an agent could be something as relatively simple as an air conditioning system. An air conditioning system typically has temperature and humidity sensors (sensors) and fans, cooling systems and humidifiers to manipulate the environment (effectors). A more complex example could be a Martian Rover, which must be able to navigate in a hostile environment on another planet, with no or minimal direct human control. The agent abstraction is illustrated in figure 2.5.

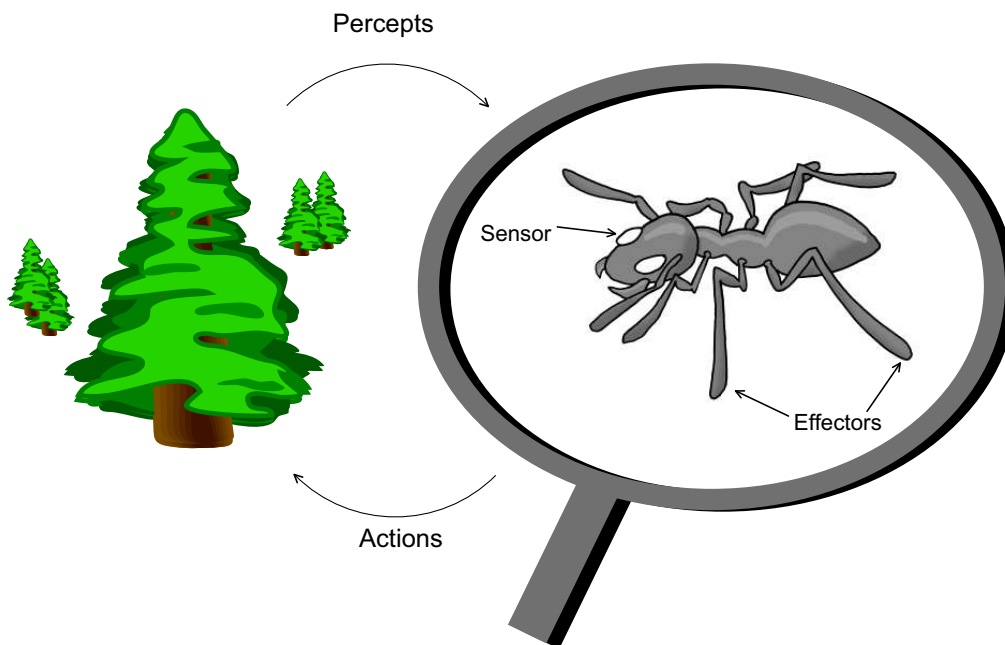


Figure 2.5: An agent with sensors and effectors in an environment

In MyreKrig ants can perceive part of their environment and they can perform several actions, such as moving around and building a base. Ants thus satisfy the definition of an agent and can be considered as agents. This leaves the question of how an agent actually works and how the agent abstraction can be used successfully. In order to be able to explain this, we need to consider a few more concepts:

- **Goals** and possibly a set of **subgoals** which the agent should achieve.
- **Beliefs** either given by the programmer, obtained from the environment or a combination of both.
- **Reasoning Rules** describing how to act under which conditions.
- **Memory** used to store the beliefs of the agent.

All agents have some reasoning rules, even though they may be very simple. These rules can use the perceptions of the environment and the current beliefs to determine what actions should be taken. Not all agents use a belief system (e.g. an air conditioning system), and most agents are unaware of their goals (e.g. an air conditioning system), even though awareness these are essential when designing reasoning rules for an agent.

In MyreKrig ants can have a memory (called brain in MyreKrig) to store beliefs, although this is not a requirement. Ants in MyreKrig usually are unaware of the overall goal and focus on smaller subgoals instead. The reason for this will become apparent later when the environment in MyreKrig is discussed.

2.3.2 Rational and Ideal Rational Agents

A rational agent is an agent which always does the right thing, which means the action performed by the agent would cause the agent to be successful. Creating an agent that always does the right thing becomes increasingly difficult, if not impossible, the more complex the desired behavior becomes. Instead, agents are usually created with the goal to always do the right thing based on the agents beliefs and perceptions. Such an agent is called an ideal rational agent.

2.3.3 Utility

The success of an agent is based on whether the agent has achieved its goals. Sometimes there are various ways to achieve a goal with some ways being better in terms of speed, accuracy or other criteria. In order to reflect these differences, a more detailed measure of success is needed. This measure is called **utility**, where the utility is a number specifying the degree of success.

The concept of utility can be illustrated with robot agents in a factory which should recognize different parts and sort them accordingly. An agent that uses less time to recognize the parts will generally have higher utility than an agent which uses more time. In a similar fashion, an agent that can recognize parts more accurately than other agents will generally have higher utility.

The utility of an agent is calculated using an utility function. The utility function specifies exactly how the utility should be calculated and how trade-offs and conflicting goals should be handled. The above example could make use of a utility function of the form $p - t$, where p would be the percentage of correctly recognized parts, and t would be the amount of time needed to recognize a part. A higher number would mean a higher degree of success in this example. If time was not considered as important as accuracy, the percentage could be multiplied by a factor greater or equal to 1, thus decreasing the significance of time. If time was more important than accuracy the opposite could be done.

2.3.4 Autonomous Agents

Some agents do not consider their experiences when they determine their next actions. These agents are said to lack autonomy, as they are unable to alter their behavior regardless of what happens. A non-autonomous agent can be successful, but it may require certain conditions in order to achieve the desired results.

Autonomous agents are often preferred, as they are able to alter their behavior based on previous actions and sensory feedback. This makes it possible to use them in various and changing environments or in environments where the conditions are not fully known. An autonomous agent will probably still need some innate knowledge in order to survive long enough to gain experience.

2.3.5 Environments

The term environment covers the set of conditions under which the agent must act. Environments can differ in various properties, which determine what and how much the agent is able to perceive and predict. The most important properties will be discussed in the following paragraphs.

Static vs Dynamic

Environments can either be static, semi-dynamic or dynamic. In a dynamic environment the state of the environment can change while the agent is choosing its next action. In a semi-dynamic environment the environment does not change while the agent is deliberating, but the agents utility depends on the time it takes to choose its next action. In a static environment the agent may spend as long time as necessary without penalty.

Deterministic vs non-deterministic

Another property of environments is whether they are deterministic or non-deterministic. In a deterministic environment the next state of the environment is only determined by the current state and the agents actions. In a nondeterministic environment the next state of the environment cannot be predicted using the agents actions. A deterministic environment is easier to operate in as it may be possible to predict the outcome of the agents actions.

Accessible vs inaccessible

Environments can be differentiated by their level of accessibility. In a accessible environment the agent has complete access to the state of the environment. If the agent has access to all the information relevant to the current situation, the environment is considered effectively accessible. Programming an agent for an accessible environment is easier than programming an agent for an inaccessible environment, as the agent has more information available to base its decisions on.

Episodic vs non-episodic

In an episodic environment the agents actions do not influence the utility of subsequent actions. This means that the agent can work on one problem a time without having to think ahead.

The MyreKrig Environment

The environment in MyreKrig is very inaccessible, as an ant can only perceive adjacent fields. This also makes the environment effectively non-deterministic to the ant, as it is impossible to predict the next state of the environment by the ants actions (and the current state of the environment) alone. In addition ants take turns in random order and ants cannot predict the actions of other ants. The environment is also semi-dynamic, as the agents score decreases the longer it deliberates the next move. Last but not least the environment is non-episodic, as the agents actions do affect the outcome of the next action.

It can be difficult to create an agent that can operate well in such an environment, as it is difficult to determine any long term outcomes. This problem can be overcome by focusing on subgoals instead of the overall goals. The overall goal in MyreKrig can be split up into subgoals like finding food, bringing food back to base and building new bases. This can be achieved by introducing the concept of agent states. The state of an agent determines which subgoal currently should be achieved, and behavior rules are specified for each subgoal separately.

2.3.6 Agent Based Programming

There is no clear definition of what exactly an agent based language is. Our definition of an agent based language is a language that provides native access to the concepts of an agent. This means that an agent based language should make it possible to access and/or specify perceptions, beliefs, actions and goals/objectives. As not all agents actually make use of all of these concepts, some agent based languages may not include all of these concepts.

Advantages of Agent Based Programming

Agent based programming languages have some advantages over traditional imperative programming languages when writing software that can be expressed as an agent. Concepts like perceptions, actions and environments are intuitively understood by most humans, making agent based programs comparably easy to understand. In general, agent-based languages offer a new level of abstraction, making them easier to use.

Agents in C

The C programming language is a imperative programming language which does not provide native support for the concepts of the agent abstraction. C is thus not an agent based language and does not benefit from the agent abstraction, which could be useful when developing ants.

2.3.7 Conclusion

This section has shown that ants can be considered as agents, and how the agent abstraction can make agent development easier by providing a new level of abstraction. C does not support the agent abstraction, so a new agent based language could make the development of ants more intuitive. This language should give ant developers easy access to the concepts of the agent abstraction. The next section will give a more detailed description of our problem statement.

2.4 Problem statement

The goal of this project is to create a new programming language which can be used to create ants for the programming game MyreKrig. The language should be agent based in order to provide a higher level of abstraction compared to C.

Since the purpose of the language is very specific, it should be possible to provide easy access to all of the MyreKrig specific elements. As another consequence of the very specific purpose, the language has to be translated to C. This is a requirement as all ants in the competition must be written in ANSI C.

It was decided that the language should have the following properties:

- High readability - the code should be easy to read and understand.
- Agent based - the different agent concepts should be easily accessible.
- Expressive - it should be possible to create many different types of ants.

The compiler should have the following properties:

- Create C code - all competing ants must be written in ANSI C
- Create efficient code - the execution time and brain size has a impact on the score.

Some of these properties are conflicting. For instance it is very hard to make a language which has a high level of abstraction without reducing the runtime performance. The language must therefore be a trade-off between the above listed properties.

LANGUAGE DESIGN

The agent based approach to ant development raises some questions about the overall design guidelines that should be applied. This includes the questions of what paradigms to use and what language constructs to provide. The decisions that were made concerning the language design will be discussed in the following sections. This chapter use special typesetting to indicate different parts of the syntax and code examples.

- `<identifier>` indicates that the syntax allows the programmer to write an identifier.
- `[...]` indicates that the part between the two square brackets is optional.

3.1 Paradigm

Agent oriented programming is often not considered to be a true paradigm such as the imperative paradigm, since there is no clear definition of what a language must be able to support to be called a agent oriented. But some decision was needed as pure imperative languages does not support the agent abstraction very well. Therefore it was necessary to include features from other paradigms in order to create the appearance of an agent based language, although a very specialized one. To summarize it can be said that AntL is a imperative language with some declarative features.

3.2 Language Features

To create a higher level of abstraction in AntL (compared to C) it was necessary to determine how the various concepts of the agent abstraction could be useful when developing ants in MyreKrig.

3.2.1 Control structures

When developing ants in C it were discovered that they often are filled with "if"/"else if"/"else" control structures and they made the code unreadable. Two new control structures were added to AntL to make the code more readable.

Roles And Triggers

The role structure introduces the concept of subgoals, as discussed in section 2.3.1. The role of an ant determines the subgoal that should be achieved, and the developer can specify the actions that should be performed for the subgoal/role. This allows the developer to focus on one subgoal at a time and should thus make the code more readable. The syntax for a role block is illustrated below:

```
Role <role name>
  <statements>
endRole
```

The general idea behind role blocks is to allow the ant developer to specify different behavior for each role. Role blocks also makes the code more readable.

Triggers are patterns of sensor input and beliefs that cause an agent to perform some actions. The ant developer can specify the conditions that should activate the triggers, and the triggers can be specified to cause some actions when activated. This improves the readability of the behavior code and thus makes the code more manageable. The syntax of triggers is illustrated below:

```
On <triger name>
  <statements>
endOn
```

Triggers are used to act on the environment. Roles can be used to change a triggers impact, if it for example had the role "Soldier" then would it disregard triggers that made the ant act if it detected food. It would however attack any enemy it detects.

If and While

In addition to these new control structures, AntL supports the conventional "if"/"else if"/"else" and while control structures. These are used in a similar fashion as in C and are thus not discussed further here.

3.2.2 Data Types

AntL only supports one traditional data type, the integer (equivalent to int data type in C). In addition to the integer data type several new types have been created and they will be described in the following paragraphs.

Integer

The integer type has to be written as 'integer' instead of 'int', a design choice that was made to increase the readability.

Position

The position type contains the x and y components of a position. Positions can be added, subtracted and compared with an equality operator. Size comparisons, multiplication and division are not possible with positions. A position value is written as (<x_component>, <y_component>) where both the x and y-component are required to be integer-values.

Direction

The direction type represents one of the directions an ant can look at or move to. It can be one of the following values:

```
here, right, down, left, up
```

Action

The action type represents the possible actions that an ant can perform in MyreKrig. An action can have be one of the following:

- `<direction>` - A move in the specified direction
- `drag <direction>` - A move in the specified direction, with food
- `build` - build a base

Boolean

The boolean data type can assume the values `true` and `false`

Role

The role data type can be any role.

3.2.3 Statements

A new problem was discovered after adding triggers and roles to AntLIf one trigger want the ant to move right and another want the ant to move left, then which should it choose. The answer is simple in C, since the action would be a return statement which stops any further execution. To add a more adaptive way of deciding what move to make it was decided to add the request, vote and force statements to AntL.

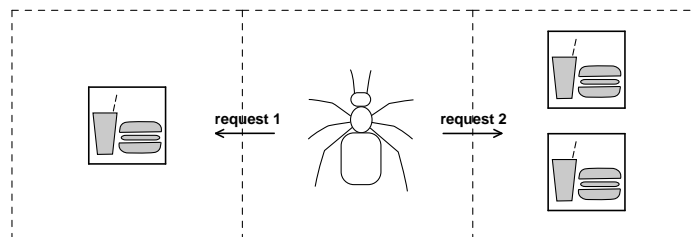


Figure 3.1: Using request statements to determine that there is a more important reason to move right

The Request Statement

If the ant has to choose between several actions, but some of the reasons for an action are more important than others, the request statement can be used. The request statement is used by specifying the priority for performing an action. The action with the highest priority is the action that is performed. If there are two equal priorities for two different actions the the action that was requested first is performed.

An example where the request statements could be used is illustrated by figure 3.1. An ant has found two piles of food. There are 10 chunks of food on the right side, and 5 chunks of food on the left side. A trigger has determined that there are food on its left side and requests

that the ant should move left with a priority of 5. Another trigger determines that there is food on its right side and requests that the ant should move right with a priority of 10. As moving to the right has a higher priority than moving to the left, the ant will move to the right.

The Vote Statement

The vote statements should be used in a situation where there are several similar actions with a low priority and a different action with a higher priority. If a request statement were used in the situation above then would the action with the highest priority win. The vote statement allows the programmer to give each action a weight. Each action is then given a priority equal to the sum of it weights.

An example for the use of vote statements is illustrated by figure 3.2. An ant has been searching for food without success and is returning to the base. On its way to the base, the ant finds two piles of food. A trigger detects the food on the left side and votes left with weight 2. Another trigger detects the food on the right side and votes right with weight 2. The ant currently has a subgoal which is to return to its base. The base is on the ant's right side and it votes right with weight 1. As there now are more votes for moving to the right than votes for moving to the left the ant will move to the right.

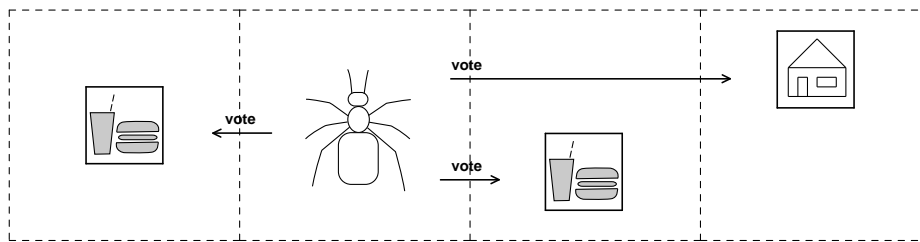


Figure 3.2: Using vote statements to determine that there are more reasons to move right

Votes Or Requests?

In the previous examples, it would have been possible to use requests instead of vote statements and vice versa. Which of the two should be used depends entirely on the situation and the developers intuition. An ant can make use of both vote and request statements at the same time although this can lead to the more confusing source code.

The Force Statement

If the ant developer wants to make sure that a specific action is performed on some condition, the force statement can be used. The force statement overrides all request and vote statements, and the action specified by the force statement is performed. If several force statements are used the one that is encountered first determines what action is performed.

An example for the use of force statements is illustrated by figure 3.3. The ant can choose between killing the enemy to the right or picking up the food to its left. Using a force statement ensures that enemy ants always are killed when the ant encounters them.

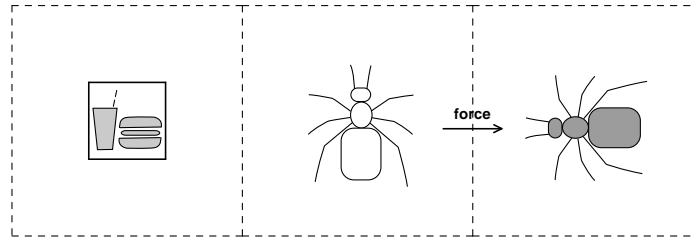


Figure 3.3: Using a force statements to specify that enemy ants should be killed, regardless of any other reasons to perform some other action.

Changing Roles

Changing roles could have been done by having a global role variable and changing the value of that variable or by issuing a change role statement. It was however decided that both cases would make the code unreadable. It would furthermore lead to infinite loops in case it were decided to execute role specific code dynamically. Figure 3.4 shows an example of such a loop.

To avoid confusion and loops we decided to use the request, vote and force statements. The use of these statements postponed the decision to the end of the behavior. The change role statement is evaluated separately from actions.

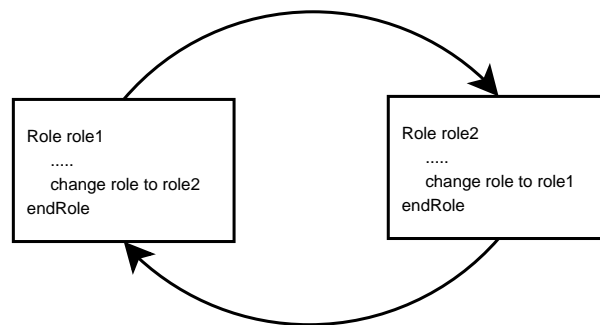


Figure 3.4: An infinite role changing loop.

Other statements

In addition to these new statements, AntL supports conventional assignments and variable declaration statements and return statements in function declarations. These are used in a similar fashion as in C, and are thus not discussed further here.

3.2.4 Expressions

Expressions in AntL can be written in a similar fashion as expressions in C are written. Two special language subexpressions have been added to AntL which can be used in expressions. These will be discussed in the following paragraphs.

Environment

Ants can perceive their environment by checking for ants, food and bases on adjacent fields. The syntax for detecting the environment is illustrated below:

```
<variable> <direction_expression>
```

<variable> can be one of the following:

- numberFood - the amount of food (integer)
- friendAnts - the number of ants that are not enemies (integer)
- enemyAnts - the number of ants that are enemies (integer)
- friendBase - whether there is a friendly base (boolean)
- enemyBase - whether there is an enemy base (boolean)

<direction_expression> is an expression that specifies the direction of the adjacent field that is examined.

Communication Variables

Communication has to do with reading from or writing to brain variables in other ants that are located on the same field. Communication can be used to tell other ants where food is located or give them orders. It would be possible to implement a military hierarchy using roles and giving orders by writing 'orders' into other ants brains. The syntax for communication variables is illustrated below:

```
<variable_name>@friends[<integer_expression>]
```

The <variable_name> specifies the name of the variable that should be accessed, and the <integer_expression> specifies the number of the ant that should be communicated with. In order to communicate with all ants on the same field a `while` loop should be used.

3.3 Program Structure

The program structure of an ant in AntL follows the "declare first, use later" philosophy. This means that the different blocks must appear in a fixed order, so that blocks can refer back to other blocks if necessary. This approach was chosen because it was deemed to result in the most intuitive code structure. The fixed order makes it possible to know what blocks to expect where when looking at an ant written in AntL.

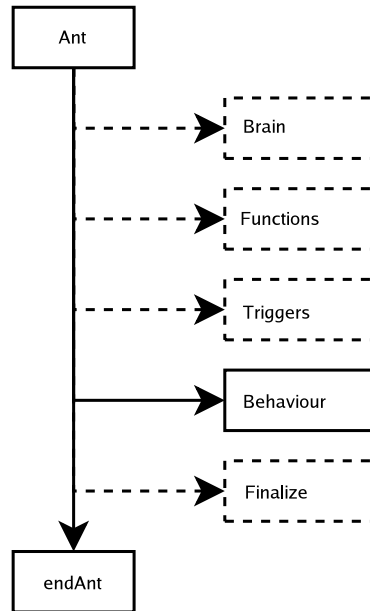


Figure 3.5: The structure of a program written in AntL. Dotted lines indicate that a block is optional.

As some of the blocks are not necessary to create an ant, it was decided to make these blocks optional. This saves the small amount of time it takes to write these blocks and in addition the developer does not need to worry about some empty blocks. The basic structure of an ant in AntL consists of 5 parts, illustrated by figure 3.5. These blocks will be discussed in more detail in the following paragraphs.

3.3.1 Brain

The brain block is the memory of the ant. Only variables declared in the brain block keep their values from one turn to the next. It is possible to create an ant that has no brain and only uses the environment to determine its next actions, thus the brain block is optional. The syntax of the brain block is illustrated below:

```

Brain
  <variable_type> <variable_name>
  ...
endBrain
  
```

The brain block is positioned at the start of the ant program, as it can be accessed from all following blocks.

3.3.2 Functions

To help developers reuse functionality throughout the program the language has support for function and procedure calls. The order of the function declarations is important, as functions can only refer to other functions that have been declared already. The syntax for the function block is illustrated below:

```

Functions
  Function <function_name> [returns <variable_type>]
    ...
    return <expression>
  endFunction
  ...
endFunctions

```

The "returns <variable_type>" part is optional and if omitted the function will be interpreted as a procedure. The function block itself is also optional, as it is possible to create an ant without any functions. The function block is the second block in the program structure, as the following blocks may use any of the functions defined in this block.

3.3.3 Triggers

The triggers block contains all trigger declarations. Each trigger is defined by a single boolean expressions. If the expression is evaluated as true, the trigger is activated. The triggers block is optional, as it is possible to create an ant that does not use any triggers (instead it may rely on if/while control structures and/or roles). The syntax for the triggers block is illustrated below:

```

Triggers
  Trigger <trigger_name>
    <boolean-expression>
  endTrigger
  ...
endTriggers

```

The triggers block is positioned before the behavior and the finalize block, as these may relay on triggers.

3.3.4 Behavior

The behavior of an ant is specified in the behavior block. The behavior can be specified using the trigger and role control structures and any other control structures and statements. The behavior block is mandatory as an ant without behavior would not do anything. The syntax of the behavior block is illustrated below:

```

Behavior
  ...
  on <trigger_name>

```

```

    ...
    vote <action> weight <weight>
endOn
Role <role_name> [, <role_name>]
    ...
    on <trigger_name>
        ...
        request <action> priority <priority>
    endOn
    Role <role_name> [, <role_name>]
        ...
    endRole
    ...
endRoleendRole
endBehavior

```

The above is only one example of how a behavior block might be structured. trigger statements and role blocks can be nested without limitation, and trigger blocks can contain role blocks and vice versa. It is possible to create an ant without any role/trigger blocks.

3.3.5 Finalize

When the behavior block has been evaluated the ant has decided on an action to execute. To enable programmers to create side effects on the action decided upon. An example would be that the programmer wants to keep track of the position of the ant. In the brain block, the ant has a position variable containing the current position on the map, and when the ant moves left the position x-coordinate is decreased and so forth. When the program execution reaches the finalize block, a variable named `move` of type action.

```

Finalize
    ...
    Role <role_name> [, <role_name>]
        if (move = drag left or move = left)
            pos.x := pos.x - 1
        endIf
    ...
endRole
endFinalize

```

3.4 Semantics

This section will discuss the operational semantics for some of AntLs different language constructs as presented in the previous sections. Some language constructs has been omitted as it was judged that the difference between the semantics of the languages constructs in AntL and C¹ was not significant. The description of AntLs semantics will be given using the notation from [8].

3.4.1 The Formal Semantics

The semantics of a language can be given in two ways, a formal and a non-formal. The non-formal specification of the semantics was the prevalent way of specifying the semantic of a language until the advent of more formal specification methods. Below is an example of how a non-formal description for arithmetic expressions could be given.

Arithmetic expressions

The value of an arithmetic expression of the form $a_1 <operator> a_2$ depends on the infix operator. Assuming that a_1 is evaluated before a_2 to v_1 and a_2 to v_2 the expression is evaluated to $v_1 + v_2$ if the operator is a plus. It is the same for the operators time, divide and modulus. If a function is called as a part of the expression, the function is evaluated and its return integer is used.

The Formal approach uses a much more explicit way of specifying the semantics. In the following sections the formal description of AntLs operational semantics for variables, functions, triggers and the behavior will be given using a big-step semantic.

To get started the syntactic categories of AntL needs to be found and their abstract syntax specified which is most commonly given in **BNF**². The Syntactic Categories and an example of how the abstract syntax looks for one of the syntactic categories are listed below.

¹The target language

²Backus-Naur Form

The Syntactic Categories

$n \in \mathbf{Num}$	Numerals
$b_v \in \mathbf{Bool}$	Booleans
$pos \in \mathbf{Pos}$	Positions
$ac \in \mathbf{Act}$	Actions
$a \in \mathbf{Aexp}$	Arithmetic Expressions
$b \in \mathbf{Bexp}$	Boolean Expressions
$p \in \mathbf{Pexp}$	Position Expressions
$ac_e \in \mathbf{ACexp}$	Action Expressions
$S \in \mathbf{Sta}$	Statement
$D_A \in \mathbf{ErkA}$	Ant Declaration
$D_{Brain} \in \mathbf{ErkV}$	Brain Declaration
$D_F \in \mathbf{ErkF}$	Function Declarations
$D_T \in \mathbf{ErkT}$	Trigger Declarations
$D_V \in \mathbf{ErkV}$	Variable Declarations
$D_B \in \mathbf{ErkB}$	Behavior Declaration
$Fname$	<i>functionname</i>
$Tname$	<i>triggername</i>
$Vname$	<i>variablename</i>

The Abstract Syntax for boolean expression

$$b ::= b_1 \text{ and } b_2 \mid b_1 \text{ or } b_2 \mid b_1 \text{ xor } b_2 \mid b_1 \text{ nand } b_2 \mid \\ a_1 \text{ gt } a_2 \mid a_1 \text{ lt } a_2 \mid a_1 \text{ eq } a_2$$

3.4.2 The Specification

A quick overview of the different environments used to specify the semantics is given below:

- Env_V : The Variable environment
Is used to store values and type information about variables. Sto_V maps the locations onto values.
- Env_T : The Trigger environment
Is used to store the value of the triggers. Sto_T maps the locations onto values.
- Env_F : The Function environment
Is used to store return type, formal parameter, EnF_v and the current function environment.
- Env_B : The Behavior environment
Will be described in greater detail below.

Most environments, with the exception of the behavior environment, work in a similar fashion as the variable environment (see 3.6) with little or no extra information and will not be described. The behavior environment is significantly different because it makes use of the auxiliary function listed below. The behavior environment is furthermore special because it is static in contrast to all other environments. It is static with regard to the fact that all mappable values are present in the environment i.e., $\text{vote}_{\text{left}}$, $\text{vote}_{\text{right}}$, force , and request are already in the environment.

- **next**
Points to the next free location.
- **new**
Gives the next free location.
- **maxRequest**
Takes the current action and its priority and compares them to the ones already in the environment. The environment is updated with the action that has the highest priority.
- **typeRetrieve**
Retrieves the type of an entry in the environment.
- **addVote**
Add the weight of a vote statement to the one already in the environment.

Apart from `next` and `new` it also makes use of a function that will retrieve the type of an entry in the environment. `StoB` maps the locations onto values.

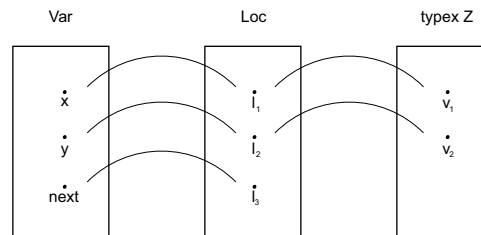


Figure 3.6: The environment structure

Brain Variables

The Transition-system

$$(\Gamma_{DV}, \rightarrow_{DV}, T_{DV})$$

The configuration

$$\begin{aligned} \Gamma_{DV} &= (\mathbf{ErkV} \times \mathbf{EnvV} \times \mathbf{StoV}) \cup \mathbf{EnvV} \times \mathbf{StoV} \\ T_{DV} &= \mathbf{EnvV} \times \mathbf{StoV} \end{aligned}$$

The variable environment is given by

$$\mathbf{EnvV} = \text{variablename} \rightarrow (\text{type} \times \mathbb{Z}) \cup (\text{type} \times (\mathbb{Z} \times \mathbb{Z}))$$

The transition relation

[Variable_{bss}]

$$\frac{\langle D_V, env_V[\text{variablename} \mapsto l][\text{next} \mapsto \text{new } l] \text{ sto}_V[l \mapsto (\text{type}, 0)] \rangle \rightarrow_{D_V} (env'_V, \text{sto}'_V)}{\langle \text{type } \text{variablename}, D_V, env_V, \text{sto}_V \rangle \rightarrow_{D_V} (env'_V, \text{sto}'_V)}$$

[Variable-empty_{bss}]

$$\langle \epsilon, env_V, \text{sto}_V \rangle \rightarrow_{D_V} (env_V, \text{sto}_V)$$

Functions

The Transition-system

$$(\Gamma_{DF}, \rightarrow_{DF}, T_{DF})$$

The configurations

$$\begin{aligned} \Gamma_{DF} &= (\mathbf{ErkF} \times \mathbf{EnvF}) \cup \mathbf{EnvF} \\ T_{DF} &= \mathbf{EnvF} \end{aligned}$$

Where the Function environment is

$$\mathbf{EnvF} = \text{Pname} \rightarrow \mathbf{Sta} \times \mathbf{Var} \times \mathbf{EnvV} \times \mathbf{EnvF}$$

The transition relation

[Function-Function_{bss}]

$$\frac{env_V \vdash \langle D_F, env_F[\text{functionname} \mapsto (S, D_V, env_V, env_F, \text{type})] \rangle \rightarrow_{D_F} env'_F}{env_V \vdash \langle \text{function } \text{functionname}(D_V) \text{ returns } \text{type } S \text{ endFunction } D_F, env_F \rangle \rightarrow_{D_F} env'_F}$$

[Function-Procedure_{bss}]

$$\frac{\langle D_F, env_F[\text{functionname} \mapsto (S, D_V, env_V, env_F, \text{void})] \rangle \rightarrow_{D_F} env'_F}{env_V \vdash \langle \text{function } \text{functionname}(D_V) S \text{ endFunction } D_F, env_F \rangle \rightarrow_{D_F} env'_F}$$

[Function-tom_{bss}]

$$env_V \vdash \langle \epsilon, env_F \rangle \rightarrow_{D_F} env_F$$

[Function-call_{bss}]

$$\frac{env_V[x_1 \mapsto l_1] \dots [x_n \mapsto l_n][next \mapsto l'], env'_F \vdash \langle S, sto_V \rangle \rightarrow sto'_V}{env_V, env_F \vdash \langle functionname(y_1, \dots, y_n), sto_V \rangle \rightarrow sto'_V}$$

where $env_F functionname = (S, x_1, \dots, x_n, env'_V, env'_F)$
and $l_1 = env_V y_1 \dots l_n = env_V y_n$
 $l' = env_V next$

Triggers

The Transition-system

$$(\Gamma_{DT}, \rightarrow_{DT}, T_{DT})$$

The configurations

$$\Gamma_{DT} = (\mathbf{ErkT} \times \mathbf{EnvT} \times \mathbf{StoT}) \cup \mathbf{EnvT} \times \mathbf{StoT}$$

$$T_{DF} = \mathbf{EnvT} \times \mathbf{StoT}$$

Where the Trigger environment is

$$\mathbf{EnvT} = triggername \rightarrow \mathbf{Bool}$$

The transition relation

[Trigger_{bss}]

$$\frac{env_V, env_F \vdash \langle D_T, env_T[triggername \mapsto l][next = new\ l]sto_T[l \mapsto v_{boolean}] \rangle \rightarrow_{DT} (env'_T, sto'_T)}{env_V, env_F \vdash \langle \mathbf{Trigger}\ triggername\ b\ \mathbf{endTrigger}\ D_T, env_T, sto_T \rangle \rightarrow_{DT} (env'_T, sto'_T)}$$

where
 $env_V, env_F \vdash b \rightarrow_b v_{boolean}$

[Trigger-tom_{bss}]

$$env_V, env_F \vdash \langle \epsilon, env_T, sto_T \rangle \rightarrow_{DT} (env_T, sto_T)$$

Behavior

The Transition-system

$$(\Gamma_B, \rightarrow_B, T_B)$$

The configurations

$$\Gamma_{DT} = (\mathbf{ErkT} \times \mathbf{EnvV} \times \mathbf{StoV} \times \mathbf{StoB} \cup \mathbf{EnvV} \times \mathbf{StoV} \times \mathbf{StoB})$$

$$T_{DF} = \mathbf{EnvV} \times \mathbf{StoV} \times \mathbf{StoB}$$

The transition relation

Block Behavior block[Behavior-Block_{bss}]

$$\frac{env_F, env_T, env_B \vdash \langle S, env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto'_B)}{env_F, env_T, env_B \vdash \langle \mathbf{Behavior} S \mathbf{endBehavior}, env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto'_B)}$$

The behavior environment finds the next move of the ant which is evaluated using the entries for the vote, request and force statements in the environment. The environment only contains the first force statement and the request with the highest priority and only if they occur in the environment. The next move will be the action specified by the force statement if there is a force statement in the environment. Vote statements are evaluated by grouping each statement by their action and given a priority equal to the sum of the weights. In case there are no force statements, but a mix of vote and request statements, the statement, either vote or request, with the highest priority or number of votes is chosen. If the priority is equal to the number of votes, the request statement is used.

[Sequential_{bss}]

$$\frac{env_F, env_T, env_B \vdash \langle S_1, env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto'_B) \quad env_F, env_T, env_B \vdash \langle S_2, env'_V, sto'_V, sto'_B \rangle \rightarrow_B (env''_V, sto''_V, sto''_B)}{env_F, env_T, env_B \vdash \langle S_1 \mathit{eol} S_2, env_V, sto_V, sto_B \rangle \rightarrow_B (env''_V, sto''_V, sto''_B)}$$

Request Behavior Request statement[Request-action_{bss}]

$$\frac{env_F, env_B \vdash env_V, sto_V, sto_B [l \mapsto \mathit{maxRequest}(ac, v)] \rightarrow_B (env_V, sto_V, sto'_B)}{env_F, env_B \vdash \langle \mathbf{request} ac \mathbf{priority} a, env_V, sto_V, sto_B \rangle \rightarrow_B (env_V, sto_V, sto'_B)}$$

where

$$env_V, env_F, sto_V \vdash a \rightarrow_a v$$

$$l = env_B \mathit{request}$$

[Request-function_{bss}]

$$\frac{env_F, env_B \vdash env_V, sto_V, sto_B [l \mapsto \mathit{maxRequest}(ac, v)] \rightarrow_B (env_V, sto_V, sto'_B)}{env_F, env_B \vdash \langle \mathbf{request} functionname \mathbf{priority} a, env_V, sto_V, sto_B \rangle \rightarrow_B (env_V, sto_V, sto'_B)}$$

where

 $ac = \mathit{typeRetrieve}functionname$ and

$$env_V, env_F, sto_V \vdash a \rightarrow_a v$$

$$l = env_B \mathit{request}$$

Vote Behavior Vote statement[Vote-action_{bss}]

$$\frac{env_B, env_F \vdash env_V, sto_V, sto_B[l \mapsto addVote(v)] \rightarrow_B (env_V, sto_V, sto'_B)}{env_F, env_B \vdash \langle \mathbf{vote} \text{ ac weight } a, env_V, sto_V, sto_B \rangle \rightarrow_B env_V, sto_V, sto'_B}$$

where

$$env_V, env_F, sto_V \vdash a \rightarrow_a v$$

$$l = env_B vote_{ac}$$

[Vote-function_{bss}]

$$\frac{env_F, env_B \vdash env_V, sto_V, sto_B[l \mapsto addVote(v)] \rightarrow_B (env_V, sto_V, sto_B)}{env_F, env_B \vdash \langle \mathbf{vote} \text{ functionname weight } a, env_V, sto_V, sto_B \rangle \rightarrow_B (env_V, sto_V, sto'_B)}$$

$$env_V, env_F, sto_V \vdash a \rightarrow_a v$$

$$ac = \mathbf{typeRetrieve}functionname$$

$$l = env_B vote_{ac}$$

Force Behavior Force statement[Force-action_{bss}]

$$\frac{env_F, env_B \vdash env_V, sto_V, sto_B[l \mapsto (ac, 0)] \rightarrow_B (env_V, sto_V, sto'_B)}{env_F, env_B \vdash \langle \mathbf{force} \text{ ac}, env_V, sto_V, sto_B \rangle \rightarrow_B (env_V, sto_V, sto'_B)}$$

$$l = env_B force$$

[Force-function_{bss}]

$$\frac{env_B, env_T \vdash env_V, sto_V, sto_B[l \mapsto (ac, 0)] \rightarrow_B (env_V, sto_V, sto'_B)}{env_F, env_B \vdash \langle \mathbf{force} \text{ functionname}, env_V, sto_V, sto_B \rangle \rightarrow_B (env_V, sto_V, sto'_B)}$$

$$l = env_B force$$

$$ac = \mathbf{typeRetrive}functionname$$

Role Behavior Role statement[Role-True_{bss}]

$$\frac{env_F, env_T, env_B \vdash \langle S_1, env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto'_B)}{env_F, env_T, env_B \vdash}$$

$$\langle \mathbf{role} \text{ role}_1, \dots, \text{role}_n S_1 \mathbf{endRole} S_2, env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto'_B)$$

if role_brain_variable = role₁ or ... or role_n

[Role-False_{bss}]

$$\frac{env_F, env_T, env_B \vdash \langle S_2, env_v, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto'_B)}{env_F, env_T, env_B \vdash \langle \text{role } role_1, \dots, role_n \ S_1 \ \text{endRole } S_2, env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto'_B)}$$

if $role_brain_variable \neq role_1$ and ... and $role_n$

On Behavior On statement

[on-true_{bss}]

$$\frac{env_F, env_T, env_B \vdash \langle S, env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto'_B)}{env_F, env_T, env_B \vdash \langle \text{on } triggername \ S \ \text{endOn}, env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto'_B)}$$

where $l = env_T triggername$
 $sto_T(l) = \#$

[on-false_{bss}]

$$env_F, env_T, env_B \vdash \langle \text{on } triggername \ S \ \text{endOn}, env_V, sto_V, sto_B \rangle \rightarrow_B (env_V, sto_V, sto_B)$$

where $l = env_T triggername$
 $sto_T(l) = \#$

Brain Variable Assignment Behavior Variable Assignment statement

[Brain-Variable-assignment_{bss}]

$$\frac{env_F, env_B \vdash \langle D_V, env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto_B)}{env_F, env_B \vdash \langle type \ variablename := exp \ env_V, sto_V, sto_B \rangle \rightarrow_B (env'_V, sto'_V, sto_B)}$$

where exp is the expression of the type $type$
 $reevaluatetriggers$

COMPILER DESIGN

This chapter will discuss the design of the AntL compiler. The first section will give a brief introduction to the subject of compiler theory. The following sections will discuss the design of the AntL compiler. Figure 4.2 shows an overview of the AntL compiler design.

4.1 Compiler Theory

A compiler in its basic form usually consists of 4 elements:

- Preprocessor
- Syntactic analyzer
- Contextual analyzer
- Code generator

Preprocessor

The preprocessor is used to prepare the input code for syntactical analysis. This is done by executing the specified preprocessor commands and in some cases removing comments from the code. Some of the typical preprocessor commands are:

- Include commands to include files
- Macro definitions

Syntactic Analyzer

The purpose of the syntactical analyzer is to prepare the data for the contextual analyzer and the code generator. The syntactical analyzer first translates the input code into a sequence of **tokens**. This process is called the lexical analysis. Tokens are the individual elements in the code, such as identifiers, keywords or operators. The sequence of tokens is then parsed to discover the phrase structure. Finally a representation of the token structure is created and stored usually in an abstract syntax tree (**AST**). The nodes in an AST represent the structures in sequence of tokens. These structures are usually referred to as **productions**. The productions in the token sequence may consist of tokens or other productions, which again may consist of other productions. For this reason, nodes in the AST may contain other nodes and tokens are represented as leaves in the AST.

The AST stands in contrast to the concrete syntax tree (**CST**), which is directly based on the grammar of the language and includes all the productions and tokens. The AST is an abstraction of the CST, which is created by transforming the grammar. These transformations remove unnecessary productions and tokens, which are only needed during lexical analysis and

parsing. When creating the AST a simpler representation is preferable, as the lexical analysis and the parsing phase have been completed and the validity of the productions has been verified.

Contextual Analyzer

The purpose of the contextual analyzer is to assure that the input from the syntactic analyzer contains no error before passing it on to the code generator. During contextual analysis the compiler verifies that all variables and functions have been declared and initialized before use. When traversing the AST the contextual analyzer may **decorate** (add additional information to) the AST in order to make code generation easier. Furthermore the contextual analyzer checks whether additional constraints, which may be more difficult or even impossible to specify in the syntax, are satisfied.

Code Generator

The code generator generates the output code for the various nodes in the AST. This code generation is done by using the AST and a set of code templates. The code generator does not necessarily create output code for all of the nodes, as some of the nodes may be used only to determine how some of the other nodes should be translated.

4.1.1 Parsing Theory

Passes

One of the design decision that have to be made when creating a compiler is whether the compiler should make a multi-pass or a single-pass over the AST. The difference between the two options is the number of times the compiler traverses the AST, as seen on figure 4.1.

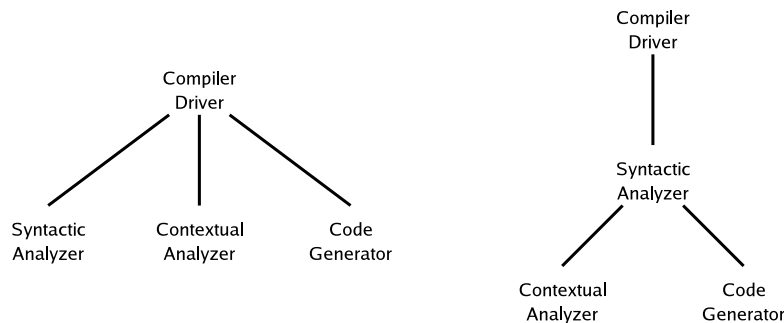


Figure 4.1: Compiler passes; multi-pass to the left, single-pass to the right

A multi-pass compiler traverses the tree in each of its components. The syntactic analyzer creates the AST, the contextual analyzer decorates the AST and the code generator uses the decorated AST to produce output code. A multi-pass compiler favors modularity because each of the compiler components is responsible for a single function and flexibility because each compiler part may traverse the AST in any order.

A single-pass compiler only goes through the input once, while the syntactic analyzer generates the AST. The contextual analyzer and the code generator are performed while the syntactic

analyzer makes its pass through the code. A single-pass compiler is generally faster since it only makes a single pass over the input program.

The choice between multi- and single-pass compilers depends on the nature of the source language. For example, if all variables/functions or similar productions need to be declared before use, a single-pass compiler is often sufficient and would also result in the advantage of faster compilations. If the syntax does not require the productions to be declared before use, multiple passes are needed in order to verify that the use of such productions is valid.

Parsing Strategies

The syntactical analyzer creates a parse tree for the given program and checks for syntax errors while doing so. There are two ways of building the tree; top-down and bottom-up. To explain how these different approaches work and hence the difference between them, the following notation will be used:

- Tokens (a, b or c)
Syntactical constructs of the language, e.g. "if" and "int"
- Productions (A, B or C)
Equivalent to statements, e.g. <if statement> and <expression>
- Unknown symbols (α, β, γ)
Symbols that haven't been parsed yet

Top-down The most common top-down parser parses using an algorithm called LL in order to determine the structure of token sequences. LL is short for: Left-to-right scan of input and Leftmost derivation. top-down parsers build the parse tree in pre-order, which means a top-down parser starts with the root node and determines what productions it can contain. In the next step the parser goes through the possible productions and determines what productions they can contain recursively. The parser always examines the leftmost unknown node in each of the productions (leftmost derivation), and the step is repeated for all productions until a production is found in which the leftmost node is a token (and thus actually a leaf).

If the next token in the token sequence is equal to the token expected in the production, the parser knows one of the productions. This information is used in combination with the next token from the token sequence to determine other productions, until eventually all productions have been determined.

If the token does not match the expected token, other possible productions must be checked. If all possibilities have been exhausted and the token does not match any of the expected tokens, a syntax error has been discovered.

The process is illustrated in the following example. The example assumes that the parser has recognized the first part of the token sequence and determined that the next production should be a production of type B. This means that the parser examines a token sequence of the following form:

$$aB\alpha \tag{4.1}$$

The parser now has to determine which production B is, which means what grammar rule with B on its left side should be applied. The example will assume that the following two grammar rules with B on the left side exist:

$$\begin{aligned} B &\rightarrow abC \\ B &\rightarrow b \end{aligned}$$

The parser uses the next token to determine which grammar rule to use. This means that the first rule should be used if the next token is an *a*, but the second rule should be used if the next token is a *b*. If the next token is neither an *a* or a *b*, a syntax error has been discovered.

There are two common LL algorithms, a recursive-descent parser and a parser that constructs a parsing table. In most cases the recursive-descent algorithm is used because it is easier to implement, since the parse tables of LL parsers tends to be very large.

A recursive-descent parser consists of a collection of subprograms for each nonterminal symbol in the grammar. It is called recursive because the subprograms often are called recursively by each other and descent because it creates a parse tree in top-down (descending) order. Grammars for a recursive-descent parser are written in EBNF¹ which eases the process of creating grammar that can contain recursive tokens.

Parsers using an LL algorithm do have some limitations concerning recursive defined productions. For example, the grammar below would be impossible to handle using an LL algorithm:

$$A \rightarrow Ab \tag{4.2}$$

The grammar would cause an infinite loop in a recursive-descent parser, because the subprogram for A would make a call to itself recursively and thereby create an infinite loop. The grammar below would cause a similar problem.

$$\begin{aligned} A &\rightarrow Ba \\ B &\rightarrow Ab \end{aligned}$$

Bottom-up This type of parsers creates a parse tree from the leafs to the root. Most bottom-up parsers make use of an LR algorithm, where LR is short for Left-to-right scan of input and Rightmost derivation. In contrast to a top-down parser a bottom-up parser begin with reading the token sequence. The parser determines what possible productions it can expect by reading the leftmost token and looking at already recognized tokens/productions. The expected following productions/tokens are called the handles. If the following production/token is not one of the possible handles, the parser tries to apply one of the grammar rules to transform the left side (of the current token/production) to another production. If there is no matching handle and no grammar rules can be applied, a syntax error has been discovered. All found productions are merged in the same way, until at last only the root node remains and the whole tree has been created.

Bottom-up parsers are able to accept grammar rules as in rule 4.2 and thus able to handle left recursion. This is because the bottom-up parser knows when it should stop applying a rule

¹Extended Backus-Naur Form

recursively. The bottom-up parser knows this by reading the next token in order to determine the next possible productions, instead of creating a list of possible productions and trying to verify them.

The following example will show how a LR bottom-up parser works. The example will assume the following grammar rules:

$$A \rightarrow A + B | B \quad (4.3)$$

$$B \rightarrow B \times C | C \quad (4.4)$$

$$C \rightarrow a \quad (4.5)$$

The grammar rules above are applied to the input string $a \times a$. The rightmost derivation can be seen applied below:

$$\begin{aligned} A &\Rightarrow \underline{B} \\ &\Rightarrow \underline{B} \times \underline{C} \\ &\Rightarrow B \times \underline{C} \\ &\Rightarrow B \times \underline{a} \\ &\Rightarrow \underline{C} \times a \\ &\Rightarrow \underline{a} \times a \end{aligned}$$

The example should be read bottom-up, as a bottom-up parser actually performs the reverse of a right-most derivation. The parser shifts the a onto a stack and applies rule 4.5. The next token is the \times token. As there is no rule with $C \times$ on the right side, the parser will try to reduce (transform) the left hand C . This can be done by applying rule 4.5, which results in $B \times a$. The process is continued until the parser arrives at the root node A .

Bottom-up parsers are often called shift-reduce algorithms as they are based on shift and reduce operations.

Shift-Reduce and Reduce-Reduce Problems The grammar for a bottom-up parser must be written carefully as it possible to create a grammar that results in shift-reduce and reduce-reduce problems.

The shift-reduce problem arises when the parser can both shift another token onto the stack and reduce the current stack. The parser has no way of knowing which operation should be applied.

The reduce-reduce problem arises when the parser reads the next token and it has two possibilities to reduce the stack, the grammar is ambiguous. The following example illustrates the problem

$$\begin{aligned} A &= B \ c \ | \ C \ c \\ B &= ab \\ C &= ab \end{aligned}$$

In the above grammar the parser cannot know whether it should reduce its stack with B or C if an A is encountered.

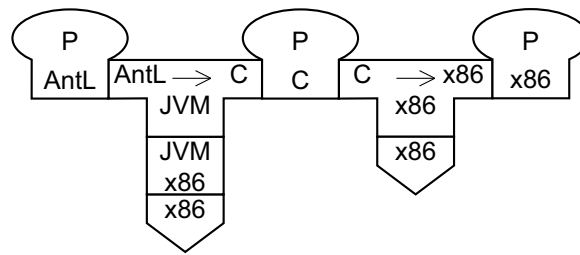


Figure 4.2: Tombstone diagram over compiler process.

4.2 Tools

When creating a compiler it is important to decide which tools should be used. This section will describe some of the available tools. The tools fall into the three categories lexer generators, parser generators and compiler compilers.

Lexer Generating Tools

The lexer tools were one of the first tools to aid the compiler writers. These tools will give a functioning lexer given a grammar. Common examples of lexer producers are Lex, Flex and JLex.

Parser Generating Tools

Having the source file split into tokens by the lexer the next step was to produce a tool that would aid in the construction of the parser. Common examples of parser generators are Yacc, Bison and CUP.

Compiler Compiler Tools

Having to specify two input files (one for each tool) and ensuring that both conform to the same language specification became too tedious and the compiler compiler was invented. The compiler compiler will produce both a lexer and a parser given a single language specification file. Common examples on compiler compilers are SableCC and JavaCC.

4.2.1 The Choice

It was decided early on in the project that choosing one of the compiler compilers was the only viable option, since this would make compiler generation easier. We chose to use SableCC because the grammar was deemed to be easier to read, as it was more similar to EBNF than javaCCs grammar. JavaCC also suffers from debugging issues since it always requires the programmer to go back to the specification file whenever the programmer wants to change some action code. SableCC separates action code from syntax and is thus easier to debug and maintain.

It was decided to use the alternate build of SableCC, made by Indrek Mandre [9], which also generates a main class. This main class uses the lexer and parser to build the Tree. The alternate version was chosen as it appeared to have more bugs fixed than the official version.

4.2.2 SableCC

SableCC [16] is a relative new compiler compiler that provides the programmer with a framework for a compiler instead of producing a complete compiler. SableCC takes one specification file as argument which contains the grammar productions and lexical definitions of a language. SableCC uses this file to generate a highly modular object-oriented framework based on the abstract syntax tree of the input program. SableCC produces the following packages:

- **lexer**

The Lexer package contains the Lexer class, which is a deterministic finite automaton based lexer. The lexer of SableCC works like any other lexer; it ensures that the input file follows the language specification. It does have an advantage compared to most other lexers since it has full (16-bit) unicode support. The package also contains the Lexer Exception class which is thrown whenever the lexer encounters an unknown string in the input file. The lexer generates a set of tokens and passes them on to the parser on a successful run.

- **parser**

The parser package contains the Parser which takes the tokens generated by the lexer and builds an AST. SableCC generates a bottom-up parser using the Look-Ahead LR (LALR) algorithm. It has been shown that LR algorithms are able to recognize a larger set of grammars than LL algorithms. The LALR algorithm can recognize a somewhat smaller subset of grammars than other LR algorithms but uses less memory and a smaller parsing table and is thus faster. For this reason LALR parsers are usually preferred. All LALR grammars can be rewritten into a LR(1) grammar, which can recognize a larger set of grammars.

- **nodes**

The node package contains all the Nodes in the AST. Each class in this package is strictly typed because of javas type system.

- **analysis**

The analysis package contains an interface and two treewalkers. The treewalkers are Depthfirst and ReversedDepthfirst based and can be extended by another class to walk through the AST.

SableCC input

SableCC takes a context free grammar / a EBNF grammar that supports the use of *, ? and +, whereas () only is partly supported. This section will explain how the grammar file looks using figure 4.3 which is a simple example. The parser described by that grammar is able to recognize input like "begin aaBaaDGdhsshjE end", but not "begin end" or "begin 1452 end".

```

Package somewhere.somepackage;
Helpers
    letter = ([ 'a'..'z' ]|[ 'A'..'Z' ]);
    newline = (32 | 9)* (10+ | (10 13)+ | 13+);
    space = 32;
    tab = 9;

Tokens
    begin = 'begin';
    end = 'end' newline;
    string = letter+;
    whitespace = ( space | tab )+;

Ignored Tokens
    whitespace;

Productions
    program = begin string end;

```

Figure 4.3: The structure of a SableCC grammar file

Helpers simplify the creation of tokens. They can contain anything that a token also could.

Tokens are lexical definitions of combinations of numbers, arithmetic operators, parentheses and blanks. Tokens can use EBNF in their definitions.

Ignored Tokens are ignored by the lexer and will not make its way to the parser

Productions are the actual language syntax. Productions can't use () in the syntax, but they can use all other EBNF extensions.

New grammar

The grammar in figure 4.3 is matched by SableCC version2, but version 3 is in Beta stage and with the new revision comes a very practical feature. Version 3 makes it possible to specify the AST in the grammar file instead of just a CST. This is done by transforming the productions and hence create a new production as in the following example:

```

Productions
    program = begin string end { -> New program(string) }

Abstract Syntax Tree
    program = string;

```

This only affects the AST and not the syntax! The difference between the visual representations of this grammar when given the same input can be seen in figure 4.4 and figure 4.5. This

feature is very useful and AntLs syntax uses it extensively (see appendix E).



Figure 4.4: The CST representation



Figure 4.5: The AST representation

4.3 Compiler

4.3.1 Input

The compiler takes the source code that the developer wants translated to the destination output language as an input. In our case the output language is C and the input language is AntL. When the compiler has read the input source code, either from standard in stream or from a file, it sends the input to the preprocessor.

Furthermore the input handling reads compiler arguments, set when running the compiler from the command-line prompt and sets internal compiler flags for each specified compiler argument.

Compiler argument: `-warnings`

As described in section 4.4 the compiler supports different types of error reporting. One type is reporting minor errors as warnings, which developers can show or suppress as wanted. The `-warnings` compiler argument turns on error reporting of warnings, which is suppressed otherwise.

Using the `-warnings` argument

```
$ java -jar antlang.jar rambo.ant -warnings
Checking syntax...done
Checking contextual...warning
Identifier variable pos3 was not assigned a value
when leaving scope at line 23.
```

Compiler argument: `-outputfile`

The compiler defaults to using `filename.c` if the input filename is `filename.ant`. To override the default output filename the developer can specify an alternative filename. The filename is specified in the subsequent argument as illustrated below:

Using the `-outputfile` argument

```
$ java -jar antlang.jar rambo.ant -outputfile
rambo2.c
```

Compiler argument: `-tree`

If the compiler is called with the `-tree` argument it displays a graphical representation of the abstract syntax tree generated by the syntactical analysis. This argument is mainly useful for the compiler developers, but can as well be used by ant-developers to debug compiler behavior.

Using the `-tree` argument

```
$ java -jar antlang.jar rambo.ant -tree
```

4.3.2 Preprocessor

The preprocessor handles all preprocessing of the input text, so the lexer can start processing the text. The preprocessor strips comments from the input and import function libraries written in AntL. The implementation of the preprocessor will be described in section 5.1.

4.3.3 Syntactic Analysis

The two steps in the syntactical analysis are automatically generated by SableCC as described in section 4.2.2, and no special design or implementation of these two will be described further.

4.3.4 Contextual Analyzer

The second step of the compiler process is the semantic analyzer, where the abstract syntax tree is checked for type errors, use of undeclared variables and functions. When the compiler progress has completed the contextual analyzer phase, the compiler must have an error-free set of data to pass on to the code-generator.

Identification table

To perform type checking and validation of variable and function use, an identification table is built, which stores variable and function names and types and in which scope they were declared.

The identification table consists of a table where each row is an entry matching a variable or function declaration in the abstract syntax tree.

The entries contains the following attributes:

- **Name** - Name of the entry
- **Type** - Type of the entry, either variable, trigger or function.
- **Evaluates to** - Variable type or function return type. This is used when type checking. Can be one of the following types: integer, boolean, position and direction.
- **Scope level** - Numeric representation of the scope in which the entry is found.
- **Initiated** - Boolean value. True if variable has been assigned a value.

Name	Type	Evaluates to	Scope level	Initiated
heading	variable	direction	0	true
distance	function	integer	0	true
FoodLeft	trigger	boolean	0	true

Table 4.1: Example identification table

Each time the analyzer enters a new scope it increases the internal scope level, and decreases it again when it leaves the scope. Any entries declared in that scope will be removed from the identification table upon leaving the scope. The use of a static/static scope doesn't allow

variables inside e.g. a function body to access variables declared accessible from the scope where the function was called, so the compiler only needs to handle scopes when entering and leaving block structures. Table 4.1 is an example of how the identification table looks when the brain, function and trigger blocks have been indexed in the code example shown in listing 4.1.

```

Ant "TestAnt"
  Brain
    position heading
  endBrain

  Functions
    Function distance(position pos1, position pos2) returns integer
      return (pos1.x-pos2.x) + (pos1.y-pos2.y)
    endFunction
  endFunctions

  Triggers
    Trigger FoodLeft
      numberFood left > 0
    endTrigger
  endTriggers

  Behavior
    ...
  endBehavior
endAnt

```

Listing 4.1: Code example matching the table 4.1 identification table.

Checking for Declarations When a new function or variable is declared, a lookup is performed in the identification table to see if an entry of the same name and type is already declared in the same scope. Since functions can only be declared in a single scope, the functions block, only functions with unique names are allowed. Variables can override variables of matching names, if they are declared in a new scope, since lookups are performed from the current scope and onward.

Checking for Initiated Variables When a variable is declared it is entered into the table and set to be uninitiated. It can't be used in expressions (comparisons and assignments) until it has been initialized by assigning it a value. Variables in the brain block and function parameter are automatically set to be initialized. Assigning values to x and y-subvariables of position variables will initialize them individually, and when both are initialized the position variable is initialized.

Type Checking

Type checking is the task of evaluating expressions and check if they match the required type in the context. Expressions need to be type checked bottom-up to ensure that each subexpression has the right type.

$$\text{boolean } a := \text{true and (distance() > 10 or atBase())} \quad (4.6)$$

Statement 4.6 shows a primary expression, containing several subexpressions.

a	→	identifier
true	→	boolean value
distance()	→	function call
10	→	integer
atBase()	→	function call

After having identified the literals and identifiers, the type of identifiers such as variables and function calls is looked up in the identification table.

a	→	boolean
true	→	boolean value
distance()	→	integer
10	→	integer
atBase()	→	boolean

Expression operators² with two surrounding literals and identifiers are then used to check whether the types of T_1 and T_2 match and form a new subexpression determined by the operators. Comparison operators result in the parent expression being a boolean expression and arithmetic operators result in an expression with type integer or position depending on the types on each side of the operator.

$$\text{distance() > 10} \rightarrow \text{boolean expression}$$

The bottom-up evaluation of the type of a primary expression results in an expression-type-tree as shown in figure 4.6.

Assignments AntL has two types of assignments, variable assignments and variable declaration assignments. The difference is that when assigning a value to a variable that is declared earlier, the type has to be looked up in the identification table and the type checking has to be performed. With variable declaration assignments the type is immediately known. On the way out of the assignment node in the tree, all subexpressions have been evaluated and the type of the main expression has been determined. The type checking then only involves comparing the type of the expression with the variable type.

²and, or, >, <, =, != are comparison operators. +, -, *, / and % are arithmetic operators.

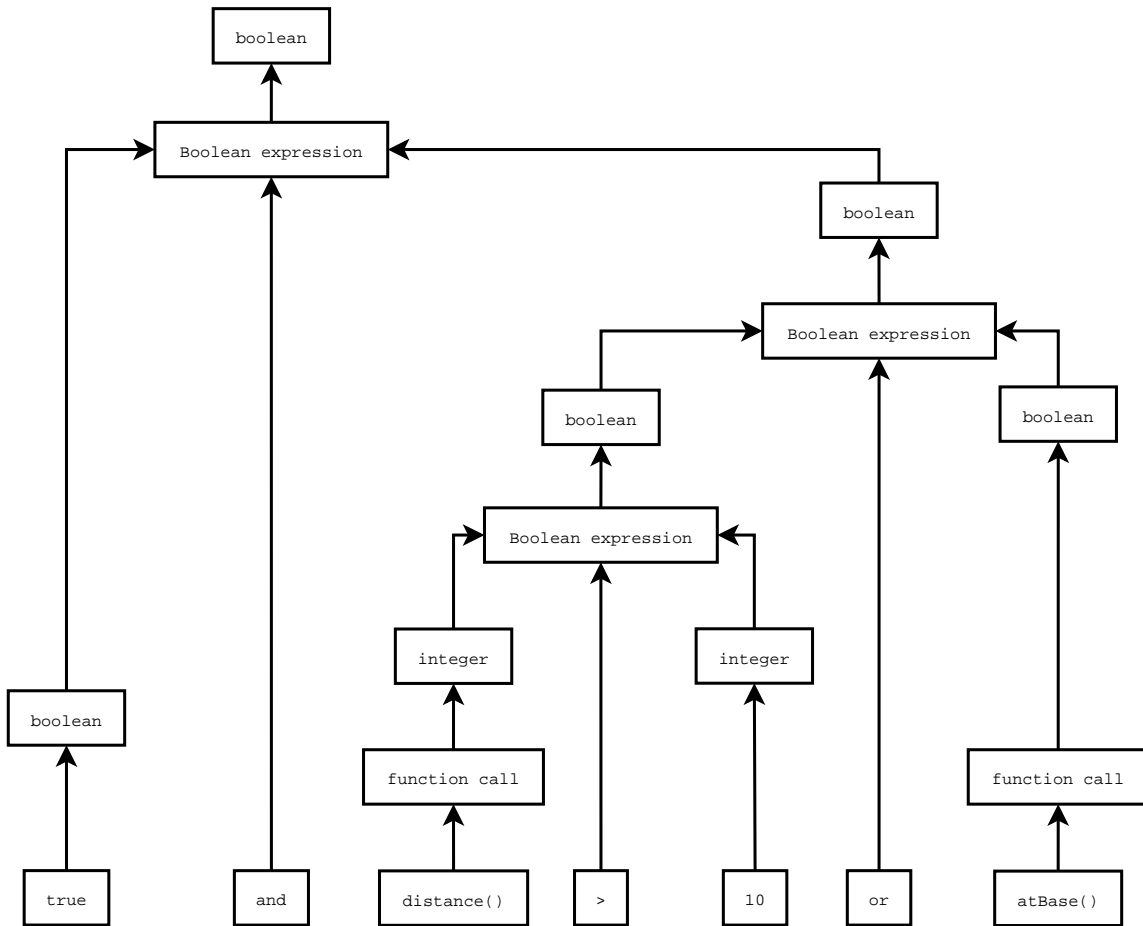


Figure 4.6: Expression tree for statement 4.6

Function Calls When a function is called, it is necessary to check whether the parameter count matches the function declaration. It is necessary to loop through all parameters in the function declaration and in the list of parameters in the function call and check whether the type of each function call parameter matches the type of the same function declaration parameter.

Additional Information Gathering

When moving through the AST the contextual analyzer saves information in the AST to decrease the work of the code generator. The extra information includes

- **Role list** - As described in section 3.3.4 it is possible to specify a default role and if not specified the first role encountered in the AST will be set as the default role. The default role is entered as the first entry in the list of roles, and then each occurrence of a role is entered subsequently.
- **Call of framework functions** - The contextual analyzer sets a flag if certain framework functions is called and the code generator then only writes the functions used.

4.3.5 Code Transformations

Code transformation is the task of optimizing the output code for a more optimal execution time. In most cases the optimization results in faster execution code, but slower and more complex compilation process. As described in chapter 4 the compiler functions as a translator between AntL and C and thus another compiler is needed to produce executable object code. In the case of MyreKrig the commonly used compiler is GCC³ and as most industrial strength compilers it effectively performs code transformations. Most of the transformations the AntL-compiler could produce will be performed by the C-compiler.

Constant Folding

Constant folding is the process of folding constants in an expression. In doing so there will be less operations that must be done in the object-code. An example in java style syntax could be.

```
public int x = (2 * 2 + 1) * x
```

This simple expression has three arithmetic operations. But using constant folding it can be reduced to just one:

```
public int x = 5 * x
```

In folding the constants information has been removed from the expression which can lead to more troublesome debugging if the folding is done to the source file. Since AntL is compiled to an intermediate language it was decided that there was no need for the AntL-compiler to do constant folding, for several reasons. One reason is that it would only obscure the translated code and that AntL was designed so that the programmer easily could make whatever adjustment he deems necessary to the generated code. Another reason as to why constant folding was discounted is that in the end the ants developed in AntL are to be compiled by gcc which can do all the constant folding that is desired. Furthermore gcc can do it at no cost to the readability of the developed ant since the constants get folded in the object code.

³GNU C and C++ compiler

Common Subexpression Elimination

The idea behind common subexpression elimination is that if an expression already has been evaluated and is used again later it is sufficient to remember the value the expression so that there is no need to reevaluate the expression. Care should be taken when using this type of optimization since the compiler needs to be aware of any changes in the code.

$$((x / 2) * y) * ((x / 2) * y) * x$$

In the example above there is no need to evaluate the subexpression $(x / 2) * y * z$ twice since the value of it has not changed between its first and second use. Since AntL is compiled into C and the nature of this optimization is best used on object code there was no need for subexpression elimination in the AntL compiler.

Code Movement

Code movement involves changing the code flow of a program, and this transformation type should not be applied to a program before the code base has matured. This is because resolving the original source code line number is a complex task if runtime errors occur or when stepping through the program in a debugger.

As with the other code transformations code movement transformations can be performed on the source code by the C-compiler.

4.3.6 Code Generator

The code generators role in the compiler is to generate the actual output code using the AST and the information gathered by the contextual analyzer. In the case of the AntL compiler the output must be C code matching the requirements set by the rules of MyreKrig.

Design Decisions

When designing the code generator it was clear that special attention should be taken regarding declaration of variables. This was due to the fact that in ANSI C all variables in a particular scope must be declared at the very start of the scope, before any of the variables are used. This is however not the case in AntL, so a splitting of the declarations and the rest of the code had to be created. This separation was done using two buffers. One buffer containing the declaration and the other containing all other code.

Structure of the Output File

Another decision regarding the code generator was how to organize the output file. An outline of the output was created with regard to the placement of structures and utility functions needed to support the functionality of the AntL, and a strategy of how to create the actual content of these was created.

The output file is furthermore organized in to a number of sections in order to improve the readability of the c-code. These sections are the following:

- Preprocessor commands

- Data type declarations
- Declaration of structures
- Utility functions
- User defined functions
- Trigger evaluation function
- Ant main-function
- Preprocessor commands

These sections are clearly marked in the output code with comments, in order to improve readability and to assist the editing of the output code (if a user wants to optimize the ant using C code)

Preprocessor Commands The preprocessor commands are the same in all ants generated by the code generator, they consist of the following code:

```
#include "Myre.h"  
#undef true  
#undef false  
#define drag 8 +
```

This code includes the file Myre.h and undefines the macros named "true" and "false" which are defined in Myre.h, (a component of MyreKrig) and defines a new macro called drag, which is assigned to the value "8 +".

The reason true and false are being undefined are that they are used as values of the datatype boolean as defined later.

Data types In this section of the output code a set of data types is declared. These data types are used to make the code more readable by hiding the underlying data representation. the defined data types are the following:

- position
- boolean
- direction
- action
- role

Most of these data types are rather trivial and do not vary between ants generated by the compiler, but they improve the readability of the output code. For instance the "position" data type is made to match the position data type in AntL and consists of a structure containing the x and y coordinate of the position as integers. This makes positions from AntL easily recognized when looking at the output code.

A data type that varies between ants and functions as more than just a mapping is the type role. Although the role type seems to be nothing more than a mapping of the role names, the order of the role names is important. The role mapped to "0" has to be the default role of the ant, since brain variables are initialized to the value "0" when a new ant is created.

Declaration of Structures In order to prevent collisions with other ant race's function and struct names, these are all prepended with the ants file name. This is illustrated in this and following paragraphs by using <AntName>.

The following structures are defined:

- <AntName>Brain
- <AntName>EnvBehavior
- <AntName>Trigger

Brain This is the structure used to hold the brain of an ant. The structure is called <AntName>Brain and consists of the variable declared in the brainblock in AntL. Furthermore if roles are used in the ant a variable is made in order to store the role. And in a similar manner if the ant uses the Random function a variable to store the random seed is set aside.

The actual content of the structure is a mapping of the brain variables defined in AntL with the addition that the variable role myRole is added if the AntL program uses roles, and that the variable random, is added if the user uses the random function.

EnvBehavior This is a structure used to store information about regarding the vote, request and force statements in AntL. In C the structure is called <AntName>EnvBehaviour. This information is later used by the evaluateAction function to determine which behavioral action should be performed.

Triggers This is a structure containing the value of each trigger, as boolean values. The actual value is computed using the evaluateTriggers function.

Utility functions In order to support some of the more advanced features of our language, this being voting and requesting, the output file contains a number of utility functions.

These functions are used to vote upon, request or force an action or a role change. Other functions are then later used to evaluate these votes and requests and determine which action/role change has to be performed.

Apart from functions used in the vote/request system, which are inaccessible to the user in AntL there is a set of helper functions provided which are used to simple tasks such as adding two positions or getting a pseudo random number.

User defined functions This block contains the functions the user has defined in the AntL source file. The functions have similar return types and arguments as defined in the source file, with the addition of pointers to the brain- and trigger structures, since these contains variables that should be accessible by a function.

Trigger evaluation function This is the function which calculates the value of each trigger. The trigger values ("true" or "false") are stored in a struct in order to logically group the information and to ease the access to these from functions.

Main function The main function of an ant consists mostly of the behavior and finalize blocks translated from AntL to C. In addition to this there is some initialization code (initializing structures and variables), and some code to evaluate the actions and role changes before running the code translated from the finalize block.

The initialization code resets the votes/requests and evaluates the triggers. This evaluation of triggers is further more done each time a brain variable is changed, since triggers can be affected by the state of brain variables.

Preprocessor commands The final part of the output file is a set of preprocessor commands:

```
#undef drag
#define false 0
#define true (!false)
DefineAnt(<AntName>, "<screenName>", <AntName>Main ,
        struct <AntName>Brain)
```

The first line undefines the drag macro, and the next two recreates the false and true macros to the state defined in Myre.h. Finally the last line links the ant ant to the MyreKrig program, in which <screenName> is the name shown in the interface.

4.4 Error Reporting and Handling

Each analysing part of the compiler can cast an error and either try to handle the error or instead immediately exit with an error message. In this section the two different error handling methods will be described with pro and cons, and how the different parts of the AntL compiler can facility the different methods.

4.4.1 Syntactical Errors

The syntactical part consists as described in section 4.3.3 of two sub-parts, the lexer and the parser. If the lexer encounters a character-sequence it can not map to a token it has the possibility to make an attempt at predicting which token the user intended, or instead mark the token as an error-token. Finally it can use the quick-exit method and immediately display an error message to the programmer and exit. Predicted tokens should result a warning, but still resume normal execution. Ignored tokens is skipped by the parser to allow for several error messages, when the syntactic analysis is completed the compiler should exit.

4.4.2 Contextual Errors

The contextual analyser can use a similar approach as the syntactical analyser and display as many error messages as it encounters and terminate the compilation process.

Non-critical errors, declaration of unused variables and changing to behavior roles that are not specified, should result in a warning message to inform the programmer of the possible error and it is then up to the programmer to determine if the error message should be ignored.

4.4.3 Runtime Error Reporting and Handling

Checking for runtime errors as arithmetic overflow, division by zero and out of bounds array access requires checking of expressions almost constantly and will decrease performance of the program and as described in section 2.2.1 this will decrease the score of an ant.

The runtime error reporting and handling would consist of checking each sub-expression before letting the expression be executed. Out of bounds access of array entries error handling could be implemented in AntL by checking that the program can maximum access index `friendAnts here - 1` of the communication variable array.

IMPLEMENTATION

This chapter revolves around the implementation details of the different parts of the compiler. The focus will be concentrated on describing general strategies used in each compiler component and on parts which can be described as non trivial or as crucial for the functionality of the component in question.

5.1 Preprocessor

The general layout of the preprocessor is shown below in pseudo code.

- line \leftarrow first line of input
- while (line \neq NULL) do
 - Remove comments from line
 - Truncate lines only containing whitespaces
 - if (line contains import statement) do
 - * Check file exists
 - * Copy external file content into input code
 - * Strip comments from imported code
 - input \leftarrow input + line
 - line \leftarrow new line of input
- Return input

The different sub tasks in the preprocessor pseudo code is described further in this section, including how they were implemented in the AntL-compiler.

5.1.1 Stripping Comments

We have chosen to remove comments from the input source file before passing it on the syntactic analyzer. In languages like java comments can have a special meaning and be used to generate code documentation, although they are still ignored by the java-to-bytecode compiler. Since the code generator of the AntL-compiler places C-code different places in the output file it would not be trivial to place original comments in the C-code, furthermore comments written in the AntL-code could be disabled statements which would make no sense to translate into C-comments.

Lines starting with none or more whitespaces (tabs and spaces) followed by a double slash // and a optional string is truncated to an empty line. Lines are truncated instead of removing them to preserve line numbering so the developer can easily find the line containing the

error in the original source file given the error message supplied by the compiler. If the comment is written after a statement, the line will be truncated to the statement. The java method `stripComments()` shown below executes a search and replace regular expression on the input line.

```
public static String stripComments(String input) {
    return input.replaceAll("[\t ]*\\|\\|\\|/(.*)", "");
}
```

The following two code examples contain a function written in AntL. The first example is with comments and the second is the result output code after the `stripComments()` method has been called on each line.

```
1 // This function calculates the distance between two positions.
2 Function distance(position pos1, position pos2) returns integer
3     ... // Some calculation
4 endFunction
```

When the above code is stripped for comments, the first lined is emptied, the third is truncated to the statement.

```
1
2 Function distance(position pos1, position pos2) returns integer
3     ...
4 endFunction
```

5.1.2 Truncating Lines containing Whitespaces only

Lines containing nothing but whitespaces is truncated using a single regular expression replacing lines with tab characters or spaces with an empty line.

```
line = line.replaceAll("[\t ]+$", "");
```

5.1.3 Importing External Libraries

As described in section 4.3.2 the preprocessor also handles inclusion of external libraries. This is implemented by opening the import file and then copy the file content into the source code from the main source file inplace of the import statement.

```
// Import libraries if one is included
if (line.matches("[ ]*import (.+)")) {
```

```
String filename = line.replaceAll("[ ]*import[ ]+", "");
File f = new File(filename);
if (!f.exists()) {
    System.err.println("ERROR: Import file \""+filename
        + "\" not found!");
    System.exit(1);
}
```

The line-numbering is changed when copying the external libraries into the main source code, thus line-numbers after import statements are no longer comparable to the original file. Solving this problem would require mapping line numbers to files and modifying the error reporting to lookup the original linenumber and filename based on the line number in the complete source code.

5.2 Contextual Analysis

SableCC generates a class named `DepthFirstAdapter` that walks through the AST. The class contains an in and out method for each node in the abstract syntax tree. The in method is called when entering a node in the tree and the out method is called after a traversal of the nodes subtree. The default behavior of these methods is to call an empty method named `defaultIn()` or `defaultOut()`. To handle entering and leaving different nodes in the tree the in and out methods is overridden in the contextual analysis class, and type checking and identifier lookups can be performed.

5.2.1 Identification Table

The contextual analyzer stores the identification table as an linked hashset containing objects. these objects have the attributes (name, type, return type, scope level and initialized) as described in section 4.3.4.

isDefined() - Checking declarations

The method `isDefined()` checks if a variable, function or trigger is declared in the identification table. The method can check for a declaration in a specified level to validate if a new declaration results in a double declaration. If level checking is not forced the method searches through all entries in the identification table until it either reaches the end in which case it returns false or it finds an entry with the specified name and type. If level checking is forced the found entry is required to have the correct level otherwise the search loop continues.

```

boolean isDefined(
    String name,
    String type,
    boolean levelCheck,
    int level
) {
    for ( Iterator i=symbol_table.iterator(); i.hasNext();) {
        IDTableEntry entry = (IDTableEntry) i.next();
        if (
            name.equals(entry.name) &&
            type.equals(entry.type) &&
            (levelCheck && entry.level == level || !levelCheck)
        ) {
            return true;
        }
    }
    return false;
}

```

The method `isDefined()` is called before entering a new identifier in the identification table, to ensure that no identifier of the same type is present at the current scope level. If an identifier already exists an "identifier already exist" error is returned.

isInitialized() - Checking if a variable is initialized

The `isInitialized()` method is used to check if a variable is initialised by searching through all entries in the identification table. It searches for the entry with the highest scope level and checks if that variable is initialised. Where `isDefined()` is only required to return if an identifier is found, `isInitialized()` is required to find the identifier closest to the current scope since accessing an identifier not initialized at runtime would result in a null-pointer exception.

```

boolean isInitialized(
    String name,
    String type,
    boolean levelCheck,
    int level
) {
    IDTableEntry tempEntry = new IDTableEntry(name, type, "", 0,
false);

    for (Iterator i=symbol_table.iterator(); i.hasNext();) {
        IDTableEntry entry = (IDTableEntry) i.next();

        if (name.equals(entry.name) && type.equals(entry.type)) {
            if (levelCheck && entry.level == level) {
                tempEntry = entry;
                break;
            } else if (
                !levelCheck &&
                entry.level <= level &&
                tempEntry.level <= entry.level
            ) {
                tempEntry = entry;
            }
        }
    }

    return tempEntry.initialized;
}

```

5.2.2 Implementation of Type checking

The type checking is performed as a part of the contextual analyzer by looking up identifiers in the identification table when visiting each node. The main node class that all nodes inherit from is extended with a type information variable and two methods `getType()` and `setType()`. When leaving a subtree of an expression, the type of the expression is set based on the type of any subtree and the operator type as described in section 4.3.4. The type checking can not be performed by leaving the expression nodes subtree, because the type of the subtree is not set before having visited it.

5.3 Code generation

The general strategy of the code generation is to let the treewalker translate each node directly to C code, which is a usable approach for most nodes that have a similar counterpart in C. Nodes without equivalent or similar counterparts in C need some extra work during translation.

The output is not written directly to a file, but is stored in a buffer. The Buffer is sent to a PrettyPrinter class, which formats the C code and writes it to a file. The buffering and some of the nontrivial node methods are described in more detail in the following paragraphs.

5.3.1 Buffering

ANSI C requires variable declarations to be made at the beginning of each block, while it is possible to make variable declarations in the middle or end of a block in AntL. To prevent problems with variable declarations in the generated C code, it is necessary to rearrange the code, so that variable declarations are placed only at the beginning of a block. The treewalker generated by SableCC provides methods for entering and exiting a node, thus not making it possible to rearrange code inside a node. This can be resolved by storing the generated code in separate buffers, which can be rearranged in the desired order.

The code has to be split up into declaration and general code for each block, thus a new buffer object is created when the treewalker enters a new block. The buffer object contains one string for variable declarations and one string for other code. The variable declaration string is set to start with the code denoting the start of the block.

A LinkedList is used to manage the buffers. When the treewalker enters a new block, a new buffer object is created and inserted into the LinkedList. All nodes which do not indicate a new block execute the following three steps (with some exceptions, which will be discussed shortly):

1. Retrieve the last buffer from the LinkedList using the `removeLast()` method.
2. Generate the appropriate code for the node and write it to the appropriate strings in the buffer object. Combined declarations and assignments are split up into a declaration and an assignment statement, which are concatenated to the respective strings in the buffer object.
3. Insert the buffer object into the LinkedList again using the `add()` method.

When the treewalker exits a block, it retrieves the last 2 buffer objects. The first retrieved buffer (block buffer) contains the code generated for the current block, the second buffer (parent buffer) is the buffer of the block encapsulating the current block. A `}` bracket is appended to the code string of the block buffer in order to end the current block. The contents of the block buffer are then concatenated to the contents of the parent buffer, which is then again inserted into the LinkedList. This maintains the correct order of the code, while making it possible to move all variable declarations to the beginning of each block.

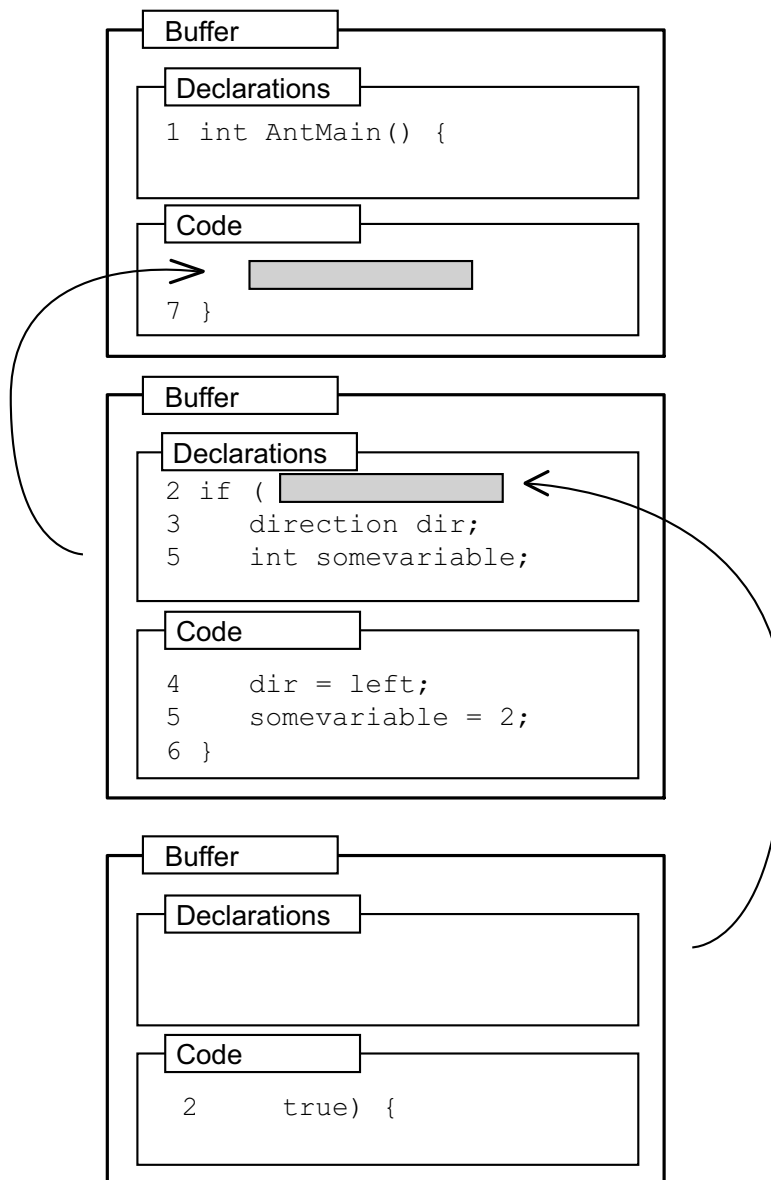


Figure 5.1: Code generator buffer handling. The line numbers indicate which line in the AntL code the line represents. For simplicity the parameters of the main function have been omitted.

```

1 Behavior
2   if true
3     direction dir
4     dir := left
5     integer somevariable := 2
6   endIf
7 endBehavior

```

Listing 5.1: Input code in AntL.

```

1 int AntMain() {
2   if (true) {
3     direction dir;
4     int somevariable;
5     dir = left;
6     somevariable = 2;
7   }
8 }

```

Listing 5.2: Output code after the buffers have been merged. For simplicity the parameters of the main function have been omitted.

The expressions in control structures like 'if' and 'while' are treated as blocks, in order to prevent them from being added to the code string of the current buffer. In addition to the expression itself, the buffer for the expression contains the '{' bracket of the control structure. When the treewalker leaves the expression the buffer is added to the declaration string of the control structures buffer instead of the code string.

The buffering system is illustrated in figure 5.1. Listing 5.1 shows the input code in AntL. Listing 5.2 shows the output C code after all buffers have been merged. The first buffer object is created to store the behavior block. The next buffer stores the if block and the last buffer stores the expression for the if block, as described above. The last buffer is merged into the second buffer when the treewalker exits the expression node, whereupon the remaining code is added to the second buffer. The second buffer is then merged into the first buffer when exiting the if control structure node, and finally the closing '}' bracket is added.

5.3.2 Main Blocks

The node `AProgram` is the first node the treewalker enters when using the depth first adapter. All type definitions and macros used in the output file are created when entering this node.

When entering the brain block node (`ABrainblock`) the treewalker creates the brain struct. All variable declarations in the brain structure are created when the treewalker enters the appropriate nodes for the variable declarations. After creating the brain block, the trigger, role and `envBehavior` structs are created. If no brain block was specified in AntL, the brain structure is created by the `AProgram` node using the `createBrainblock()` method.

When entering the function declaration node (`AFunctionsblock`) the treewalker first writes all C utility functions using the `createUtilityFunctions()` method. As a part of the utility functions, a prototype for the trigger evaluation function is created, since triggers can make use of functions, and some functions may need to reevaluate the trigger variables. All user defined functions are created in the appropriate nodes for function declarations. If no functions block was specified in AntL, the utility function declarations are created by the `ABrainblock` or the `AProgram` node using the `createUtilityFunctions()` method.

When entering the trigger declaration (`ATriggerblock`) node the treewalker creates the C utility function that evaluates the triggers. All trigger declarations in the brain struct are created when the treewalker enters the appropriate nodes for the boolean expressions. If no trigger declaration block was specified in AntL, the trigger evaluation function is created by the `AFunctionsblock` node, the `ABrainblock` node or the `AProgram` node using the `createTriggerblock()` method.

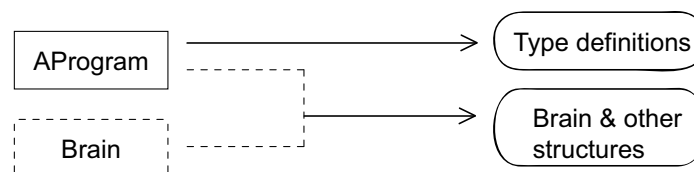


Figure 5.2: If there is no brain block the `AProgram` node creates the brain structure.

The process is illustrated in figure 5.2 and 5.3. The last behavior and finalize nodes create the main function. Many of the nodes under these nodes are directly translated to their counterparts in C. The less trivial parts are discussed in the remaining sections of this chapter.

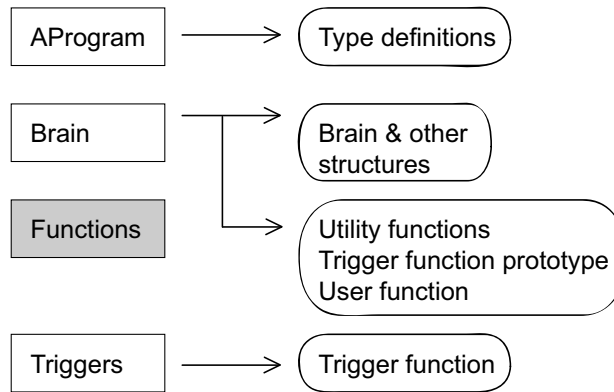


Figure 5.3: If one of the optional blocks does not exist, the block above creates the output code for that block. If this block does not exist either, the block above creates the code. If none of the optional blocks above exists, the AProgram node creates the code for all of these blocks.

5.3.3 Control Structures

The control structures in AntL are quite trivial to translate with few exceptions, as they all have equivalent counterparts in C or can be expressed using `if` statements in C.

A minor complication arises due to the way `if/elseif/else` blocks are nested in the abstract syntax tree. An `elseif` block, for example, is not placed at the same level as the `if` block it refers to, but is instead nested under the `if` block. All `elseif/else` blocks thus end the previous block by writing a `}}` bracket, before starting writing the start of the new block. As this results in a missing `}}` bracket for the last `elseif/else` block, the treewalker writes a `}}` bracket when leaving a `if` block.

Role blocks are treated as `if` blocks, where the expression checks whether the `currentRole` variable has a certain value. If several roles have been specified, the expression checks whether the `currentRole` variable has one of the specified values, by separating the different possibilities with `||`.

On (trigger) blocks are also translated as `if` blocks, where the expression checks whether a certain variable in the trigger struct has the value `'true'`.

5.3.4 Force, Request and Vote

Force, request and vote statements are translated into function calls, where there are separate functions for actions and roles. The functions take the desired action/role and the priority/number of votes as arguments and uses these to manipulate the behavior environment. In addition, the `envBehavior` struct is supplied as argument, which is used to store the force/requests/votes on roles and actions for later evaluation.

5.3.5 Communication

To access the brain variables of other ants, the position of the struct in the array of brain structs must be specified. In AntL the name of the accessed variable is specified before the position in the array is specified, whereas C requires the order to be reversed. This is illustrated below:

Syntax in AntL:

```
<variable name>@friends[<integer expression>]
```

Syntax in C:

```
beliefs[<integer expression>].<variable name>
```

This means that the treewalker must not write the name of the accessed variable before the expression specifying the position in the array has been written. Thus the method `inANormalSimpleVariable()`, which is used to write variable names, omits writing the variable name if the parent node is a `ACommunicationVariable` node. The variable name is first written when leaving the `ACommunicationVariable` node, using the `getSimpleVariable()` method.

The name of the struct-array and the `[]` brackets are written by the expression specifying the position in the array, if the parent node is a `ACommunicationVariable` node.

5.3.6 Environment Variables

Most environment variables depend on both the team and the number of ants/bases on a field. In order to reflect this, the following C construct (called a ternary conditional operator) is used to translate environment variables:

```
<boolean expression> ? <>true statement> : <>false statement>
```

The `<>true statement>` is returned if the `<boolean expression>` is true, otherwise the `<>false statement>` is returned. The following example shows how the environment variable `friendAnts left` is translated:

```
(!(environment[left].Team) && environment[left].NumAnts) ?
    environment[left].NumAnts : 0
```

5.3.7 Positions

Positions values are translated by placing a typecast (`position`) in front of the value and thus casting the variable or integer set to the type position (as defined by the `AProgramStart` node). Positions are added, subtracted and compared for equality and non-equality using functions, taking the two positions as arguments. The following example shows how two positions which are compared for non-equality are translated:

```
<filename>cmpNotPos(<pos 1>,<pos 2>)
```

`<filename>` is the filename of the ant, `<pos 1>` and `<pos 2>` are the positions that should be compared.

REFLECTIONS

This chapter will discuss some of the properties of AntL including some of the underlying concepts.

6.1 Concept and Language Discussion

The features of AntL were designed without much emphasis on the agent theory. This is because most features were added on a basis of what was considered to make ant development easier. Most of the features still have a solid foundation in the theory of agents which can be illustrated with the force, vote and request system. The weight in vote statements and the priority in request statements can be considered as the specification of the expected utility for that action. Since the action with the highest number of votes or priority is performed, the action with the highest expected utility will be performed. This means that the developer in essence specifies a utility function for the ant by using the vote, request and force statements. The fact that this is only a function for the expected utility should be kept in mind, since the real utility of an agent is calculated by MyreKrig as the overall score.

Our own experiences with using AntL during the user testing and through out the project are that the AntL has a rather steep learning curve. This means that it takes some time to get used to the new control structures and to understand the vote/request/force system and how to use them in an appropriate manner. However when one has gotten used to the language the productivity when creating more complex ant races is greatly improved. This increase in productivity is mainly attributed to the introduction of triggers and roles which eliminate much of the confusing control structures that are present in most C ants.

In chapter 3 we presented a set of properties which we would like our language to have:

- Native support for the agent abstraction.
- Easily read control structures.
- Easily read syntax in general.
- Flexibility and expressiveness within the scope of MyreKrig.

These properties have been implemented to the extent that limitations in time and resources allowed it. One of the fields in which the agent abstraction is not possible to use in full is during communication. The communication in AntL and in C doesn't differ much since the execution of ants is strictly serial, more advanced communication models such as broadcasting or ask-and-reply are not possible to implement efficiently.

It is difficult to make any final conclusions on the readability of the control structures and of the language as a whole, apart from our own experiences gathered through the use of AntL. We have found that ants written in AntL are easier to read, especially when dealing with larger ants with multiple roles. How other people would rate the readability is difficult to determine without a large scale user-test with 3th party participants.

6.2 Compiler Discussion

The goals for the compiler were that it should create valid and efficient C code. Through testing we have verified that the generated output code from the AntL-compiler is valid, but due to time limitations we have not implemented the optimizations that could improve the score in MyreKrig. Some of these optimization techniques are described in the chapter 7.

A feature that would improve the user's experience of the compiler and a feature that contextual analyzers of most compilers support is the ability for the contextual analyzer to keep reading tokens even though an error has occurred. This approach will give the programmer more than the first error the contextual analyzer meets as described in section 4.4 on page 51.

FUTURE DEVELOPMENT

In this chapter some of the possible extensions and additional features of the language and of the compiler will be presented. Some of these extensions were discussed during the planning phase of the project and eventually rejected due to time limitations or because they would fit better in a second iteration of the language and compiler.

7.1 Optional Runtime-checking

In section 4.4 the pros and cons of runtime error-checking and -reporting were discussed. If a developer experiences problems during execution runtime-checking and debug-information could be a valuable tool. An extra compiler argument `-runtimechecking` can be added and when set the code generator writes runtime-checking code to the output code. When the errors have been removed and the program is going to be submitted to the official tournament as described in section 2.2.3 the runtime error-checking and debug information can be removed by recompiling the program without the `-runtimechecking` argument.

Example of use

Using the `-runtimechecking` argument

```
$ java -jar antlang.jar rambo.ant -runtimechecking
```

7.2 Variable Types

AntL supports 5 different variable types (integer, boolean, position, direction and action) in a future iteration it would be appropriate to extend the language with an extra variable type, floats. Float values could allow programmers to perform more complex mathematical calculations when mathematical functions would support floats e.g. trigonometric, square root and logarithms.

Example of Float Syntax

The EBNF for float values

```
digit = [0-9];  
minus = '-';  
dot = '.';  
float = (digit+ | minus digit+) dot digit+
```

7.3 Debug Information

It is often difficult to determine what state an ant is in and whether ants enter a specific trigger block or change roles correctly. In C this information could be made accessible by including `stdio.h` and printing the variable information using `printf()` and then either removing

the debug code before submitting to the tournament or by placing the code inside `#ifdef` statements to make sure the code is only included when compiling with debug information on. Another solution would be to use the `assert()` function that may be used according to the official tournament rules.

Adding a `assert` or printing feature to the AntL language and compiler would require adding support for strings and possibly string concatenations. If added the compiler should support compilation without including the debug code in the output code or by translating it into `assert` statements.

7.4 Execution Time Optimization

Since the execution time of the ant directly impacts the score of the race this is one of the fields in which optimization could have the most optimal effects. One way to optimizing execution time could be to revert from evaluating triggers if the ant is in a role which doesn't use the triggers in question. Another optimization regarding trigger evaluation is to reevaluate only the triggers that depend on brain variables when a change is made to a brain variable. (using the current version of the compiler all triggers are reevaluated each time a brain variable is changed) A more general performance optimization that could be employed by the compiler is to generate output code in which some functions are declared as `inline`. This would in most cases result in a larger binary file, but a slight improvement in execution time.

7.5 Memory Optimization

Another thing that directly impacts the score of the ant race is the size of the brain structure. One strategy for optimizing the brain size could be to use data types with smaller memory footprint where possible. An example of such optimization could be to use a `char` instead of an `integer` to store the current role. This should however only be done if the ant in question does not use more than 256 different roles, the number of unique values that can be stored in a `char`.

A more advanced method of reducing the brain size could be to allow the user of AntL to allocate variables of arbitrary size e.g. 6 or 10 bit, and let the compiler organize these variables and map them into ANSI C variables. One example of optimization could be to store 8 boolean values in one `char`, or store the two 6 and 10 bit variables mentioned above in one short, a 16 bit C variable. This would of course result in an increase of the execution time since calculations are needed to fetch and store values in these odd sized variables.

In order to minimize this overhead of reading and writing to odd sized brain variables one could decode all brain variables and store them in standard C variables before running the behavior and finalize code. At the very end of the execution of the main function these values should be written back into the odd sized brain variables.

7.6 Inherit Roles from Super Roles

Instead of having to write `Role general, rambo`, specify a shared behavior and then specifying a specific behavior it would be possible to support a hierarchy of roles. This hierarchy should allow a role to inherit a behavior block from all roles they are descendants of. Given

a hierarchy of roles as shown in figure 7.1 it would be possible to specify the behavior of the parent role (soldier) and behavior of the individual sub-roles. Figure 7.2 shows how the role hierarchy concept could be used to specify shared behavior without having to list all sub roles.

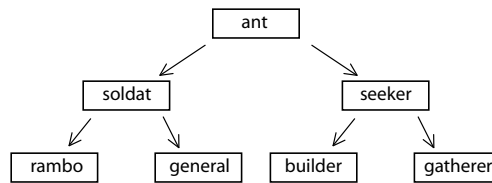


Figure 7.1: Example role hierarchy tree.

Behavior

```

Role soldier
    // Shared behavior

    Role general
        // Communicate orders to other soldiers
    endRole

    Role rambo
        // Search for enemies
    endRole
endRole
endBehavior
  
```

Figure 7.2: Example code using the role hierarchy concept.

7.7 Editor

For a programming language to be comfortable to use by programmers an editor or even a developing environment with some of the following features could be useful:

- Integrated execution of compiler and handling of error messages.
- Syntax highlighting.
- Code completion.
- Collapsing of code blocks, in AntLsource code would make it possible to collapse code blocks based on the start and end token for each of the blocks and control structures.

7.7.1 Syntax Highlighting

Syntax highlighting of the source code gives the programmer a better overview of where blocks start and end and easier separation of elements in complex expressions. The highlighting involves both marking keywords (block start and end tokens), expression variables, values and function calls.

The syntax highlighting could be based on building the AST and walking through it while highlighting each node if necessary. Another less expensive approach - in terms of runtime - would be to use a simple automate (DFA or NFA) instead of building and walking the tree each time the code view is updated and in addition it would work on incomplete trees.

7.7.2 Code Completion

Figure 7.3 shows how code completion functionality can be displayed to the user. Every time the user either stops typing for a specified amount of time or when pressing a key combination a drop-down appears after the cursor location in the editor text field.

The AntL-compiler error handling implementation presented on section 4.4 outputs an error message containing a list of possible tokens that is accepted where an unexpected tokens occurs.

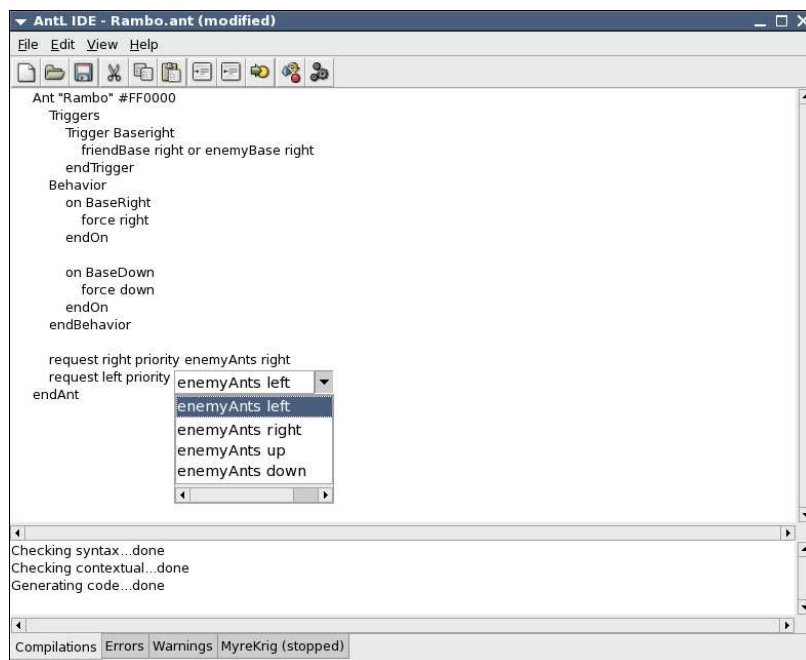


Figure 7.3: Illustration of how an editor could be structured to feature the discussed topics.

SOFTWARE PROCESSES

8.1 Development Strategy

An important part of creating software is to choose a development strategy suited to the type of software and the logistical circumstances in question. The following section describes which development strategies were used in the development of the AntL and the AntL-compiler and some of the reasoning behind this choice.

8.1.1 Spiraling development model

The nature of the spiral model is that there are no fixed borders between specification, design and implementation. Risks are identified and taken care of throughout the development process, and the different phases of the development are constantly reevaluated.

This development model was chosen in order to allow changes in the language and in the compiler as soon as the need for change was realized. The model allowed us to make adjustments to our language specification throughout the development process when inconsistencies or errors were recognized this meant a quicker response to errors. An unfortunate consequence of the spiral model is a bigger risk of total project failure. This is due to the fact that there never is a finished product until the very end of the project. Therefore there is no possibility of releasing the project prematurely and have a working but slightly erroneous result.

8.1.2 Alternatives

An alternative approach could have been to choose an iterative development model in which one iteration is finished before identifying new risks and then addressing these in the following iterations. In this type of development there exists an option of ending prematurely and consider the product of the last iteration the final version. This model was however rejected due to the lack of flexibility in respect to revising the language specification when an error or a possible improvement were realized.

8.1.3 Conclusion on Development Strategies

Throughout the development of the language and the compiler several new ideas surfaced, and the spiral development model allowed us to incorporate these changes immediately into the language as they occurred. The main pitfall of the spiral model was avoided and the strategy proved a success.

8.2 Quality Goals and Evaluation

In order to ensure the quality of software a list of quality goals must be devised before the design process. In the case of our language we chose to set up various goals for the language

AntL itself and for the AntL-compiler.

The quality goals for the language can be summarized to two things; readability and flexibility. Ants written in AntL have to be easy to read when a user has learned the basis of the language. The flexibility goal was to hinder rigid language-construct that would not hamper the development of ants.

The quality goals for the compiler are different from the language even though readability is still paramount but creating correct and secondary efficient output code have been prioritized higher.

8.2.1 Evaluation of Quality Goals

The above mentioned quality goals have been evaluated throughout the development spiral, and no final evaluation has been carried out. Through the evaluations during each development cycle we have however come to the conclusion that the quality goals for the language are meet. The goals for the compiler have however not been meet in full since there has been no attempt to minimize the memory usage and execution times of generated ants. Several strategies for such optimizations are described in chapter 7.

8.3 Testing

The focus for testing has been on behaviorally equality between the output code and the AntL code when testing the compiler. This was done using different strategies including a systematic test of all nontrivial constructions of the AntL and furthermore a small scale user test was created.

8.3.1 Testing of the Output

To test our compilers output code a set of simple test ants were created. Each of these ants tests a specific part of AntL and is made in such a manner that it can be visually determined if the part of AntL is translated into correct output code.

An example of a test ant:

```

Ant "voterequest" # ff66ff
  Behavior
    request right priority 10
    request right priority 10
    vote left weight 10
    vote left weight 10
  endBehavior
endAnt

```

The above example shows an ant that requests right twice with a priority of "10", and votes left twice with a weight of "10" votes. Since votes are cumulated the total number of votes on the action "left" is "20", and the priority of the action "right" is "10". The resulting move should therefore be that the ant moves left. This can easily be determined by

compiling the ant using our compiler, and then compile the MyreKrig program including only the test ant. When running the graphical version of MyreKrig it is easy to determine visually whether the test ant walks left.

Ants that test changes in role, communication, comparisons and many other central parts of the AntL were also created. This testing is an extreme variant of black box testing of the compiler since the output code is not looked at but only the actual behavior of the ant.

Conclusion on Test Ants

The testing proved to be valuable because several errors that otherwise would have gone unnoticed were caught using the test ants. One of the errors that were found by using the test ants were that it were not possible to add or subtract two positions.

8.3.2 User Test of the Compiler

Apart from the black box testing of the compiler a small scale user test of the compiler was created. This test consisted of a set of activities. These were the creation of new ants using the AntL in order to test both the compiler and the actual use of the language.

Apart from making new ants using the AntL, the ant Rambo (one of the ants from the homepage) was recreated using AntL. This was done to see whether the behavior of Rambo could be expressed in AntL and to see whether the behavior of the new Rambo actually matched the original. In order to verify the behavioral equality of the two versions of Rambo ants the code of the original Rambo was compared with the Rambo created by the AntL-compiler.

Description of the Behavior of Rambo

In order to compare the Rambo created in AntL with the original Rambo a formal description of the Rambo ants behavior was needed. Rambo has an empty brain struct which ensures that it's only dictated by the environment. It follows the two rules listed below in the order they appear:

1. Stay on the homebase
2. Kill enemy ants

These rules ensures that Rambo stays on the homebase unless there is an enemy ant outside the homebase. An ant scans the surrounding fields for enemies and if found moves onto the field with the most enemies when standing on the homebase. When an ant is adjacent to the base it makes an unconditional move back to the base.

Rambo ant ensures that there are no enemies outside its base as long as there are at least 4 ants at the beginning of a battle. MyreKrig does however prevent Rambo from being a nearly invincible ant because the order in which the teams get their chance to move is random (see section 2.2.1 for more info).

Analysis of Rambo created using AntL

Rambo expressed in AntL and compiled to C is in appendix C. The analysis focuses on the compiled ants main function which contains the behavior of the ant, this is located at line 268.

The first five lines of code creates the structures used to hold the triggers and the behavior environment. Lines 275 to 286 ensures that an ant moves back to its homebase if its placed next to it and not on it. The first if statement that is triggered uses the `Rambo3force` function to set `force = true` and `forceAction = action` where `action` is the parameter given to the force function. Setting `force = true` ensures that the other if statements won't override the `forceAction`.

Lines 287 to 294 performs a request for each field around the base with a priority equal to the number of enemy ants on the field. An action is given priority 0 if there isn't any enemies on a field. `Rambo3request` only overrides the request it holds if the priority of a given action is higher than the one it holds. `Rambo3request` contains the action here with priority 0 per default which would cause the ant to stay if there weren't any enemy ants on a surrounding field.

`Rambo3Main` calls `Rambo3evaluateAction` at line 295 and returns the value it returns. `Rambo3evaluateAction` will return the action stored in `forceAction` if `force == true`. The function returns `requestAction` if `force != true` (the semantics describing this behavior is in section 3.4). `Rambo` expressed in AntL should be behaviorally equal to the original `Rambo` because it will move onto the base if it is placed next to it. It will move onto the field with the most enemies if it is placed on the homebase and there are enemies outside the base. It stays if there aren't any enemies and it is placed on the homebase.

Running the Two Versions

A game in `MyreKrig` were run with both ants (one at the time) under similar circumstances to tests if the two version were behaviorally equal. `MyreKrig` was run with two instances of the Null ant and with a specific random seed to ensure similar circumstances. The random seed determines among other things where each teams base is placed and where food is placed at the beginning of a battle. Two games in `MyreKrig` are the same given the same random seed.

The result which can be seen in appendix F and G shows that that the only difference between the versions of `Rambo` is how much time they use. The consequence of this is that the ant that uses less time would get a higher ranking ¹. This leads to the conclusion that the two ants are behaviorally equal. It is impossible to conclude anything from the difference in time between the two `Rambo` ants, because the machine that `MyreKrig` was running on were in used during the test runs.

Conclusion of User Testing

The user test did not reveal any errors in the compiler, but it did give us some valuable experience of actually using the language and it did reveal possible improvements to the language itself. These improvement are discussed in the chapter 7.

8.3.3 Conclusion on Testing

The different test of the compiler proved to be valuable in respect to the number and severity of the errors found and corrected as a result of the testing. However the user test was a very small scale testing and could have been even more valuable if executed in a much larger scale.

¹This is however not an issue since `Rambo` doesn't win a single battle, see section 2.2.3 for further explanation

CONCLUSION

Through this report we have documented the development of a language designed to support agent oriented programming within the scope of MyreKrig. As a part of this development we have made a formal description of the syntax and of essential parts of the semantics of the language.

Furthermore we have implemented a compiler for the language and through this worked with different compiler compilers and different strategies for parsing. Working with compiler compilers gave us an extensive knowledge about creating grammars for SableCC and syntax creating in general.

We have achieved the goal of creating an agent based language, as the language fulfills the requirements for a agent based language as specified in our problem analysis. The goal of creating a language that is easier to use has also been achieved to some degree. As we have argued, our language seems to have a steep learning curve due to the unusual control structures and statements. This disadvantage may be canceled out by the increase in productivity as soon as a understanding for the language has been developed.

We used the operational semantics to give us a framework in which we could discuss exactly how we understood certain features of the language. Without the semantics it would have been hard to exemplify how for example the vote/request/force system should be understood.

During the implementation theories of compiler design were put into perspective through practical use. Even though our compiler does not feature more advanced compiler-algorithms these were also considered but found no practical use.

Bibliography

- [1] Alphaworks. Robocode. webpage, January 2004.
<http://www.alphaworks.ibm.com/tech/robocode>.
- [2] Jasmina Arifovic, Svetlana Pevnitskaya, John Ledyard, Thomas R. Palfrey, and Walter Yuan. Turing tournament. webpage, January 2004.
<http://turing.ssel.caltech.edu/node15.html>.
- [3] Aske Simon Christensen. www.myrekrig.dk. webpage, January 2003.
<http://www.myrekrig.dk>.
- [4] CogniToy. Mindrover. webpage, January 2004.
<http://www.mindrover.com/mindrover/mindrover.htm>.
- [5] The RoboCup Federation. Robocup. webpage, January 2004. <http://www.robocup.org/>.
- [6] FIRA. Federation of international robot-soccer association. webpage, January 2004.
<http://www.fira.net/>.
- [7] Marc Gueury, Remi Coulon, and Maido Remm. Robot auto racing simulator. webpage, January 2004. <http://rars.sourceforge.net/>.
- [8] Hans Hüttel. Pilen ved træets rod, Operationel semantik for programmeringssprog. Aalborg Universitet, 2004.
- [9] Indrek Mandre. Alternate sablecc. webpage, January 2004.
<http://www.mare.ee/indrek/sablecc/>.
- [10] NASA. Mars exploration rovers. webpage, January 2004.
<http://www.jpl.nasa.gov/missions/current/marsexplorationrovers.html>.
- [11] Loebner Prize. Loebner prize. webpage, January 2004.
<http://www.loebner.net/Prizef/loebner-prize.html>.
- [12] Stuart Russel and Peter Norvig. Artificial Intelligence - A Modern Approach. Alan Apt, 1995.
- [13] Robert W. Sebesta. Concepts of Programming Languages. Addison Wesley, 2004.
- [14] A. M. Turing. Computing machinery and intelligence. Mind, 1950.
<http://www.loebner.net/Prizef/TuringArticle.html>.
- [15] David A. Watt and Deryck F. Brown. Programming language processors in Java. Prentice Hall, 2000.
- [16] Étienne Gagnon. Sablecc, an object oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, March 1998.
<http://www.sablecc.org/thesis/thesis.php>.

ACCOMPANYING CD

The accompanying CD includes the following material:

- A run-able version of the AntL-compiler.
`compiler/dist/antlang.jar`
- Java Source code for the AntL-compiler.
`compiler/source/`
- Javadoc for the AntL-compiler.
`docs/index.html`
- SableCC grammar file
`syntax/antlang.xss`
- Example ants written in AntL.
`testants/`
- Electronic version of this report in a Postscript and PDF version.
`report/`

The `index.html` file in the root directory of the CD contains a description of the CD content and how the AntL-compiler can be run.

RAMBO.ANT

```
Ant "Rambo3" #FF0000
  Triggers
    Trigger BaseRight
      friendBase right or enemyBase right
    endTrigger
    Trigger BaseDown
      friendBase down or enemyBase down
    endTrigger
    Trigger BaseLeft
      friendBase left or enemyBase left
    endTrigger
    Trigger BaseUp
      friendBase up or enemyBase up
    endTrigger
  endTriggers
  Behavior
    on BaseRight
      force right
    endOn
    on BaseDown
      force down
    endOn
    on BaseLeft
      force left
    endOn
    on BaseUp
      force up
    endOn
    request right priority enemyAnts right
    request down priority enemyAnts down
    request left priority enemyAnts left
    request up priority enemyAnts up
  endBehavior
endAnt
```



```

62  struct Rambo3EnvBehavior {
63      boolean force;
64      int forceAction;
65      int requestPriority;
66      int requestAction;
67      int voteHere;
68      int voteRight;
69      int voteDown;
70      int voteLeft;
71      int voteUp;
72      int voteDragHere;
73      int voteDragRight;
74      int voteDragDown;
75      int voteDragLeft;
76      int voteDragUp;
77      int voteBuild;
78      /* For handling votes , requests and force statements concerning roles */
79      boolean forceRole;
80      int forceRoleName;
81      int requestRolePriority;
82      int requestRoleName;
83      /* For keeping the votes on each role */
84  }
85  ;
86  struct Rambo3Triggers {
87      int BaseRight;
88      int BaseDown;
89      int BaseLeft;
90      int BaseUp;
91  }
92  ;
93  /******
94      *                               UTILITY FUNCTIONS                               *
95      *****/
96  /* Utility functions used to add, subtract and compare positions (position)*/
97  position Rambo3addPos(position a, position b) {
98      return (position){
99          a.x + b.x, a.y + b.y}
100     ;
101     }
102  position Rambo3subPos(position a, position b) {
103     return (position){
104         a.x - b.x, a.y - b.y}
105     ;
106     }
107  boolean Rambo3cmpPos(position a, position b) {
108     return (a.x == b.x) && (a.y == b.y);
109     }
110  /* Utility function used to request actions*/
111  void Rambo3request(action act, int priority, struct Rambo3EnvBehavior *envBehavior) {
112     if (!(envBehavior -> force)) {
113         if (priority > envBehavior -> requestPriority) {
114             envBehavior -> requestPriority = priority;
115             envBehavior -> requestAction = act;
116         }
117     }
118     }
119  /* Utility function used to force actions*/
120  void Rambo3force(action act, struct Rambo3EnvBehavior *envBehavior) {
121     if (!(envBehavior -> force)) {
122         envBehavior -> force = true;
123         envBehavior -> forceAction = act;
124     }
125     }
126  /* Utility function used to vote upon actions*/
127  void Rambo3vote(action act, int vote, struct Rambo3EnvBehavior *envBehavior) {
128     switch (act) {
129         case (here):
130             envBehavior -> voteHere += vote;
131     }

```

```

132     case ( left ):
133         envBehavior -> voteLeft += vote ;
134     break ;
135     case ( down ):
136         envBehavior -> voteDown += vote ;
137     break ;
138     case ( right ):
139         envBehavior -> voteRight += vote ;
140     break ;
141     case ( up ):
142         envBehavior -> voteUp += vote ;
143     break ;
144     case ( drag here ):
145         envBehavior -> voteDragHere += vote ;
146     break ;
147     case ( drag left ):
148         envBehavior -> voteDragLeft += vote ;
149     break ;
150     case ( drag down ):
151         envBehavior -> voteDragDown += vote ;
152     break ;
153     case ( drag right ):
154         envBehavior -> voteDragRight += vote ;
155     break ;
156     case ( drag up ):
157         envBehavior -> voteDragUp += vote ;
158     break ;
159     default :
160         envBehavior -> voteBuild += vote ;
161     break ;
162 }
163 }
164 /* Utility function used to reset the behaviour environment */
165 void Rambo3resetEnvBehavior( struct Rambo3EnvBehavior *envBehavior ) {
166     envBehavior -> force = false ;
167     envBehavior -> forceAction = 0 ;
168     envBehavior -> requestPriority = 0 ;
169     envBehavior -> requestAction = 0 ;
170     envBehavior -> voteHere = 0 ;
171     envBehavior -> voteRight = 0 ;
172     envBehavior -> voteDown = 0 ;
173     envBehavior -> voteLeft = 0 ;
174     envBehavior -> voteUp = 0 ;
175     envBehavior -> voteDragHere = 0 ;
176     envBehavior -> voteDragRight = 0 ;
177     envBehavior -> voteDragDown = 0 ;
178     envBehavior -> voteDragLeft = 0 ;
179     envBehavior -> voteDragUp = 0 ;
180     envBehavior -> voteBuild = 0 ;
181     /* For handling votes , requests and force statements concerning roles */
182     envBehavior -> forceRole = false ;
183     envBehavior -> forceRoleName = 0 ;
184     envBehavior -> requestRolePriority = 0 ;
185     envBehavior -> requestRoleName = 0 ;
186 }
187 int Rambo3evaluateAction( struct Rambo3EnvBehavior *envBehavior ) {
188     int p ;
189     action a ;
190     if ( envBehavior -> force ) {
191         return envBehavior -> forceAction ;
192     }
193     else {
194         p = envBehavior -> requestPriority ;
195         a = envBehavior -> requestAction ;
196         if ( envBehavior -> voteHere > p ) {
197             p = envBehavior -> voteHere ;
198             a = here ;
199         }
200         if ( envBehavior -> voteRight > p ) {
201             p = envBehavior -> voteRight ;

```

```

202         a = right;
203     }
204     if (envBehavior -> voteDown > p) {
205         p = envBehavior -> voteDown;
206         a = down;
207     }
208     if (envBehavior -> voteLeft > p) {
209         p = envBehavior -> voteLeft;
210         a = left;
211     }
212     if (envBehavior -> voteUp > p) {
213         p = envBehavior -> voteUp;
214         a = up;
215     }
216     if (envBehavior -> voteDragHere > p) {
217         p = envBehavior -> voteDragHere;
218         a = (drag here);
219     }
220     if (envBehavior -> voteDragRight > p) {
221         p = envBehavior -> voteRight;
222         a = (drag right);
223     }
224     if (envBehavior -> voteDragDown > p) {
225         p = envBehavior -> voteDragDown;
226         a = (drag down);
227     }
228     if (envBehavior -> voteDragLeft > p) {
229         p = envBehavior -> voteDragLeft;
230         a = (drag left);
231     }
232     if (envBehavior -> voteDragUp > p) {
233         p = envBehavior -> voteDragUp;
234         a = (drag up);
235     }
236     if (envBehavior -> voteBuild > p) {
237         p = envBehavior -> voteBuild;
238         a = build;
239     }
240     return a;
241 }
242 }
243 void Rambo3evaluateTriggers(struct Rambo3Triggers *triggers ,
244     struct SquareData *environment , struct Rambo3Brain *beliefs);
245 /*****
246 *           FUNCTIONS DEFINED BY THE USER IN ANTL
247 *****/
248 /*****
249 *           TRIGGER EVALUATION FUNCTION
250 *****/
251 void Rambo3evaluateTriggers(struct Rambo3Triggers *triggers ,
252     struct SquareData *environment , struct Rambo3Brain *beliefs) {
253     triggers -> BaseRight = (!(environment[ right].Team) && environment[ right].Base)
254     ? true : false) || ((environment[ right].Base && environment[ right].Team)
255     ? true : false);
256     triggers -> BaseDown = (!(environment[ down].Team) && environment[ down].Base)
257     ? true : false) || ((environment[ down].Base && environment[ down].Team)
258     ? true : false);
259     triggers -> BaseLeft = (!(environment[ left].Team) && environment[ left].Base)
260     ? true : false) || ((environment[ left].Base && environment[ left].Team)
261     ? true : false);
262     triggers -> BaseUp = (!(environment[ up].Team) && environment[ up].Base)
263     ? true : false) || ((environment[ up].Base && environment[ up].Team) ? true : false);
264 }
265 /*****
266 *           ANT MAIN FUNCTION
267 *****/
268 int Rambo3Main (struct SquareData *environment , struct Rambo3Brain *beliefs) {
269     struct Rambo3Triggers triggers;
270     struct Rambo3EnvBehavior envBehavior;
271     action move;

```

```

272 Rambo3resetEnvBehavior(&envBehavior);
273 Rambo3evaluateTriggers(&triggers, environment, beliefs);
274 /*START OF DYNAMIC GENERATED BEHAVIOR CODE*/
275 if ( triggers.BaseRight) {
276     Rambo3force ( right, &envBehavior);
277 }
278 if ( triggers.BaseDown) {
279     Rambo3force ( down, &envBehavior);
280 }
281 if ( triggers.BaseLeft) {
282     Rambo3force ( left, &envBehavior);
283 }
284 if ( triggers.BaseUp) {
285     Rambo3force ( up, &envBehavior);
286 }
287 Rambo3request ( right, ((environment[ right].NumAnts && environment[ right].Team) ?
288     environment[ right].NumAnts : 0), &envBehavior);
289 Rambo3request ( down, ((environment[ down].NumAnts && environment[ down].Team) ?
290     environment[ down].NumAnts : 0), &envBehavior);
291 Rambo3request ( left, ((environment[ left].NumAnts && environment[ left].Team) ?
292     environment[ left].NumAnts : 0), &envBehavior);
293 Rambo3request ( up, ((environment[ up].NumAnts && environment[ up].Team) ?
294     environment[ up].NumAnts : 0), &envBehavior);
295 return Rambo3evaluateAction(&envBehavior);
296 }
297 /*****
298 *                               PREPROCESSOR COMMANDS                               *
299 *****/
300 #undef drag
301 #define false 0
302 #define true (!false)
303 DefineAnt(Rambo3, "Rambo3#FF0000", Rambo3Main, struct Rambo3Brain)

```

EXAMPLE.ANT

```

1 ///////////////////////////////////////////////////////////////////
2 // Example ant, used to illustrate how an ant can be structured
3 //
4 // Very basic and not very efficient ant
5 ///////////////////////////////////////////////////////////////////
6
7 Ant "Example" #ff9999 defaultrole Seeker
8   Brain
9     position location
10  endBrain
11
12  Functions
13    Function dist(position p) returns integer
14      return abs(p.x) + abs(p.y)
15    endFunction
16
17    Function rndMove() returns direction
18      integer rnd := random(1, 4)
19
20      if (rnd = 1)
21        return right
22      elseif (rnd = 2)
23        return down
24      elseif (rnd = 3)
25        return left
26      elseif (rnd = 4)
27        return up
28      endIf
29
30      return here
31    endFunction
32
33    Function moveTowards(position to) returns direction
34      position relativeVector := to - location
35      direction xdir
36      direction ydir
37      direction dir
38
39      if relativeVector.x > 0
40        xdir := right
41      else
42        xdir := left

```

```
43         endIf
44
45         if relativeVector.y > 0
46             ydir := up
47         else
48             ydir := down
49         endIf
50
51         if random(1, abs(relativeVector.x)) > random(1, \
52 abs(relativeVector.y))
53             dir := xdir
54         else
55             dir := ydir
56         endIf
57
58         return dir
59     endFunction
60 endFunctions
61
62 Triggers
63     Trigger atBase
64         friendBase here
65     endTrigger
66
67     // food triggers
68     Trigger foundFood
69         numberFood here >= friendAnts here
70     endTrigger
71
72     Trigger foundFoodRight
73         numberFood right > friendAnts right
74     endTrigger
75
76     Trigger foundFoodDown
77         numberFood down > friendAnts down
78     endTrigger
79
80     Trigger foundFoodLeft
81         numberFood left > friendAnts left
82     endTrigger
83
84     Trigger foundFoodUp
85         numberFood up > friendAnts up
86     endTrigger
87
88     // enemy triggers
89     Trigger enemyRight
90         enemyAnts right > 0
91     endTrigger
```

```
92
93     Trigger enemyDown
94         enemyAnts down > 0
95     endTrigger
96
97     Trigger enemyLeft
98         enemyAnts left > 0
99     endTrigger
100
101     Trigger enemyUp
102         enemyAnts up > 0
103     endTrigger
104 endTriggers
105
106 Behavior
107     // communicate base location
108     integer a := 1
109     while a < friendAnts here
110         if dist(location@friends[a]) < dist(location)
111             location := location@friends[a]
112         endIf
113         a := a + 1
114     endWhile
115
116     Role Seeker
117         // Default behavior for seeker is a random move
118         request rndMove() priority 1
119
120         // Triggered behavior
121         on foundFood
122             request drag moveTowards((0,0)) priority 100
123             request changerole Retriever priority 100
124         endOn
125
126         on foundFoodRight
127             request right priority 10
128         endOn
129
130         on foundFoodDown
131             request down priority 10
132         endOn
133
134         on foundFoodLeft
135             request left priority 10
136         endOn
137
138         on foundFoodUp
139             request up priority 10
140         endOn
```

```
141     endRole
142
143     Role Retriever
144         vote drag moveTowards((0,0)) weight 10
145
146         on atBase
147             vote changerole Seeker weight 10
148         endOn
149     endRole
150
151     // enemy handling , the same for all roles
152     on enemyRight
153         force right
154     endOn
155
156     on enemyDown
157         force down
158     endOn
159
160     on enemyLeft
161         force left
162     endOn
163
164     on enemyUp
165         force up
166     endOn
167 endBehavior
168
169     Finalize
170         if ((move = right) or (move = drag right)) and \
171 (friendAnts right != MaxSquareAnts)
172             location := location + (1, 0)
173         elseif ((move = down) or (move = drag down)) and \
174 (friendAnts down != MaxSquareAnts)
175             location := location + (0, -1)
176         elseif ((move = left) or (move = drag left)) and \
177 (friendAnts left != MaxSquareAnts)
178             location := location + (-1, 0)
179         elseif ((move = up) or (move = drag up)) and \
180 (friendAnts up != MaxSquareAnts)
181             location := location + (0, 1)
182         endIf
183     endFinalize
184 endAnt
```

SYNTAX

```
Package dk.aau.cs.antlang;
```

```

/*****
/*                               HELPERS                               */
*****/

```

Helpers

```

newline = (32 | 9)* (10+ | (10 13)+ | 13+);
letter  = ([ 'a'..'z' ] | [ 'A'..'Z' ]);
digit   = [ '0'..'9' ];
minus   = '-';
space   = 32;
tab     = 9;
quote   = 34;
hex     = '#' ([ '0'..'9' ] | [ 'a'..'f' ] | [ 'A'..'F' ])
          ([ '0'..'9' ] | [ 'a'..'f' ] | [ 'A'..'F' ])
          ([ '0'..'9' ] | [ 'a'..'f' ] | [ 'A'..'F' ])
          ([ '0'..'9' ] | [ 'a'..'f' ] | [ 'A'..'F' ])
          ([ '0'..'9' ] | [ 'a'..'f' ] | [ 'A'..'F' ])
          ([ '0'..'9' ] | [ 'a'..'f' ] | [ 'A'..'F' ]);

```

```

/*****
/*                               TOKENS                               */
*****/

```

Tokens

```

// Block tokens
//-----
// Ant block
ant = newline? 'Ant';
endant = 'endAnt' newline;

// Brain block
brain = 'Brain' newline;
endbrain = 'endBrain' newline;

// Behavior block
behavior = 'Behavior' newline;
endbehavior = 'endBehavior' newline;

// Triggers block

```

```
triggers = 'Triggers ' newline;
endtriggers = 'endTriggers ' newline;
trigger = 'Trigger ';
endtrigger = 'endTrigger ' newline;

// Functions block
functions = 'Functions ' newline;
endfunctions = 'endFunctions ' newline;
function = 'Function ';
endfunction = 'endFunction ' newline;
returns = 'returns ';
return = 'return ';

// Finalize block
finalize = 'Finalize ' newline;
endfinalize = 'endFinalize ' newline;

// Control blocks
while = 'while ';
endwhile = 'endWhile ' newline;
if = 'if ';
elseif = 'elseif ';
else = 'else ' newline;
endif = 'endIf ' newline;
on = 'on';
endon = 'endOn ' newline;
role = 'Role ';
endrole = 'endRole ' newline;

// Move action and types
//-----
request = 'request ';
vote = 'vote ';
force = 'force ';
priority = 'priority ';
weight = 'weight ';
changerole = 'changerole ';

// Operators
//-----
and = 'and ';
or = 'or ';
not = 'not ';
xor = 'xor ';
nand = 'nand ';
notequal = '!=';
equal = '=';
greater = '>';
less = '<';
```

```

greaterorequal = '>=';
lessorequal = '<=';
minus = '-';
plus = '+';
times = '*';
divide = '/';
modulus = '%';
assignmentoperator = ':=';

// Misc
//-----
parameterseparator = ',';
dot = '.';
true = 'true';
false = 'false';
defaultrole = 'defaultrole';
dotx = '.x';
doty = '.y';

// General
//-----
variabletype = ('integer'|'position'|'boolean'|'direction'|
                'role'|'action');
direction = ('left'|'right'|'up'|'down'|'here');
gamevariables = ('numberFood'|'enemyAnts'|'enemyBase'|
                 'friendAnts'|'friendBase');

drag = 'drag';
build = 'build';

parenthesisleft = '(';
parenthesisright = ')';
parenthesissquareleft = '[';
parenthesissquareright = ']';

antnumber = 'friends';
at = '@';

name = quote ( letter | digit | space)* quote;
color = hex;
idname = letter ( letter | digit)*;
integer = ( digit+ | minus digit+);
whitespace = ( space | tab )+;
eol = newline;

```

Ignored Tokens

```
whitespace;
```

```

/*****
/*          PRODUCTIONS          */
*****/

```

Productions

```

program =
  antdefinition brainblock? functionsblock? triggersblock?
  behaviorblock finalizeblock? endant
  {→ New program(antdefinition , brainblock ,
  functionsblock , triggersblock , behaviorblock ,
  finalizeblock)};

antdefinition =
  {deffull}   ant name color defaultrole idname eol
  {→ New antdefinition.deffull(name, color , idname)} |

  {defnocolor} ant name defaultrole idname eol
  {→ New antdefinition.defnocolor(name, idname)} |

  {defnorole} ant name color eol
  {→ New antdefinition.defnorole(name, color)} |

  {defname}   ant name eol
  {→ New antdefinition.defname(name)} |

  {defsimple} ant eol {→ New antdefinition.defsimple()};

// Brain
//-----

brainblock =
  brain variabledeclaration* endbrain
  {→ New brainblock([variabledeclaration])};

// Functions
//-----

functionsblock =
  functions functionblock* endfunctions
  {→ New functionsblock([functionblock])};

// Formel function declaration
functionblock =
  {function} function idname parenthesisleft
  functionparameters? parenthesisright
  functionreturnstatement eol statement* functionreturn
  endfunction
  {→ New functionblock.function(idname ,
  functionparameters , functionreturnstatement.vartype ,

```

```

        [statement.aststatement], functionreturn))} |

{procedure} function idname paranthesisleft
    functionparameters? paranthesisright eol
    procedurestatement* endfunction
{->New functionblock.procedure(idname,
    functionparameters,
    [procedurestatement.aststatement])});

functionparameters =
    functionparameter additional_function_parameters*;

additional_function_parameters =
    parameterseparator functionparameter;

functionparameter =
    variabletype idname;

functionreturnstatement {-> [vartype]: variabletype } =
    returns [vartype]: variabletype {-> vartype};

functionreturn =
    return expression eol {->New functionreturn(expression)};

// Triggers
triggersblock =
    triggers triggerblock* endtriggers
    {->New triggersblock([triggerblock])});

triggerblock =
    trigger idname [e1]:eol expression [e2]:eol endtrigger
    {->New triggerblock(idname, expression)};

// Behavior
//-----

behaviorblock =
    behavior behaviorbody* endbehavior
    {->New behaviorblock([behaviorbody.aststatement])});

behaviorbody {-> aststatement} =
    {statement} behaviorstatement
    {->New aststatement.void(behaviorstatement.aststatement)} |

    {roleblock} roleblockbehavior
    {->New aststatement.roleblock(roleblockbehavior.roleblock)} |

    {ontrigger} ontriggerbehavior
    {->New aststatement.ontriggerblock

```

```

        (ontriggerbehavior.ontriggerblock)} |

    {request}      request expression prioritystatement eol
    {-> New aststatement.request(expression , prioritystatement)} |

    {vote}        vote expression weightstatement eol
    {-> New aststatement.vote(expression , weightstatement)} |

    {force}       force expression eol
    {-> New aststatement.force(expression)};

prioritystatement =
    priority expression {-> New prioritystatement(expression)};

weightstatement =
    weight expression {-> New weightstatement(expression)};

ontriggerbehavior {-> ontriggerblock } =
    on idname eol behaviorbody* endon
    {-> New ontriggerblock(idname , [behaviorbody.aststatement])};

roleblockbehavior {-> roleblock } =
    role idname additionalrole* eol behaviorbody* endrole
    {-> New roleblock(idname , [additionalrole],
        [behaviorbody.aststatement])};

// Finalize block
//-----

finalizeblock =
    finalize finalizebody* endfinalize
    {-> New finalizeblock([finalizebody.aststatement])};

finalizebody {-> aststatement } =
    {ontrigger} ontriggerfinalize
    {-> New aststatement.ontriggerblock
        (ontriggerfinalize.ontriggerblock)} |

    {statement} finalizestatement
    {-> New aststatement.void(finalizestatement.aststatement)} |

    {roleblock} roleblockfinalize
    {-> New aststatement.roleblock(roleblockfinalize.roleblock)};

ontriggerfinalize {-> ontriggerblock } =
    on idname eol finalizebody* endon
    {-> New ontriggerblock(idname , [finalizebody.aststatement])};

```

```

roleblockfinalize {-> roleblock } =
    role idname additionalrole* finalizebody* endrole
    {->New roleblock(idname, [additionalrole],
        [finalizebody.aststatement])});

additionalrole =
    parameterseparator idname {->New additionalrole(idname)};

// Statements
//-----

statement {-> aststatement } =
    {control}      controlstatement
    {->New aststatement.control(controlstatement)} |

    {assignment}  assignmentstatement
    {->New aststatement.assignment(assignmentstatement)} |

    {variable}    variabledeclaration
    {->New aststatement.variable(variabledeclaration)} |

    {functioncall} functioncall eol
    {->New aststatement.functioncall(functioncall)} |

    {return}      functionreturn
    {->New aststatement.return(functionreturn)};

controlstatement =
    {if}    ifblock |
    {while} whileblock;

ifblock =
    {if}    if expression eol statement* endif
    {->New ifblock.if(expression, [statement.aststatement])} |

    {elseif} if expression eol statement* elseifblock
    {->New ifblock.elseif(expression,
        [statement.aststatement], elseifblock)} |

    {else}   if expression eol statement* elseblock
    {->New ifblock.else(expression,
        [statement.aststatement], elseblock)};

elseifblock =
    {elseif}    elseif expression eol statement* elseifblock
    {->New elseifblock.elseif(expression,
        [statement.aststatement], elseifblock)} |

```

```

{elseifelse} elseif expression eol statement* elseblock
{-> New elseifblock.elseifelse(expression ,
    [statement.aststatement] , elseblock)} |

{elseifend} elseif expression eol statement* endif
{-> New elseifblock.elseifend(expression ,
    [statement.aststatement])};

elseblock =
    else statement* endif {-> New elseblock(
        [statement.aststatement])};

whileblock =
    while expression eol statement* endwhile
    {-> New whileblock(expression , [statement.aststatement])};

procedurestatement {-> aststatement } =
    {control}      procedurecontrolstatement
    {-> New aststatement.control
        (procedurecontrolstatement.controlstatement)} |

    {assignment}  assignmentstatement
    {-> New aststatement.assignment(assignmentstatement)} |

    {variable}    variabledeclaration
    {-> New aststatement.variable(variabledeclaration)} |

    {functioncall} functioncall eol
    {-> New aststatement.functioncall(functioncall)};

procedurecontrolstatement {-> controlstatement } =
    {if}          procedureifblock
    {-> New controlstatement.if(procedureifblock.ifblock)} |

    {while}       procedurewhileblock
    {-> New controlstatement.while
        (procedurewhileblock.whileblock)};

procedureifblock {-> ifblock } =
    {if}          if expression eol procedurestatement* endif
    {-> New ifblock.if(expression ,
        [procedurestatement.aststatement])} |

    {elseif}      if expression eol procedurestatement*
    procedureelseifblock
    {-> New ifblock.elseif(expression ,
        [procedurestatement.aststatement] ,
        procedureelseifblock.elseifblock)} |

```

```

    {else}    if expression eol procedurestatement*
    procedureelseblock
    {->New ifblock.else(expression ,
        [procedurestatement.aststatement],
        procedureelseblock.elseblock)};

procedureelseifblock {-> elseifblock } =
    {elseif}    elseif expression eol procedurestatement*
    procedureelseifblock
    {->New elseifblock.elseif(expression ,
        [procedurestatement.aststatement],
        procedureelseifblock.elseifblock)} |

    {elseifelse} elseif expression eol procedurestatement*
    procedureelseblock
    {->New elseifblock.elseifelse(expression ,
        [procedurestatement.aststatement],
        procedureelseblock.elseblock)} |

    {elseifend} elseif expression eol procedurestatement* endif
    {->New elseifblock.elseifend(expression ,
        [procedurestatement.aststatement])};

procedureelseblock {-> elseblock } =
    else procedurestatement* endif
    {->New elseblock([procedurestatement.aststatement])};

procedurewhileblock {-> whileblock } =
    while expression eol procedurestatement* endwhile
    {->New whileblock(expression ,
        [procedurestatement.aststatement])};

behaviorstatement {-> aststatement } =
    {control}    behaviorcontrolstatement
    {->New aststatement.control
        (behaviorcontrolstatement.controlstatement)} |

    {assignment}    assignmentstatement
    {->New aststatement.assignment(assignmentstatement)} |

    {variable}    variabledeclaration
    {->New aststatement.variable(variabledeclaration)} |

    {functioncall} functioncall eol
    {->New aststatement.functioncall(functioncall)};

behaviorcontrolstatement {-> controlstatement } =
    {if}    behaviorifblock
    {->New controlstatement.if(behaviorifblock.ifblock)} |

```

```

{while} behaviorwhileblock
{-> New controlstatement . while
    (behaviorwhileblock . whileblock) } ;

behaviorifblock {-> ifblock } =
{if}    if expression eol behaviorbody* endif
{-> New ifblock . if(expression , [behaviorbody . aststatement])} |

{elseif} if expression eol behaviorbody* behaviorelseifblock
{-> New ifblock . elseif(expression ,
    [behaviorbody . aststatement],
    behaviorelseifblock . elseifblock) } |

{else}   if expression eol behaviorbody* behaviorelseblock
{-> New ifblock . else(expression , [behaviorbody . aststatement],
    behaviorelseblock . elseblock) };

behaviorelseifblock {-> elseifblock } =
{elseif}    elseif expression eol behaviorbody*
behaviorelseifblock
{-> New elseifblock . elseif(expression ,
    [behaviorbody . aststatement],
    behaviorelseifblock . elseifblock) } |

{elseifelse} elseif expression eol behaviorbody*
behaviorelseblock
{-> New elseifblock . elseifelse(expression ,
    [behaviorbody . aststatement],
    behaviorelseblock . elseblock) } |

{elseifend} elseif expression eol behaviorbody* endif
{-> New elseifblock . elseifend(expression ,
    [behaviorbody . aststatement])};

behaviorelseblock {-> elseblock } =
else behaviorbody* endif
{-> New elseblock ([behaviorbody . aststatement])};

behaviorwhileblock {-> whileblock } =
while expression eol behaviorbody* endwhile
{-> New whileblock(expression , [behaviorbody . aststatement])};

finalizestatement {-> aststatement } =
{control}    finalizecontrolstatement
{-> New aststatement . control
    (finalizecontrolstatement . controlstatement) } |

{assignment} assignmentstatement

```

```

{-> New aststatement.assignment(assignmentstatement)} |

{variable}      variabledeclaration
{-> New aststatement.variable(variabledeclaration)} |

{functioncall} functioncall eol
{-> New aststatement.functioncall(functioncall)};

finalizecontrolstatement {-> controlstatement } =
  {if}      finalizeifblock
  {-> New controlstatement.if(finalizeifblock.ifblock)} |

  {while} finalizewhileblock
  {-> New controlstatement.while
      (finalizewhileblock.whileblock)};

finalizeifblock {-> ifblock } =
  {if}      if expression eol finalizebody* endif
  {-> New ifblock.if(expression , [finalizebody.aststatement])} |

  {elseif} if expression eol finalizebody* finalizeelseifblock
  {-> New ifblock.elseif(expression ,
      [finalizebody.aststatement],
      finalizeelseifblock.elseifblock)} |

  {else}    if expression eol finalizebody* finalizeelseblock
  {-> New ifblock.else(expression , [finalizebody.aststatement],
      finalizeelseblock.elseblock)};

finalizeelseifblock {-> elseifblock } =
  {elseif}    elseif expression eol finalizebody*
  finalizeelseifblock
  {-> New elseifblock.elseif(expression ,
      [finalizebody.aststatement],
      finalizeelseifblock.elseifblock)} |

  {elseifelse} elseif expression eol finalizebody*
  finalizeelseblock
  {-> New elseifblock.elseifelse(expression ,
      [finalizebody.aststatement],
      finalizeelseblock.elseblock)} |

  {elseifend} elseif expression eol finalizebody* endif
  {-> New elseifblock.elseifend(expression ,
      [finalizebody.aststatement])};

finalizeelseblock {-> elseblock } =
  else finalizebody* endif
  {-> New elseblock([finalizebody.aststatement])};

```

```

finalizewhileblock { -> whileblock } =
    while expression eol finalizebody* endwhile
    { -> New whileblock(expression , [ finalizebody . aststatement ])};

assignmentstatement =
    { declaration } variabletype idname assignmenttop expression eol
    { -> New assignmentstatement.declaration(variabletype , idname ,
        assignmenttop , expression ) } |

    { assignment } variablename assignmenttop expression eol
    { -> New assignmentstatement.assignment(variablename ,
        assignmenttop , expression )};

assignmenttop =
    assignmentoperator ;

functioncall =
    idname paranthesisleft actual_parameter_list?
    paranthesisright
    { -> New functioncall(idname , actual_parameter_list)};

actual_parameter_list =
    expression additionalparameters* ;

additionalparameters =
    parameterseparator expression ;

variabledeclaration =
    variabletype idname eol
    { -> New variabledeclaration(variabletype , idname)};

variablename = { simple }          simple_variable |
                { communication } communication_variable ;

simple_variable =
    { normal }      idname |
    { positionx }  idname dotx |
    { positiony }  idname doty ;

communication_variable =
    simple_variable at antnumber paranthesisleft
    expression paranthesisright
    { -> New communication_variable(simple_variable , expression)};

// Expressions
//-----

expression =

```

```

    {single} expression_sec |
    {multi} expression operator expression_sec;

expression_sec =
    unary_operator ? expression_prim;

expression_prim =
    {value}          boolean_value |
    {integer}        integer |
    {position}       position_exp |
    {gamevariables} gamevariables expression_prim |
    {action}         action |
    {variable}       variablename |
    {functioncall}  functioncall |
    {expression}    paranthesisleft expression paranthesisright;

action =
    {direction}     direction |
    {build}         build |
    {drag}          drag expression_prim |
    {changerole}   changerole expression_prim;

position_exp =
    paranthesisleft [e1]:expression parameterseparator
    [e2]:expression paranthesisright
    {->New position_exp(paranthesisleft , e1 , parameterseparator ,
        e2 , paranthesisright)};

operator =
    {bool}          boolean_operator |
    {arithmetic}   arithmetic_operator |
    {arithmetic_comp} arithmetic_comp;

boolean_operator =
    {and}   and |
    {or}    or |
    {xor}   xor |
    {nand}  nand;

arithmetic_operator =
    {plus}    plus |
    {minus}   minus |
    {times}   times |
    {divide}  divide |
    {modulus} modulus;

arithmetic_comp =
    {greater}      greater |
    {greaterorequal} greaterorequal |

```

```

{less }          less |
{lessorequal }  lessorequal |
{equal }        equal |
{notequal }     notequal ;

```

```

unary_operator =
  {boolean } not |
  {arithmetic } minus ;

```

```

boolean_value =
  {true } true |
  {false } false ;

```

```

/*****
/*          ABSTRACT SYNTAX TREE          */
*****/

```

Abstract Syntax Tree

```

program =
  antdefinition brainblock ? functionsblock ? triggersblock ?
  behaviorblock finalizeblock ? ;

```

```

antdefinition =
  {deffull } name color idname |
  {defnocolor } name idname |
  {defnrole } name color |
  {defname } name |
  {defsimple } ;

```

```

brainblock =
  variabledeclaration * ;

```

```

functionsblock =
  functionblock * ;

```

// Formel function declaration

```

functionblock =
  {function } idname functionparameters ? variabletype
  aststatement * functionreturn |

  {procedure } idname functionparameters ? aststatement * ;

```

```

functionparameters =
  functionparameter additional_function_parameters * ;

```

```

additional_function_parameters =
  parameterseparator functionparameter ;

```

```

functionparameter =

```

```

    variabletype idname;

functionreturnstatement =
    variabletype;

functionreturn =
    expression;

// Triggers
triggersblock =
    triggerblock*;

triggerblock =
    idname expression;

// Behavior
//-----

behaviorblock =
    aststatement*;

prioritystatement =
    expression;

weightstatement =
    expression;

ontriggerblock =
    idname aststatement*;

// Finalize block
//-----

finalizeblock =
    aststatement*;

// Statements
//-----

aststatement =
    {control}          controlstatement |
    {assignment}      assignmentstatement |
    {variable}        variabledeclaration |
    {functioncall}    functioncall |
    {void}            aststatement |
    {roleblock}       roleblock |
    {ontriggerblock} ontriggerblock |
    {request}         expression prioritystatement |
    {vote}            expression weightstatement |

```

```

    {force }           expression |
    {return }         functionreturn ;

controlstatement =
    {if }      ifblock |
    {while }   whileblock ;

ifblock =
    {if }      expression aststatement * |
    {elseif } expression aststatement * elseifblock |
    {else }    expression aststatement * elseblock ;

elseifblock =
    {elseif }    expression aststatement * elseifblock |
    {elseifelse } expression aststatement * elseblock |
    {elseifend } expression aststatement * ;

elseblock =
    aststatement * ;

whileblock =
    expression aststatement * ;

assignmentstatement =
    {declaration } variabletype idname assignmentop expression |
    {assignment }  variablename assignmentop expression ;

assignmentop =
    assignmentoperator ;

functioncall =
    idname actual_parameter_list ? ;

actual_parameter_list =
    expression additionalparameters * ;

additionalparameters =
    parameterseparator expression ;

variabledeclaration =
    variabletype idname ;

variablename = { simple }      simple_variable |
               { communication } communication_variable ;

simple_variable =
    {normal }    idname |
    {positionx } idname dotx |
    {positiony } idname doty ;

```

```

communication_variable =
    simple_variable expression;

roleblock =
    idname additionalrole* aststatement*;

additionalrole =
    idname;

// Expressions
//-----

expression =
    {single} expression_sec |
    {multi} expression operator expression_sec;

expression_sec = unary_operator? expression_prim;

expression_prim =
    {value}          boolean_value |
    {integer}        integer |
    {position}       position_exp |
    {gamevariables} gamevariables expression_prim |
    {action}         action |
    {variable}       variablename |
    {functioncall}  functioncall |
    {expression}    paranthesisleft expression paranthesisright;

action =
    {direction}     direction |
    {build}         build |
    {drag}          drag expression_prim |
    {changerole}   changerole expression_prim;

position_exp =
    paranthesisleft [e1]:expression parameterseparator
    [e2]:expression paranthesisright;

operator =
    {bool}          boolean_operator |
    {arithmetic}   arithmetic_operator |
    {arithmetic_comp} arithmetic_comp;

boolean_operator =
    {and}   and |
    {or}    or |
    {xor}   xor |
    {nand}  nand;

```

```
arithmetic_operator =
    {plus }      plus |
    {minus }     minus |
    {times }     times |
    {divide }    divide |
    {modulus }  modulus;

arithmetic_comp =
    {greater }      greater |
    {greaterorequal } greaterorequal |
    {less }         less |
    {lessorequal } lessorequal |
    {equal }        equal |
    {notequal }     notequal;

unary_operator =
    {boolean }     not |
    {arithmetic } minus;

boolean_value =
    {true }  true |
    {false } false;
```

NULL VS RAMBO

MyreKrig version 2.4.7 (26.06.03)

NumTeams : 3
 NewBaseAnts : 25
 NewBaseFood : 50

	MinVal	MaxVal
MapWidth :	173	347
MapHeight :	173	347
StartAnts :	15	40
NewFoodSpace :	10	50
NewFoodMin :	10	30
NewFoodDiff :	5	20
HalfTimeTurn :	10000	10000
TimeOutTurn :	20000	20000
WinPercent :	75	75
HalfTimePercent :	60	60
BattleSize :	3	3
NumBattles :	42	
RandomSeed :	1085575458	

Team letters and ant memory sizes :

A Null	0
B Null	0
C Rambo	0

Batt	Randomseed	Widt	Heig	Ant	Spc	Min	Dif	Teams	Turns	T	Winner
1	1085575458	320	320	19	37	20	10	CBA	2553	W B	Null
2	1813926403	320	320	23	49	23	5	CAB	20000	T A	Null
3	1647989112	320	320	32	30	30	10	BCA	2087	W A	Null
4	4289591057	192	256	38	47	15	12	BCA	20000	T B	Null
5	2452165022	320	320	35	40	21	20	BAC	20000	T A	Null
6	1973692463	256	256	21	14	16	18	BCA	3197	W A	Null
7	2891587348	384	320	30	43	14	17	BCA	3878	W B	Null
8	3865297117	384	256	38	40	12	10	BCA	20000	T B	Null
9	1630468954	192	320	16	33	12	6	CBA	2661	W A	Null
10	1730142363	384	320	27	14	14	19	ACB	2180	W B	Null
11	278051824	256	384	21	50	14	7	ABC	20000	T B	Null
12	987286249	256	256	18	15	23	19	BCA	2533	W A	Null
13	3994577494	256	256	17	39	17	19	CAB	20000	T B	Null
14	1002030919	320	256	32	50	28	15	ABC	2059	W A	Null
15	1913545740	384	320	24	33	29	6	BCA	2588	W A	Null

16	3314549045	320	320	21	45	20	12	ABC	2905	W	A	Null
17	1721678482	320	256	40	31	22	6	BAC	20000	T	B	Null
18	3595724851	384	320	40	39	22	17	ACB	6589	W	B	Null
19	22319976	320	320	35	39	29	19	ABC	4768	W	B	Null
20	334663617	320	256	39	30	11	11	BCA	10000	H	B	Null
21	1084131342	384	320	16	49	24	5	CAB	10000	H	B	Null
22	4277047135	320	256	19	42	16	18	CBA	10000	H	A	Null
23	2183529988	256	256	27	26	14	12	ABC	20000	T	B	Null
24	1599857293	320	192	22	41	20	9	BCA	20000	T	A	Null
25	3390287562	256	320	25	32	29	16	BAC	6832	W	B	Null
26	4267919563	256	384	40	26	10	17	CAB	2775	W	B	Null
27	3432560608	256	384	21	32	20	12	ACB	3236	W	A	Null
28	2422580633	192	384	30	38	27	15	BAC	20000	T	A	Null
29	1247103686	320	256	15	40	28	16	CAB	20000	T	B	Null
30	1682038903	320	384	39	28	19	13	ACB	8448	W	B	Null
31	91481340	256	256	38	28	10	19	CAB	10000	H	A	Null
32	2322842853	320	384	16	19	20	11	BCA	4582	W	B	Null
33	1351727106	320	320	19	20	11	11	ACB	2109	W	B	Null
34	3271811683	384	320	25	47	23	7	BCA	10000	H	A	Null
35	1238108504	256	256	23	25	23	17	BAC	10000	H	B	Null
36	729836657	256	320	16	11	27	11	ABC	10000	H	B	Null
37	193630846	256	256	30	31	26	13	BCA	2154	W	B	Null
38	1853299343	256	256	27	12	26	19	CAB	1272	W	B	Null
39	3386826996	320	320	38	11	28	10	CBA	2779	W	A	Null
40	2888304701	192	320	33	47	28	15	CBA	20000	T	A	Null
41	3331372602	320	256	18	27	20	15	ACB	2792	W	A	Null
42	2288620795	320	256	34	47	16	6	ABC	20000	T	B	Null

Team	Battles	Won	Bases	Ants	Size	Ages	Comb	Time
Null	42	18	1.0	456	169	1558	53.1%	686
Null	42	24	1.0	565	192	1294	45.8%	695
Rambo	42	0	1.0	27	28	3860	88.5%	652
Tot ./ Aver .	126	42	1.0	349	130	1424	50.0%	688

Team	Vict	Perf	Pres
Null	42.9%	42.9%	42.9%
Null	57.1%	56.4%	57.1%
Rambo	0.0%	0.0%	0.0%
Tot ./ Aver .	33.3%	33.1%	33.3%

NULL VS RAMBO3

MyreKrig version 2.4.7 (26.06.03)

NumTeams : 3
 NewBaseAnts : 25
 NewBaseFood : 50

	MinVal	MaxVal
MapWidth :	173	347
MapHeight :	173	347
StartAnts :	15	40
NewFoodSpace :	10	50
NewFoodMin :	10	30
NewFoodDiff :	5	20
HalfTimeTurn :	10000	10000
TimeOutTurn :	20000	20000
WinPercent :	75	75
HalfTimePercent :	60	60
BattleSize :	3	3
NumBattles :	42	
RandomSeed :	1085575458	

Team letters and ant memory sizes :

A Null	0
B Null	0
C Rambo3	0

Batt	Randomseed	Widt	Heig	Ant	Spc	Min	Dif	Teams	Turns	T	Winner
1	1085575458	320	320	19	37	20	10	CBA	2553	W B	Null
2	1813926403	320	320	23	49	23	5	CAB	20000	T A	Null
3	1647989112	320	320	32	30	30	10	BCA	2087	W A	Null
4	4289591057	192	256	38	47	15	12	BCA	20000	T B	Null
5	2452165022	320	320	35	40	21	20	BAC	20000	T A	Null
6	1973692463	256	256	21	14	16	18	BCA	3197	W A	Null
7	2891587348	384	320	30	43	14	17	BCA	3878	W B	Null
8	3865297117	384	256	38	40	12	10	BCA	20000	T B	Null
9	1630468954	192	320	16	33	12	6	CBA	2661	W A	Null
10	1730142363	384	320	27	14	14	19	ACB	2180	W B	Null
11	278051824	256	384	21	50	14	7	ABC	20000	T B	Null
12	987286249	256	256	18	15	23	19	BCA	2533	W A	Null
13	3994577494	256	256	17	39	17	19	CAB	20000	T B	Null
14	1002030919	320	256	32	50	28	15	ABC	2059	W A	Null
15	1913545740	384	320	24	33	29	6	BCA	2588	W A	Null

16	3314549045	320	320	21	45	20	12	ABC	2905	W	A	Null
17	1721678482	320	256	40	31	22	6	BAC	20000	T	B	Null
18	3595724851	384	320	40	39	22	17	ACB	6589	W	B	Null
19	22319976	320	320	35	39	29	19	ABC	4768	W	B	Null
20	334663617	320	256	39	30	11	11	BCA	10000	H	B	Null
21	1084131342	384	320	16	49	24	5	CAB	10000	H	B	Null
22	4277047135	320	256	19	42	16	18	CBA	10000	H	A	Null
23	2183529988	256	256	27	26	14	12	ABC	20000	T	B	Null
24	1599857293	320	192	22	41	20	9	BCA	20000	T	A	Null
25	3390287562	256	320	25	32	29	16	BAC	6832	W	B	Null
26	4267919563	256	384	40	26	10	17	CAB	2775	W	B	Null
27	3432560608	256	384	21	32	20	12	ACB	3236	W	A	Null
28	2422580633	192	384	30	38	27	15	BAC	20000	T	A	Null
29	1247103686	320	256	15	40	28	16	CAB	20000	T	B	Null
30	1682038903	320	384	39	28	19	13	ACB	8448	W	B	Null
31	91481340	256	256	38	28	10	19	CAB	10000	H	A	Null
32	2322842853	320	384	16	19	20	11	BCA	4582	W	B	Null
33	1351727106	320	320	19	20	11	11	ACB	2109	W	B	Null
34	3271811683	384	320	25	47	23	7	BCA	10000	H	A	Null
35	1238108504	256	256	23	25	23	17	BAC	10000	H	B	Null
36	729836657	256	320	16	11	27	11	ABC	10000	H	B	Null
37	193630846	256	256	30	31	26	13	BCA	2154	W	B	Null
38	1853299343	256	256	27	12	26	19	CAB	1272	W	B	Null
39	3386826996	320	320	38	11	28	10	CBA	2779	W	A	Null
40	2888304701	192	320	33	47	28	15	CBA	20000	T	A	Null
41	3331372602	320	256	18	27	20	15	ACB	2792	W	A	Null
42	2288620795	320	256	34	47	16	6	ABC	20000	T	B	Null

Team	Battles	Won	Bases	Ants	Size	Ages	Comb	Time
Null	42	18	1.0	456	169	1558	53.1%	750
Null	42	24	1.0	565	192	1294	45.8%	730
Rambo3	42	0	1.0	27	28	3860	88.5%	732
Tot ./ Aver .	126	42	1.0	349	130	1424	50.0%	739

Team	Vict	Perf	Pres
Null	42.9%	41.9%	42.9%
Null	57.1%	57.3%	57.1%
Rambo3	0.0%	0.0%	0.0%
Tot ./ Aver .	33.3%	33.0%	33.3%