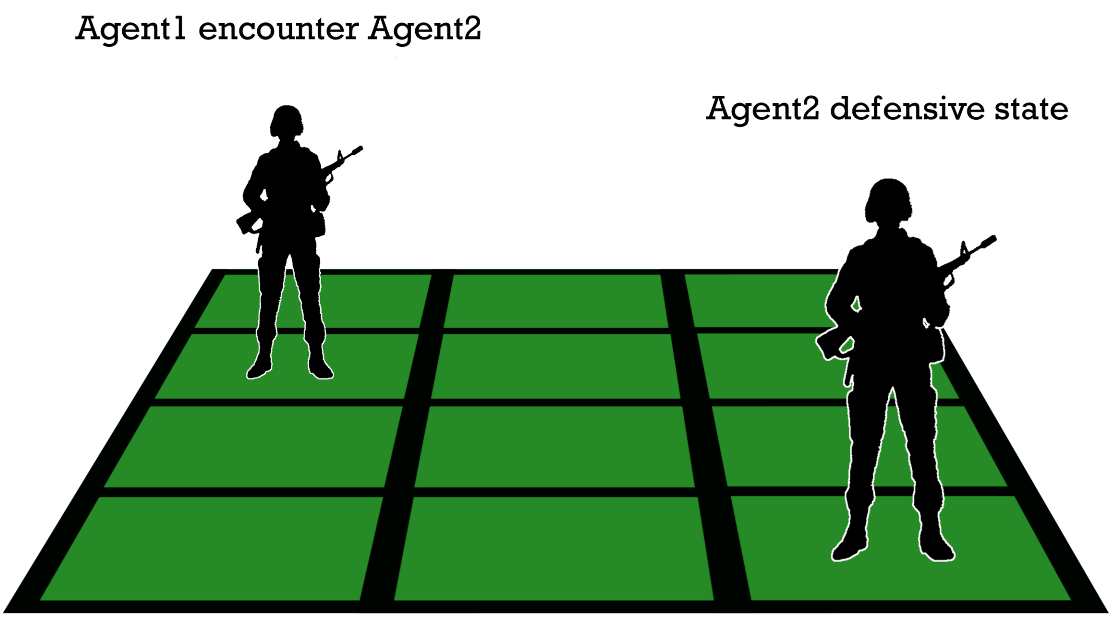
Language and Compiler Development

- A Multi Agent System Wargame



Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
Telephone +45 9940 9940
Telefax +45 9940 9798
http://cs.aau.dk

Title: Wargame
Subject: Language engineering
Semester: Spring Semester 2011
Project group: sw402a

Participants:
Henrik Klarup
Kasper Møller Andersen
Kristian Kolding Foged-Ladefoged
Lasse Rørbæk
Rasmus Aaen
Simon Frandsen

Supervisor:
Jorge Pablo Cordero Hernandez
Paolo Viappiani*

Number of copies: 9

Number of pages: MISSING

Number of appendices: 3

Completed: 27. May 2011

Synopsis:

In this project, an agent oriented language is designed and implemented. The implementation is done via a high-level to high-level compiler. The language is specialized towards a concept we call "multi agent wargame". This wargame gives the user the possibility to simulate programmed battle scenarios.

The language is designed using BNF and EBNF grammar, and implemented via abstract syntax trees and tree traversal. The implementation is described through a big step semantic.

Furthermore we discuss the different aspects of the language and ways to improve it and then compare it to an object oriented language to determine the up- and downsides of this kind of specialized language.

We arrive at the conclusion that the language does exactly what it is supposed to do; provide programmers with a simple language to express a battle scenario.

**Substitute supervisor for Jorge from 1. may.*

Preface

This report is written in the fourth semester of the software engineering study at Aalborg University, spring 2011.

The goal of this project is to acquire knowledge about fundamental principles of programming languages and techniques to describe and translate programming languages in general. Another goal is to get a basic knowledge of central computer science and software technical subjects with a focus on language processing theories and techniques.

We will achieve these goals by designing and implementing a language optimized for controlling a multi agent system in the form of a wargame, which we call *MASSIVE* - **M**ulti **A**gent **S**imulation **S**ystem **I**n **V**irtual **E**nvironment. The product have been written entirely in C# using Visual Studio, this have been done because Visual Studio have a great framework to help develop interfaces. Since its easy to create interfaces with the Visual Studio framework, there have been more time to create the more important part of the project.

Source code examples in the report is represented as follows:

```
if (spelling.ToLower().Equals(spellings[i]))
  {
    this.kind = i;
    break;
  }
```

Source code 1: This is a sorce code example

We expect the reader to have basic knowledge about object oriented programming and the C# language.

Contents

Part I

Introduction

In this part we introduce the project, we cover the subjects multi agent systems, agent oriented languages and existing multi agent environments. Furthermore we specify the rules and usage of the wargame we develop.

Project Introduction

There exist many different programming languages for different purposes, and in this report we have focus on multi agent wargame. In this project we are developing a language and compiler to generate code for a multi agent wargame. This leads to our problem statement:

How can we develop a programming language and compiler, optimized to control agents of a multi agent wargame?

To answer these questions we first need some background knowledge about multi agent systems, agent oriented languages, and the main idea with compilers and interpreters, which will be described in the first part of the report, together with a description of the multi agent system that we are developing.

In *Design*, we describe the basics of languages and compilers.

In *Implementation*, we explain how we have have done the implementation of the language, compiler and the multi agent system environment.

In the *Discussion* we discuss some of our language development choices, and we conclude on the project as a whole.

In the *Epilogue* we discuss what could be improved in future work, and the last part *Appendix* contains other relevant material, such as our full language grammar.

CHAPTER 1

Multi Agent System

The purpose of a Multi Agent System (MAS) is to simulate scenarios in which a number of self-interested agents make decisions that help them, or the an group of agents, to achieve a predefined goal or condition.

In order to achieve this, a number of mechanisms are needed. First of all agents have to be able to make decisions. In order to make smart decisions, agents, like people, need some kind of goal. These goals can be defined in a lot of different ways, one of which is to associate states with values, and make agents strive to be in at the highest value.[**?**]

Another way to implement goals is to introducing a rate of utilization of the robot, again, higher utilization is better. The utilization reward given to a robot performing a task could then be calculated based on expenses associated with the job, and opportunity cost of not being able to perform other actions while performing the current. Agents are typically selfish in this setup, meaning that they will only do things that benefit their own utilization, regardless of the utilization of other agents. This does not mean that they are not able to help each other, it means that they will only do so if it benefits all the agents performing the given task.[**?**], [**?**], [**?**]

1.1 Agent Oriented Languages

Creating a MAS using traditional programming language can be rather difficult and tiresome, you will need to make a agents and their envorioment, therefore it requires some programming skills and time witch can be a problem. In order to overcome this problem, languages specifically designed to create MASes and MAS-enviroments, are being developed, these languages are called Agent Oriented Languages (AOL).

Using an Agent Oriented Language one do not have to make their own environment or functions. One can use the Agent Oriented Language environment and call the functions one needs from the language. By doing so, one do not need the full knowlegde of an OOP language. It is easier and faster to use an Agent Oriented Language to create advanced agent simulations, since all necessary functions are already programmed together with an environment.
Agent Oriented Languages is often more simple to use than OOP langauges, therefore more people have the chance to create agent simulations. The next chapter will look into some existing MAS environments, 2.

CHAPTER 2

Existing Environments

To get an idea of how others have designed a multi agent systems, we will take a look on NetLogo and RoboCode.

2.1 NetLogo

NetLogo is a widespread environment for programming a MAS. NetLogo developed by Uri Wilensky in 1999, at the Northwestern University [?].

NetLogo features a very easy programming language for both creating agents and defining environments, NetLogo also provides a way of manipulating the cosmetics of the MAS simulation. NetLogo has the advantages that even though the programming language is simple, it is also rich on features, and can create MASes that can simulate almost any possible scenario, right from advanced traffic scenarios to how many tadpoles will survive the first week of their lives. [?]

The code shown in the following code-snippet, will generate a simple test with color mixing, to simulate passing of genes.

```
to setup
  clear-all
  ask patches
    [ set pcolor (random colors) * 10 + 5
        if pcolor = 75  ;; 75 is too close to another color so
            change it to 125
```

```
              [ set pcolor 125 ] ]
  reset-ticks
end

to go
  ask patches [ set pcolor [pcolor] of one-of patches ]
  tick
end


; Copyright Uri Wilensky. All rights reserved.
```

NetLogo Source code 2.1: This is a NetLogo source code example.

This example will, together with the NetLogo GUI, create the simulation shown in 2.1. The simulation data is saved in NetLogos custom file format, so that they can be run by someone else.

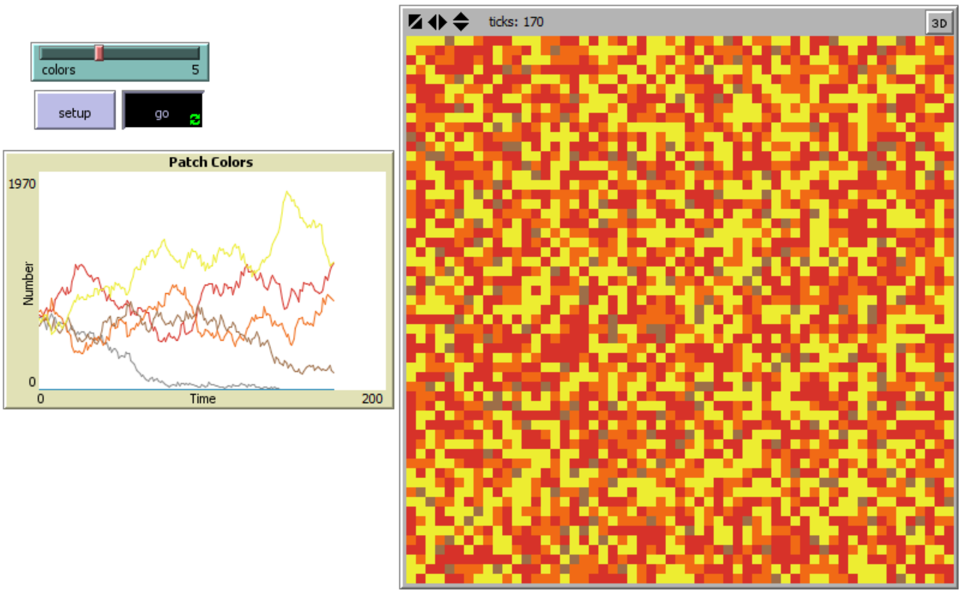


Figure 2.1: Simple Netlogo Simulation

2.2 RoboCode

Another simulation environment is `RoboCode`. This environment is designed to let agents (tanks) fight each other, either in teams or individually. RoboCode allow both pre-programmed agent behaviour, or agents that are controlled by the user in real-time. The battlefield used by RoboCode have a user defined size and is illustrated in figure 2.2. The user of RoboCode can set up battles using pre-defined agents, or new ones can be programmed. The language is build as a Java package. Below is an example of Java code used to produce an agent, usable in the RoboCode battles.

```java
package lrn;
import robocode.*;

public class MyFirstRobot extends Robot {
```

Source code 2.2: The First part of a RoboCode robot. Note the robocode package is imported

```java

  public void run()
  {
    // Initialization of the robot should be put here

    // Robot main loop
    while(true)
    {
      ahead(500);
      turnLeft(50);
      turnGunRight(360);
    }
  }

\begin{java}{onScannedRobot is the method when an agent spots
    another robot}{}

  public void onScannedRobot(ScannedRobotEvent e) {

    fire(1);
  }
```

Source code 2.3: Initialization of the robot and the main loop is in the `run()` function

```java

  public void onHitByBullet(HitByBulletEvent e) {
    // Replace the next line with any behavior you would like
    back(10);
  }
```

Source code 2.4: This method is run everytime the agent is hit

The below code uses a method called `getBearing()`. This returns the heading of the object `e`, which in this case is the agent itself.

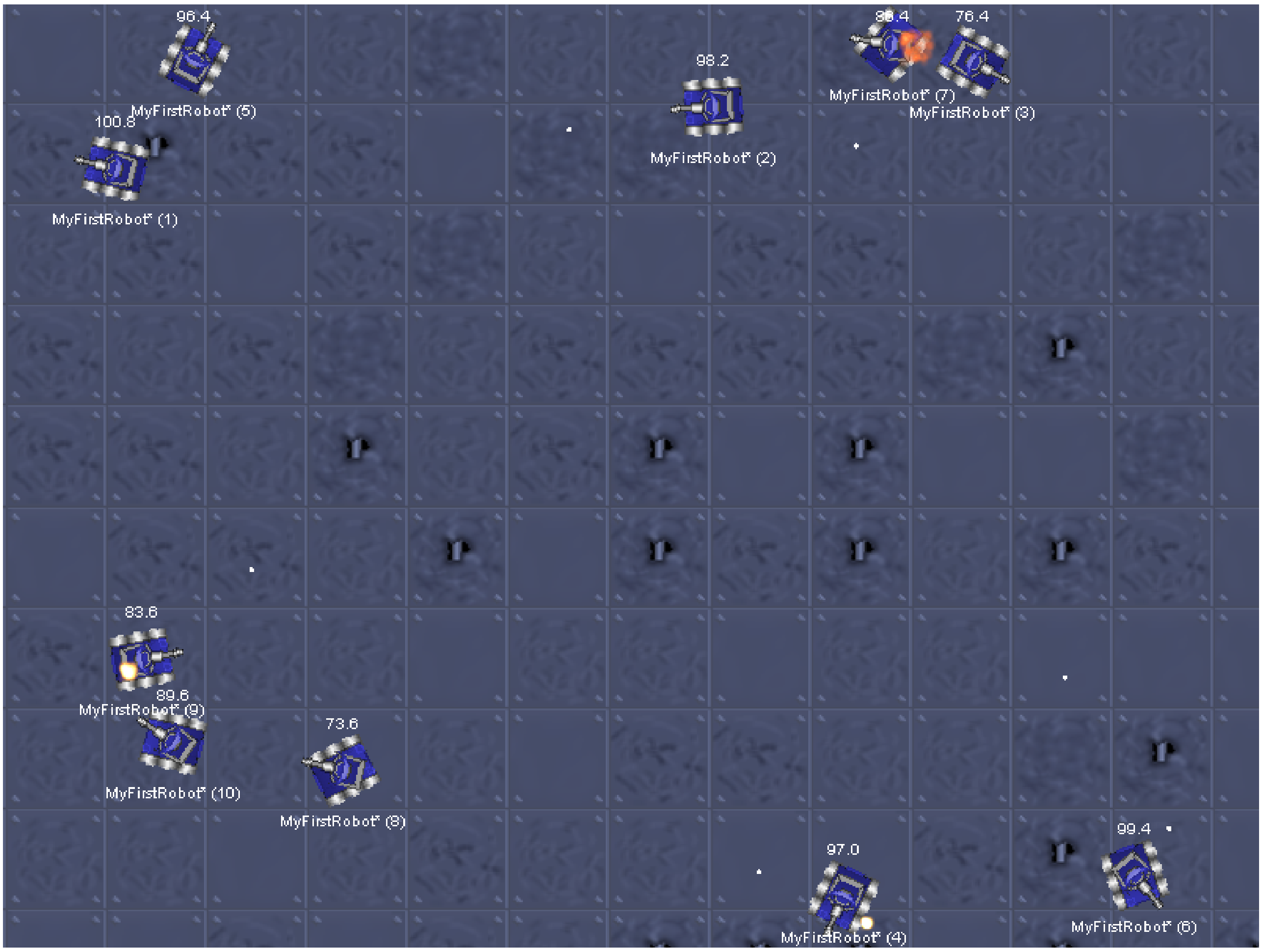


Figure 2.2: The Battlefield of RoboCode

```

  public void onHitWall(HitWallEvent e) {
    back(20);
    turnLeft(e.getBearing());
    ahead(100);
  }
}
```

Source code 2.5: This code describes what the agent should do when it hits a wall

The language has a lot more features than described here. The language documentation can be found on http://robocode.sourceforge.net.

CHAPTER 3

Wargame scenario

Before launching the wargame, the user should be able to express agents and predefine agent behaviors. The user of the game should then be able to choose whether to use the predefined behavior, or take control of the agent himself. The user should also be able to define the behavior of an agent when it come close to other hostile agents.

3.1 Rules

These rules should apply to the wargame:

- The game is turn-based. The turn end when the user is done executing commands, and press the *End Turn* button.
- The game is played on a grid 3.1.
- Each agent can move three grid-points in each turn.
- A higher ranked agent has a higher chance of winning.
- Agents fight when they are standing on the same grid location.
- The game rules are based on a classical deathmatch game. There is one winner only, and the winner is the team which eliminates all other team.

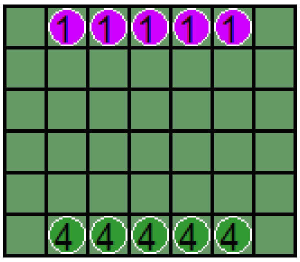


Figure 3.1: Example of the grid setup with two opposing teams

To get an overview of how the game operates, the layout of a game round is added in psuedocode.

```
function gameRound()
{
  gameFrame();
  EndTurn();
}
```

Source code 3.1: Game Round

The two functions called in the gameRound function, can be seen below.

```
for(i = 0; i <= 3; i = i + 1)
{
  CheckForEncounters();
  RandomAgentMovement();

  //Check if the list is empty
  if(moveAgents contains no items)
    return;

  UpdateAgentPositions();

  CheckForAgentCollisions();
}
```

Source code 3.2: Game Frame

The CheckForEncounters function will check if any of an agent is encountering, is within the reach of, another agent.

```
foreach(agent a in agents)
{
  if(a is within bounderies of another agent)
  {
    a.RemoveAllMovements();
    a.encounter.Compile();
```

```
  }
}
```

Source code 3.3: Check for encounters

If the current agent has no movements in his movement list, he finds a random agent from another team, and moves to their current location.

```
foreach(agent a in agents)
{
  if(a has no movement)
  {
    agent moveToAgent = getRandomAgent();
    a.MoveToAgent(moveToAgent);
  }
}
```

Source code 3.4: Random agent movement

The UpdateAgentPosistions function calculate the next agent move, taken from the moveAgents list. If the agent is still inside the warzone he can be moved. If the agent has reached his location his move gets removed from the list.

```
foreach(agent a in agents)
{
  if(a.team == currentteam)
  {
    foreach(agent moveAgent in moveAgents)
    {
      a.CalculateNextPosition();
      if(a.NextPosition.IsInBounds())
      {
        a.MoveAgent();
      }
      if(a.IsAtEndPosition())
      {
        moveAgents.Remove(a);
      }
    }
  }
}
```

Source code 3.5: Update agent positions

The CheckForAgentCollisions function will check if any agents from diffrent teams are standing on top of each other. If they happen to do so they will roll for the highest value, using their rank as a factor, to get the outcome of the fight. The agent with the lowest rolled value dies.

```
for(agentCount = 0; agentCount < agnets.TotalAgents; agentCount
    ++)
{
  foreach(agent a in agents)
  {
    if(a.CollideWithAgentOnOtherTeam())
    {
      if(a.Roll > CollidedAgent.Roll)
      {
        agents.Remove(CollidedAgent);
      }
      else
      {
        agents.Remove(a);
      }
    }
  }
}
```

Source code 3.6: Check for agent collisions

The EndTurn function will check if any of the teams, as the only team, has agents left, which will result in a win for the current team. If there are no teams standing alone on the warzone, the turn is passed on to the next team.

```
if(only team 1 has agents)
{
  Team 1 wins!
}

...

else if(only team n has agents)
{
  Team n wins!
}

else
{
  switchTurn();
}
```

Source code 3.7: End turn

Our problem statement focus on how one can make a compiler and a language optimized for MASes. We have gained some background knowledge *on multi agent systems (MAS), agent oriented languages (AOL) and language processors. A MAS uses agents to simulate some sort of scenario, where the agents strive to achieve a goal. One example of such systems is NetLogo[**?**]. AOLs are a type of languages developed specific for creating these MASes.*

The MAS we develop is a turn-based wargame, where the user has the opportunity to define the agents and behaviors with our language, and then play the game in our wargame environment.

Part II

Design

*In this part we outline the constituents of a programming language, covering the grammar and semantics. We explain the EBNF grammar notation, and the advantages of this. Section 4.2 is based on reference [**?**]. Furthermore we describe the grammar and semantics of our language, MASSIVE, and how the language is used.*

CHAPTER 4

Language Components

4.1 Design Critia

In this section the design critias for MASSIVE and action language will be explained. These critias are based on thoughts made prior to making both languages.

4.1.1 MASSIVE Language

Before creating the MASSIVE language we made some design critias. Those critias will determine which goals the language should fulfill when it is finished. The language should:

- Be simple to use for building a wargame scenario.
- Contain build-in classes for teams, agents, squads and actionpatterns.
- Contain build-in functions for manipulating build-in classes.
- Only use a few data types, to make it easier to choose which one to use.
- Have a simple syntax, specified for quickly writing a wargame scenario

Those language goals need to be considered every step of the way when the MASSIVE language is designed, but some of them can not be seen untill the language is finished. When making the MASSIVE language, language critias had a large influence on how the language was designed. Multiple critias was made to make the MASSIVE as good as possible:

- Writeability - This will ensure that the language is clear and corret in it's formulation.
- Reliability - This will ensure that the language does not behave unexpected.

4.1.2 Action language

When designing the GUI a scripting language for controling the agents was needed. This language is called Action language, and is build around making it avaliable for the user to move agents, teams and squads. The action language is a simple language based around a few commands.

- Move
- Encounter
- Hold
- Up
- Down
- Left
- Right

Those commands are made so it is clear to see what the code compiled is gonna do. The action language is made with the language critias of high Readability and Writeability. For the action language, the syntax should be simple and quick to write.

4.2 Grammar

BNF (Backus-Naur Form) is a formal notation technique used to describe the grammar of a context-free language [?]. There are several variations of BNF, for example Augmented Backus-Naur Form (ABNF[1]) and Extended

[1]ABNF has been popular among many Internet specifications. ABNF will not be further expanded on in this project.

Backus-Naur Form (EBNF). EBNF is used to describe the grammer of the language developed in this project [?].

The EBNF is a combination of BNF and regular expressions (REs, see table 4.1), and combines advantages of both regular expressions and BNF. The expressive power in BNF is retained while the use of regular expression notation makes specifying some aspects of syntax more convenient.

	Regular expression	Product of expression
empty	ε	The empty string.
singleton	t	The string consisting of t alone.
concatenation	$X \cdot Y$	The concatenation of any string generated by X and any string generated by Y.
alternative	$X \vert Y$	Any string generated either by X or Y.
iteration	X^*	Any string generated either by X or Y.
grouping	(X)	Any string generated by X.

Table 4.1: Table of regular expressions [?]. X and Y are arbitrary REs, and t is any terminal symbol.

Here are a few examples of the use of REs:
A B | **A C** generates **AB, AC**
A (**B** | **C**) generates **AB, AC**
A* **B** generates **B, AB, AAB, AAAB, ...**

In order to make the grammar easier to implement in the compiler, the left factorization techniques can be utilized.

Left Factorization

Given that we have choices on the form $AB \mid AC$, where A, B and C are arbitrary extended REs, then we can replace these alternatives with the corresponding extended RE: $A(B|C)$. These two expressions are said to be equivalent because they generate the exact same languages.

Elimination of Left Recursion

Here is an example of how left recursion can be eliminated with EBNF. If we have a BNF production rule $N ::= X|NY$, where N is a nonterminal symbol, and X and Y are arbitrary extended REs, then we can replace this with an equivalent EBNF production rule: $N ::= X(Y)^*$. These two rules are said to be equivalent because they generate the exact same language.

Substitution of Nonterminal Symbols

In an EBNF production rule $N ::= X$ we can substitute X for any occurrence of N on the right-hand side on another production rule. If we do this, and if $N ::= X$ is nonrecursive where this rule is the only rule for N, then we can eliminate the nonterminal symbol N and the rule $N ::= X$.

Whether or not such substitution should be made is a matter of convenience. If N is only represented a few times, and if X is uncomplicated, then this specific substitution might simplify the grammar as a whole.

Starter Sets

The starter set of a regular expression X (*starters[[X]]*) is the set of terminal symbols that can start a string generated by X. As an example, we have the type starters $n|N|s|S|b|B$, where the types are `num`, `string` and `bool`. Since the starters are case insensitive, we have both the uppercase and lowercase letters in the starter set for type. The full starter set overview can be found in appendix 13.2.

4.3 Semantics

The semantics of a programming language is a mathematical notation that explains langauge behavior. It defines the behaviour of all the elements in a language [**?**].

As an example of semantics, we view the semantics of the language *Bims*. The first part of the language semantics are the syntactic categories, which define the different syntactic elements in the language.

- Numeric values $n \in$ Num.
- Variables $v \in$ Var.
- Arithmetic expressions $a \in$ Aexp.
- Boolean expressions $b \in$ Bexp.

- Statements $S \in \text{Stm}$.

The next part of the semantics are the formation rules. These rules define the different operations that can be executed in the language. Here are the rules for statements:

$$S ::= x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S$$

These rules imply what kind of transitions can be done in the language. A transition happens when an operation is executed, and the program is moved into its next configuration. All transitions and configurations are defined by a transition system, which consists of three things.

- Γ represents all possible configurations.
- $\rightarrow$ represents all possible transitions.
- T represents the terminal configurations, which are the configurations with no transitions leading away from them.

The environment-store model is a way of storing variables, and it is the one we will be using in our semantics. We will therefore explain it here. The model consists of the variable environment and the store function. The variable environment is the environment where variables are referenced, mimicking memory addresses in a computer. The store function then uses the reference to find the actual value of the variable.

Finally, we will be using bigstep semantics to describe the different transition rules. Bigstep semantics represent transitions with a one to one mapping. The opposite of this is the smallstep semantic, where each transition has several semantic steps described, but we will not detail this.

The first example is the bigstep transition rule for declaring a variable.

$$\text{(VAR-DECL)} \qquad \frac{\langle D_v, env''_v, sto[l \mapsto v]\rangle \rightarrow_{DV} (env'_v, sto')}{\langle varx := a; D_v, env_v, sto\rangle \rightarrow_{DV} (env'_v, sto')}$$

$$\begin{array}{c}\text{where } env_v, sto \vdash a \rightarrow_a v \\ \text{and } l = env_v \text{ next} \\ \text{and } env''_v = env_v[x \mapsto l][\text{next } \mapsto \text{new } l]\end{array}$$

This transition rule expects one variable declaration to be followed by another. This next declaration can then either be empty, in order to end all the declarations, or a new variable declaration. That is what the D_v in the rule means.
The premises of this rule are the things that are written above the line. These are the premises the transition will happen under. This means the variable declaration will end with the environment being updated with the next available location l being set to the value v, which is the value contained in a.
The *next* location in the environment refers to the next available location, while *new* refers to the neighbour of any variable given to it.
Furthermore, we will be using dynamic scope rules, which means all variables are available in scopes opened after they are declared.

CHAPTER 5

Language Documentation

5.1 Grammar

When defining the grammar of a programming language, one defines every component in the language. It is important that the language is not ambiguous, as this could lead to misunderstanding at compile-time. The first thing we define in the language is the different datatypes, in our language there are three types; num, string and bool. These datatypes help define what is allowed in the language. Once these are defined, they can be broken up into even smaller parts, i.e. num is made up by digits or digits followed by the char '.' followed by digits, which in the grammar looks like this;

$$number ::= digits \mid digits.digits.$$

Then this is again split into even smaller parts, taking digits defined as;

$$digits ::= digit \mid digit\ digits.$$

And then the last part;

$$digit ::= 1|2..9|0.$$

This is done for every datatype if the language.
We choose only to make these datatypes as this would make the users decision of which datatype to use easier. Num can hold both integers decimals,

strings handles every aspect of text and bools is the only logical values in our language.

In the grammar it is also defined how the general structure of the program is to be build. In the grammar it is defined where each part of a program can be placed, within what sections different things can be nested. A general program written in our language must consist of a mainblock, in which everything else is contained. The mainblock will be made up by the keyword Main, followed by the two brackets '(' ')', followed by a block. The block consists of a left bracket '' some commands and then a right bracket ''. In the grammar the mainblock and block look like this: mainblock ::= Main() block block ::= commands

Each of the elements in the grammar is described this way. The full document is in the appendix 13.

5.2 Semantics

The transition rules for the MASSIVE language are operational semantics written in bigstep notation. See section 4.3 for more theory on semantics.

Here we will be describing the transition rules for some of the transitions in MASSIVE. First the transition system for commands is shown.

$\Gamma_{COM} = EnvV \times Sto$
$T_{COM} = EnvV \times Sto$

Because commands can transition into any available state and any command can command can be the last command in the code, all combinations of sto and EnvV are possible transitions.

The first transition we will demonstrate is the one that happens with if commands. This actually requires two separate transitions, because the if command can behave in several different ways depending on the input it is given.
The first transition is for an if command with no `else` block attached, where the expression it is given to evaluate, evaluates to true.

$$\text{(IF-TRUE)} \quad \frac{env_v \vdash \langle S_1, sto\rangle \rightarrow sto'}{env_v \vdash \langle \texttt{if (b)} \; \{S_1\}, sto\rangle \rightarrow sto'}$$

$$\text{if } env_v, sto \vdash b \rightarrow tt$$

Here we see that if the boolean value b evaluates to true for this transition to happen. The execution of S_1 leads to sto being altered, because we now S_1 can change the values of any variables in our environment.
If we then change the if command to where b evaluates to false, and it has an `else` block, the transition rule looks like this:

$$\text{(IF-ELSE-FALSE)} \quad \frac{env_v \vdash \langle S_2, sto\rangle \rightarrow sto'}{env_v \vdash \langle \texttt{if (b)} \; \{S_1\}\texttt{else} \; \{S_2\}, sto\rangle \rightarrow sto'}$$

$$\text{if } env_v, sto \vdash b \rightarrow ff$$

Here we see that the premise only has S_2 and not S_1 to alter sto with. This is because we know b will evaluate to false, and so S_1 will never be evaluated, and therefor not have any effect on the environment.
Next we look at the method for adding an agent to a squad. This method comes built into the language, and alters a squad by adding an agent to it.

$$\text{(ADD-AGENT-SQUAD)} \quad \frac{env_v \vdash \langle s, a, sto\rangle \rightarrow s', sto'}{env_v \vdash \langle \texttt{s.add(a)} \;, sto\rangle \rightarrow s', sto'}$$

This transitions uses an agent a and a squad s, and adds a to s, which leads to both s and sto being altered.
The next example is the string declaration, which can add new variables to the environment. Therefor, new transitions are needed for the available transtions.

$\Gamma_{DS} = (DecS \times EnvV \times Sto) \cup (EnvV \times Sto)$
$T_{DS} = EnvV \times Sto$

$$\text{(STRING-DECL)} \quad \frac{\langle D_s, env_v'', sto[l \mapsto s]\rangle \rightarrow (env_v', sto')}{\langle stringx = s; env_v, sto\rangle \rightarrow (env_v', sto')}$$

where $env_v, sto \vdash s \rightarrow_s v$
and $l = env_v$ next
and $env_v'' = env_v[x \mapsto l][\text{next} \ \mapsto \text{new } l]$

The last transition that is demonstrated, is the declaration of an actionpattern. An actionpattern needs a name when it is created, which is seperated as a string declaration.

$\Gamma_{DAP} = (DecAP \times EnvV \times Sto) \cup (EnvV \times Sto)$
$T_{DAP} = EnvV \times Sto$

$$\text{(AP-DECL)} \quad \frac{\langle D_{ap}, D_s, env_v'', sto[l \mapsto ap]\rangle \rightarrow (env_v', sto')}{\langle \text{new actionpattern ap}(name), env_v, sto\rangle \rightarrow (env_v', sto')}$$

where $env_v, sto \vdash ap \rightarrow_{ap} v$
and $D_s \vdash name \rightarrow_s ap.name$
and $l = env_v$ next
and $env_v'' = env_v[x \mapsto l][\text{next} \ \mapsto \text{new } l]$

5.3 Language Reference

This section will provide codeexamples of all the features of the MASSIVE language.

The first thing declared must always be the *main* function. MASSIVE will not compile without this function, as every bit of code goes into it. The main function is declared as follows:

```
main()
{
    /* Entire program code */
}
```

Source code 5.1: How to declare the main function in MASSIVE

There are two different loops in our language, the for-loop and the while-loop.

The while-loop is used in the following manner:

```
while(/* Expression */)
{
    /* Code */
}
```

Source code 5.2: While-loop

The for-loop is used in the following way:

```
for(num i = 0; /* Some Expression */; i = i + 1)
{
    /* Code */
}
```

Source code 5.3: For-loop

Declaring variables can be done as long as the assigned value matched the datatype selected. Only three datatypes exist in MASSIVE, and can be declared as follows

```
num count = 42;
string text = ``hello world'';
bool logicoperator = true;
```

Source code 5.4: Variable assignment

Besides declaring variables, they and also be used in the language, or in mathematical expressions. Below is examples of all the mathematical expressions usable in MASSIVE. The parser will not be able to compile if any redundant parenthesis are used.

```
num result = 0;

result = (42 * 55)/(67-55) + 49;

/ * The below will fail * /
result = ((42 * 55)/(67-55) + 49);
```

Source code 5.5: Examples of mathematical expressions

To create new agents, team and squads MASSIVE is using constructors. These can be used with a different number of inputs, as demonstrated in the next two code examples.

```
new team testTeam([name as string], [Hexcode as a string]);
new squad testSquad([name as string]);
new agent testAgent([name as string], [rank as num]);
```

Source code 5.6: Object assignment

Agent can also take a team as an argument, as demonstrated below.

```

new agent testAgent([name as string], [rank as num], [team as a team]);
```

Source code 5.7: Creating an agent with all possible arguments

You can also add agents to squads later on, as demonstrated in the below code example

```

testSquad.Add([agent as an agent]);
testTeam.Add([agent as an agent]);
```

Source code 5.8: Adding agents to a squad and team

There is only one kind of conditional statement in the MASSIVE language, the `if ... then ... else`-statement. Below is an example of the statement used along with all the logical operators available in MASSIVE.

```
num testNumber = 10;
bool boolean = true;

if(testNumber == 20)
{
    /* Code */
}
if(testNumber =< 20)
{
    /* Code */
}
if(testNumber => 20)
{
    /* Code */
}
if(testNumber != 20)
{
    /* Code */
}
if(boolean == false)
{
    /* Code */
}
else
{
    /* Code */
}
```

Source code 5.9: Statements

As illustrated above the MASSIVE language uses the dot-syntax extensively. This syntax can also be used to change properties about agents, teams, squads and actionpatterns. The below code examples will be used to demonstrate that.

This code demonstrates how to change the properties of an agent, a squad and an actionpattern, the only property being the name.

```

agent.name = "new name";
squad.name = "new name";
actionapttern.name = "new name";
```

Source code 5.10: Changing properties using dot-syntax

The below code demonstrates how to change the properties of teams; these being the name and the color.

```

team.name = "new name";
team.color = "new hexcolor as a string";
```

Source code 5.11: Changing properties using dot-syntax

The EBNF notation is a very usefull technique to descripe the grammar of a programming language. The use of regular expressions makes it possible to do left factorization, elimination of left recursion, and substitution of non-terminal symbols in a convenient way.

In our semantics we use the environment-store model to store variables. This model consist of the variable environment, where variables are referenced, and a store function, which uses the reference to find the value of the variable.

We use big-step operational semantics to descripe our semantics, which has a one to one mapping of the transition.

A program written in MASSIVE can contain while-loops, for-loops, variable assignment, object assignment, if-statements, and else-statements.

Part III

Implementation

In order to give the reader a top-down understanding of our product, we find that it is very important that the reader understands basic concepts of compiling. In the chapter 6 we explain core concepts and ideas as to how to compile source code into executable code. After that we outline our implementation of the compiler. Further more we describe the graphical user interface to our MASSIVE environment.

CHAPTER 6

Compiler Components

There are a number of different kind of language processors, however, we focus on the ones important to our project, Translators.
A translator is exactly what it sounds like; it is a program that translates one language into another, this being Chinese into English, C# into Java, or MASSIVE into C#.

In particular, we will focus on two types of translators; compilers and interpreters. We describe the usage of them, as well as differences and similarities between them.

6.1 Compilers

A compiler is a translator, typically capable of translating a language with a high level of abstraction, into a language that has a low level of abstraction. This could for example translate the language C into runnable machine code. A compiler has the defining property that it has to translate the entire input before the result can be used, however, it will then be run at full machine speed. If the input is very large it may take quit a while to finish translating.

A basic compiler can be broken down to three simple steps, which are illustrated in 10.1.

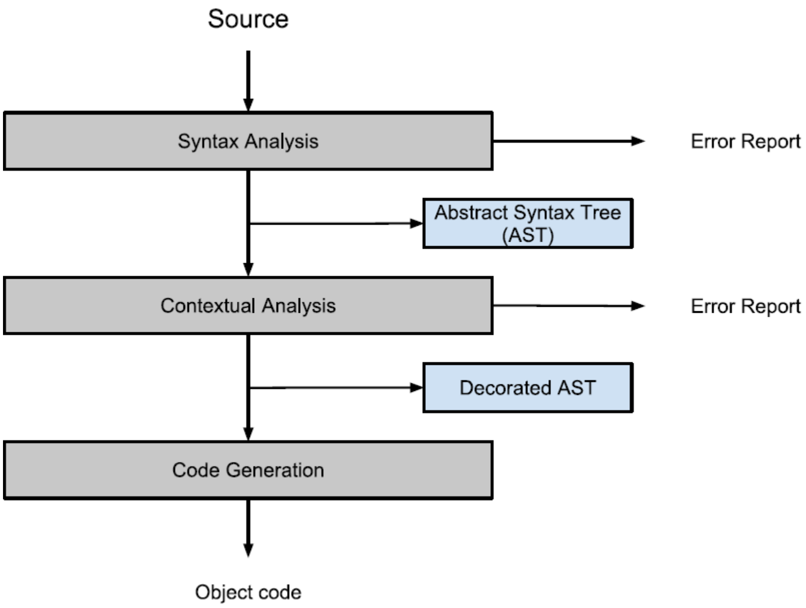


Figure 6.1: Illustration of the general structure of the compiler components.

6.2 Interpreters

An interpreter is also a translator, but instead of translating the entire input, the interpreter runs one instruction at a time from the input, thus enabling it to start utilizing the input right when it receives it. This boosts the time it will initially take to start running the output, but reduces the speed at which it can be run.

6.3 Scanner

The purpose of the scanner is to recognize tokens in the source program. Tokens are abstractions of the code, and the scanner simplifies the code by recognising a string as a token. For example a "+" is recognised as an OPERATOR token. This process is called *lexical analysis* and is a part of the *syntactic analysis*.

Terminal symbols are the individual characters in the code, which the scanner reads and creates equivalent tokens of [?]. The source program contain separators, such as blank spaces and comments, which separate the tokens and make the code readable for humans. Tokens and separators are identified as nonterminal symbols.

The development of the scanner can be divided into three steps:

1. The lexical grammar is expressed in EBNF 4.2.
2. For each EBNF production rule $N ::= X$, a transcription to a scanning method `scanN` is made, where the body is determined by X.
3. The scanner needs the following variables and methods:
   (a) `currentChar`, which holds the current character to scan.
   (b) `take()`, which compares the current character to an expected character and adds it to the `spelling` of the token.
   (c) `takeIt()`, which updates the current character to the next character in the string, and adds it to the `spelling` of the current token.
   (d) `scanN()`, as seen in step 2, though improved so it records the kind and spelling of the token as well.
   (e) `scan()`, which scans the combination 'Separator* Token', discarding the separator and returning the token.

See more about the BNF and EBNF notation in section 4.2 and see the full implementation of the grammar in the appendix 13.

6.4 Parser

The scanner 6.3 produces a stream of tokens. This stream provides an abstraction of the original input, and is used in determining the phrase structure, which is the purpose of the parser [?]. We strive to make the language unambiguous[1] to avoid the complication an ambiguous sentence would bring.

There are two basic parsing strategies, *bottom-up* and *top-down*, both of which produce an abstract syntax tree (AST). An AST is a representation of the phrase structure of the code, where the tokens found by the scanner are turned from a list into a tree, as defined by the structure of your grammar. We will here expand on the *top-down* strategy, because that is what we have implemented.

[1] This means that every sentence has exactly one abstract syntax tree (AST). See section ?? for more about the abstract syntax tree.

The *top-down* parsing algorithm is characterized by the way it builds the AST. The parser does not *need* to make an AST, but it is convenient to describe the parsing strategy by making the AST. The *top-down* approach considers the terminal symbols of a string, from left to right, and constructs its AST from top to bottom (from root node to terminal node).

6.4.1 Data Representation

Here is an example of how the *top-down* parsing algorithm works, demonstrated with an AST [**?**].

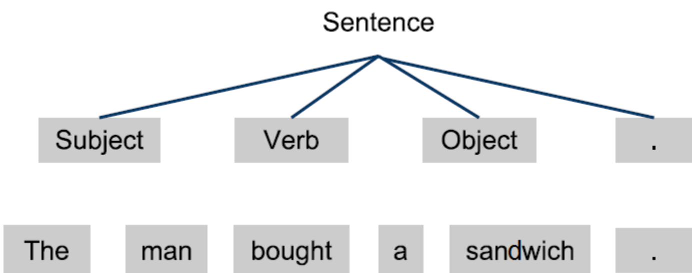


Figure 6.2: The first step for the parser is to decide what to apply ind the root node. Here it has only one option: "Sentence ::= Subject Verb Object."

The words that are not shaded are final elements in the AST. The words that are shaded and has a line to the previous node, is called stubs, and are not final elements, because they depend on the terminal nodes. The shaded nodes with no connection lines are the terminal symbols that are not yet examined.

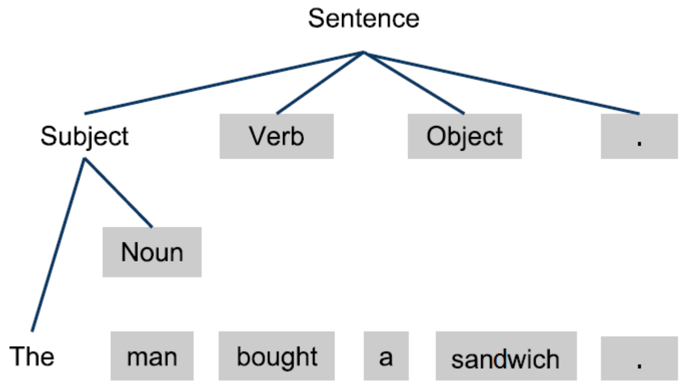


Figure 6.3: In the second step the parser looks at the stub to the left. Here the correct production rule is: "Subject ::= **The** noun".

The parser chooses the production rules by examining the next input terminal symbol. If the terminal symbol in figure 6.3 had been "A" then it would have chosen the production rule: "Subject ::= **A** noun".

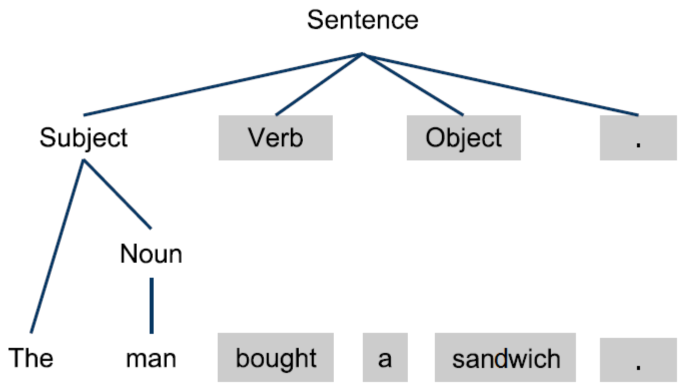


Figure 6.4: In third step the noun-stub is concidered, and the production rule becomes: "Noun ::= man".

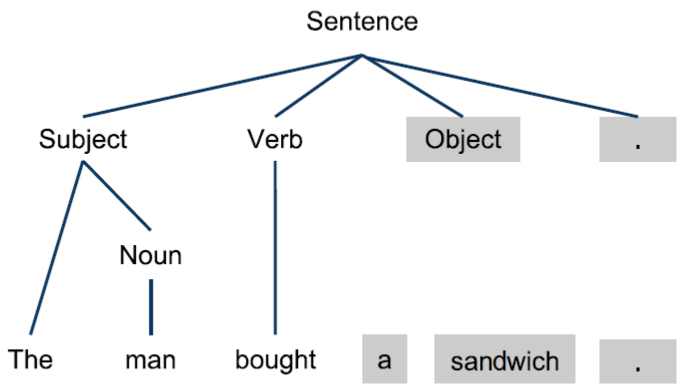


Figure 6.5: In fourth step the verb-stub is concidered, and the production rule becomes: "Verb ::= bought".

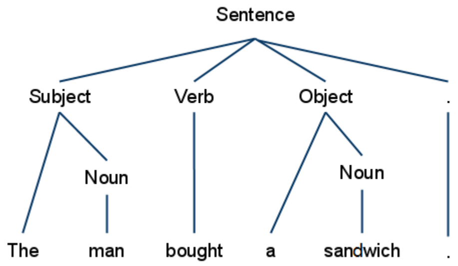


Figure 6.6: Here is the final syntax tree when the parser is done.

This method is continued until the whole sentence has been parsed. Here the final syntax tree is quite simpel, but one can imagine how the tree will grow when the input is a larger program text. See section 7.3 on how the AST is implemented.

6.5 Decoration

Decoration refers to decorating the abstract syntax tree. Until this pointonly the structure of the code compiled has been checked, and decoration is the part where the code is validated according to type and scope checking.
To do this, the AST have to be traversed. In order to traverse the AST, the visitor pattern is introduced.

6.5.1 Visitor Pattern

This design pattern is specifically used for traversing data structures and executing operations on objects without adding the logic to that object beforehand.
Using the visitor pattern is convenient as we do not need to know the structure of the tree when it is traversed. For example, every block in the code contains a number of commands. The compiler does not know what type of command the command object is refering to, the compiler only knows that there is an object of type command. When that object is "visited", the visitor is automatically redirected to the correct function based on the type of the object that is visited.
As an example the block visitor calls the visit method on each of the command objects in the block.

```
foreach (Command c in block.commands)
  {
    c.visit(this, arg);
  }
```

Source code 6.1: The loop in the block visitor, ensuring that all commands are visited.

This is done from within a visitor class, so `this` refers to an instance of the visitor. The reason the visitor is sent as input, is so all the visit functions can be kept in that visitor, and multiple visitors with different functionality can be used. If say, the next command is a `for`-loop (which inherits from the Command class), the visit function will lead to the `visitForCommand` function being called.

```
public class ForCommand : Command
    {
        ...
        public override object visit(Visitor v, object arg)
        {
            return v.visitForCommand(this, arg);
        }
    }
```

Source code 6.2: The `ForCommand` class from the AST.

And the `visitForCommand` function will then visit all the objects in the `for`-loop as they come.

```
internal override object visitForCommand(ForCommand forCommand,
    object arg)
    {
        IdentificationTable.openScope();

        // visit the declaration, the two expressions and the
            block.
        forCommand.CounterDeclaration.visit(this, arg);
        forCommand.LoopExpression.visit(this, arg);
        forCommand.CounterExpression.visit(this, arg);

        forCommand.ForBlock.visit(this, arg);

        IdentificationTable.closeScope();

        return null;
    }
```

Source code 6.3: The `visitForCommand` function.

6.6 Code Generation

We are compiling to C#, and thereby utilizing the underlying memory management in C#. Therefore we will not expand on memory management for this reason. Code generation is a matter of printing the correct code.
A great tool for doing this is code templates. Code templates are recipes for how code should be written, under the current circumstances, which makes the visitor pattern well suited for this task as well (see section 6.5.1).
When a specific pattern is recognized by the code generation visitor, it prints the correct C# code based on what code template matches the pattern. For example, when an object declaration occurs in MASSIVE, the compiler recognizes this declaration, the correct code template is found and the C# code is printed from that template.

```
// Object declaration in MASSIVE:
new Object Identifier(input);

// Object declaration in C#:
Object Identifier = new Object(input);
```

Source code 6.4: The code template for object declarations.

CHAPTER 7

Implementation of Compiler

7.1 Making the Scanner

The first method of the scanner 6.3 is a switch created to sort the current word according to the token starters (which can be found in appendix 13.2). E.g. if the first character of a word is a letter, the word is automatically assigned as an identifier, and a string with the word is created.
When an identifier is saved as a `Token`, the `Token` class searches for any keyword, that would be able to match the exact string, e.g. if the string spells the word "for", the `Token` class changes the string to a **for**-token.

```
public Token(int kind, string spelling, int row, int col)
        {
            this.kind = kind;
            this.spelling = spelling;
            this.row = row;
            this.col = col;

            if (kind == (int)keywords.IDENTIFIER)
            {
                for (int i = (int)keywords.IF_LOOP; i <= (int)
                    keywords.FALSE; i++)
                {
                    if (spelling.ToLower().Equals(spellings[i]))
                    {
                        this.kind = i;
                        break;
```

```
                    }
                }
            }
        }
```

Source code 7.1: The token method with overloads, where `(int)keywords` refers to an enum list of all keywords.

In the token overload method, IF_LOOP and FALSE is a part of an enum and then casted as an integer. Kind is an integer identifier. Spellings is a string array of the kinds of keywords and tokens available, as seen below.

```
public static string[] spellings =
        {
            "<identifier>", "<number>", "<operator>", "<string>"
                , ";", ":", "(", ")", "=", "{", "}",
            "if", "else", "for", "while", "bool", "new", "main",
                "team", "agent", "squad", "coord", "void",
            "actionpattern", "num", "string", "true", "false", "
                ,", ".", "<EOL>", "<EOT>", "<ERROR>"
        };
```

Source code 7.2: The string array spellings.

The structure of the `Token` method applies for operators and digits as well. If the current word is an operator, the scanner builds the operator. If the operator is a boolean operator i.e. $<$, $>$, $<=$, $>=$, $==$, $=<$, $<=$, $!=$, the scanner ensures that it has built the entire operator before completing the token. In case the token build is just a $=$, the scanner accepts it as the `Becomes`-token.
Digits are build according to the grammar and can therefore contain both a single number and a number containing one ".".

Every time the `scan()` method is called, the scanner checks if there is anything which should not be implemented in the token list, i.e. comments, spaces, end of line, or indents. Whenever any of these characters has been detected, the scanner ignores all characters untill the comment has ended or there is no more spaces, end of lines, or idents.
All tokens returned by the scanner is saved in a list of tokens, which makes it easier to go back and forth in the list of tokens.

7.2 Making the Parser

The parser (see section 6.4) takes the stream of tokens and keywords generated by the scanner, and builds an abstract syntax tree (see section 6.4.1) from it, while also checking for grammatical correctness. To accomodate all the different tokens, each token has a unique parsing method, which is called whenever a corresponding token is checked. Each of these methods then generate their own subtree which is added to the AST.

```
public AST parse()
        {
            return parseMainblock();
        }
```

Source code 7.3: This is the main parsing method, which parses a mainblock and returns it as the AST.

```
private AST parseMainblock()
        {
            Mainblock main;

            accept(Token.keywords.MAIN);
            accept(Token.keywords.LPAREN);
            Input input = (Input)parseInput();
            accept(Token.keywords.RPAREN);
            main = new Mainblock(parseBlock());
            accept(Token.keywords.EOT);

            main.input = input;

            return main;
        }
```

Source code 7.4: This method parses a mainblock and returns a mainblock object, consisting of all subtrees created by the underlying parsing methods.

In the `parseMainblock` example, we see that it returns a `Mainblock`-object, which inherits from the AST class, called `main`. The constructor for the `Mainblock` takes a `Block`-object as its input, so `main` is instantiated with a `parseBlock`-call.
The parser checks for grammatical correctness by checking if each token is of the expected type. For example, a command should always end with a semicolon, so the parser checks for a semicolon after each command. If there is no semicolon, the parser returns an error together with the line number and token which did not match an expected token.

7.3 The Abstract Syntax Tree

The AST is the virtuel image of a compiled source code. When the scanner has scanned the input successfully and created a list of tokens, the parser, as described in section 6.4, creates a syntax tree. The AST for the following source code example is represented in figure 7.1.

```
Main (  )
{
  new Team teamAliens("Aliens", "#FF0000");
  new Agent agentAlice("Alice", 5);

  teamAliens.add(agentAlice);
}
```

Source code 7.5: Source code example.

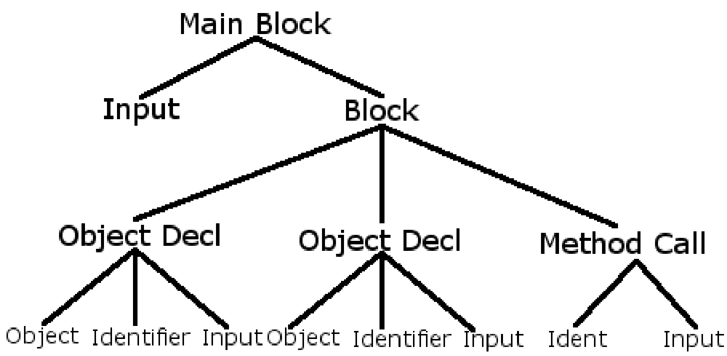


Figure 7.1: Example of the AST compiled from the source code above.

The AST can be printed by a pretty printer[1] to give a better overview of the compiled source code. In the MASSIVE compiler, the pretty printer prints all completed parses in the windows console. The MASSIVE pretty printer indents whenever a new branch is added. The source code above will be printed as seen in figure 7.2.

[1] A method for printing ASTs.

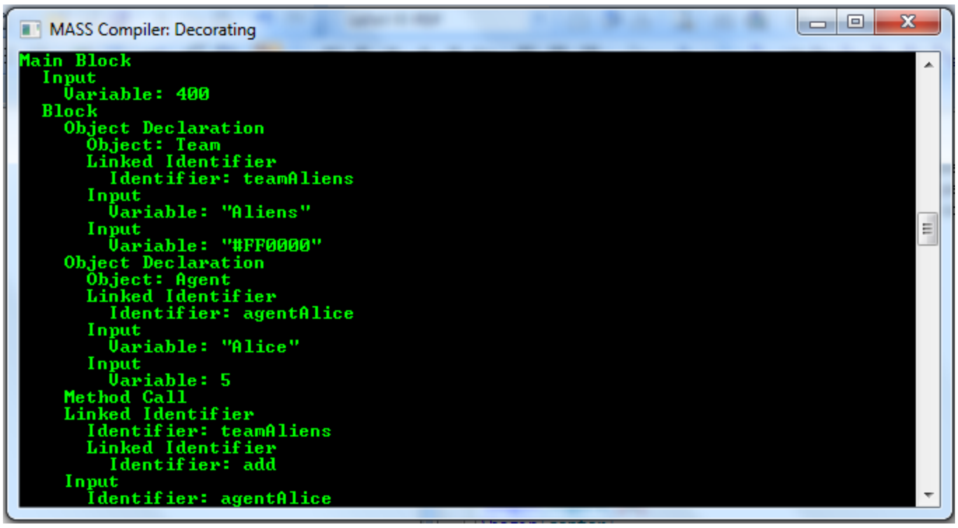


Figure 7.2: Example of the AST compiled with the MASSIVE compiler.

7.4 Decoration

To decorate the AST we use the visitor pattern (see section 6.5.1) with several different visitors handling different parts of the task.

7.4.1 Type and scope checking

The first one is the `TypeAndScopeVisitor` which visits every node of the abstract syntax tree, and checks if the types and scopes of variables in the code are correct. Therefore this is where type safety is enforced in the compiler.
This works by taking the type of the variable's token and comparing it to the values it is being used with.
In the scope checking we want to make sure that variables are not used outside their scopes, which is done with the `IdentificationTable` class. This class contains a list of declared variables, the current scope and methods for entering and retrieving variables, and methods for changing the scope.
Every time a scope is exited, every variable that was declared inside that scope is deleted from the list. This way the list will only contain the variables that are accessible from the current scope, as long as the scopes are updated correctly.

```
internal override object visitBlock(Block block, object arg)
    {
```

```
        IdentificationTable.openScope();
        ...
        IdentificationTable.closeScope();

        return null;
    }
```

Source code 7.6: A block is visited, and the scope is opened and closed respectively.

7.4.2 Input validation

The second decoration visitor is the `InputValidationVisitor`. The job of this visitor is to make sure that all methods and constructors in the language recieve the proper input, depending on the available overloads. The overloads in our language represent the option for methods and constructors to work with different inputs. For example are all the following declaration are legal in our langauge:

```
new Team teamAliens("Aliens", "#FF0000");
new Team teamRocket("Team Rocket");

new Agent agentJohn("John", 5, teamAliens);
new Agent agentJane("Jane", 5);
```

Source code 7.7: Examples of overloads.

Every overload of every method and constructor in the language is handled as a class of its own in the compiler. The compiler then takes the information it needs, to determine if the given input i valid, from these classes. It is therefore possible to add new overloads to existing methods and constructors, as well as add new methods and constructors, because you only need to create a new class for it and initialize it.

7.4.3 Variable Checking

The `VariableVisitor` is the third visitor, and its job is to check if the variables that are declared are also used. While this will catch every unused variable, the main reason for the creation of this visitor is to catch unused objects, so the compiler can warn about unused agents, squads and teams.

7.5 Code Generation

In order to print the C# code, the AST is traversed and it is determine what code should be printed. Therefore the visitors (see section 6.5.1) are again used to to accomplish this. This visitor is the `CodeGenerationVisitor` and is responsible for printing out the correct C# code, such that it can be compiled an run without errors. To accomplish this, code templates are used for each class in the AST.
A code template is a recipe for how the input code should be converted into C# code. Many templates are printed as the code is visited by the visitor. For example a `for`-statement first have `for (` printed, followed by a type declaration `num i = 0;`, a boolean expression `i < 10;`, an assignment statement `i = i + 1`, and a `)` at the end.
For the methods in our language, we have a different solution. Every class for a method or constructor, see 7.4.2, in our language must define an overload for the method `PrintGeneratedCode`.

```
public override string PrintGeneratedCode(string identifier,
    string name)
        {
            // squad one = new squad(two)
            return "squad " + identifier.ToLower() + " = new
                squad(" + name.ToLower() + ")";
        }
```

Source code 7.8: The code printed for the squad constructor.

In the code for the `squad`-constructor, two strings are given as input. The first is the variable name, and the second is the input given as a string. A more complex example is the `agent`-constructor, which takes both a name, a rank and a team as input.

```
public override string PrintGeneratedCode(string one, string two
    )
        {
            string[] input = two.Split(',');

            // agent one = new agent(two);
            // one.team = two
            return "agent " + one.ToLower().Trim() +
              " = new agent(" + input[0].ToLower().Trim() +
              ", " + input[1].ToLower().Trim() + ");\n" +
              one.ToLower().Trim() + ".team = " +
              input[2].ToLower().Trim();
        }
```

Source code 7.9: The code printed for the agent constructor taking three arguments as input.

The input string must be split up and put in the correct places, but the method still takes the same arguments as the other overloads. These templates are what makes it possible to print the code for any method or constructor used in our language.

7.5.1 Compiling to XML

This section follows a piece of code from the time it is first coded in MASSIVE, to the generation of a .cs file, and untill it is stored in an XML file. The scenario is to create two teams with an agent on each team. This scenario can look like the source code below, when written in the MASSIVE language.

```
main()
{
  // Creating two teams.
  new Team teamAliens("Aliens", "#FF0000");
  new Team teamRocket("Rockets", "#00CC00");

  // Creating two agents, rank 5 and 2, and adding them to the
     teams.
  new Agent Alien("Alien", 5, teamAliens);
  new Agent Rocket("Rocket", 2, teamRocket);
}
```

Source code 7.10: MASSIVE code, creating two teams with an agent on each.

When the MASSIVE compiler compiles the MASSIVE code to a .cs file, the MASSIVE compiler have to always start by adding the defaults to the .cs file. These defaults include the libraries used by the C# compiler to be able to compile to XML and the initialization of the lists.

```
using System;
using System.Drawing;
using System.Collections.Generic;
using MASClassLibrary;

namespace MultiAgentSystem
{
  class Program
  {
    static void Main(string[] args)
    {
```

```
        Lists.agents = new List<agent>();
        Lists.squads = new List<squad>();
        Lists.teams = new List<team>();
        Lists.actionPatterns = new List<actionpattern>();
```

Source code 7.11: The default code added to the .cs file.

When this is done the MASSIVE compiler can add the teams and agents to the lists, which looks like this in the .cs file.

```
team teamaliens = new team("aliens", "#ff0000");
team teamrocket = new team("rockets", "#00cc00");

agent alien = new agent("alien", 5);
alien.team = teamaliens;
agent rocket = new agent("rocket", 2);
rocket.team = teamrocket;
```

Source code 7.12: The creation of the teams and agents.

To complete the .cs file the MASSIVE compiler ensures that the teams and agents are stored in the XML files by using the method XML.generateXML.

```
        XML.generateXML(@"C:\Users\Rasmus\Documents\Studier\
            Projekter\P4 - Multi Agent System\Project\MAS");
        Console.WriteLine("XML generation complete.");
      }
    }
}
```

Source code 7.13: The method, which stores the data in the XML libraries.

Finally the MASSIVE compiler call the C# compiler, which will compile and run the .cs file and generate the XML libraries. Below is the final XML files generated by the .exe file.

```
<oldTeam>
  <id>1</id>
  <name>aliens</name>
  <color>#FF0000</color>
</oldTeam>
<oldTeam>
  <id>2</id>
  <name>rockets</name>
  <color>#00CC00</color>
</oldTeam>
```

Source code 7.14: Teams stored in the XML file.

```
<agent>
  <name>alien</name>
  <team>
    <name>aliens</name>
    <color />
    <colorStr>#ff0000</colorStr>
  </team>
  <rank>5</rank>
  <posx>0</posx>
  <posy>0</posy>
</agent>
<agent>
  <name>rocket</name>
  <team>
    <name>rockets</name>
    <color />
    <colorStr>#00cc00</colorStr>
  </team>
  <rank>2</rank>
  <posx>0</posx>
  <posy>0</posy>
</agent>
```

Source code 7.15: Agents stored in the XML file.

7.6 Error handling

It is important that a programmer knows if the code he is writing is correct or not, so it is convenient if the compiler tells him of any errors it encounters. Our compiler can catch errors after every parsing of the code, and it will also complete the parse, so it can report every error encountered in that parse.
The programmer also gets a choice of whether he wants to print the compilation of the code, and if he does, the code and error markers will be printed. We have also made it such that the programmer can recompile his code, once he has corrected any errors, without restarting the compiler.
The are also warning messages, but these only occur during the variable check (see section 7.4.3). The programmer can choose to either recompile or continue with the current compilation when a warning has been found.

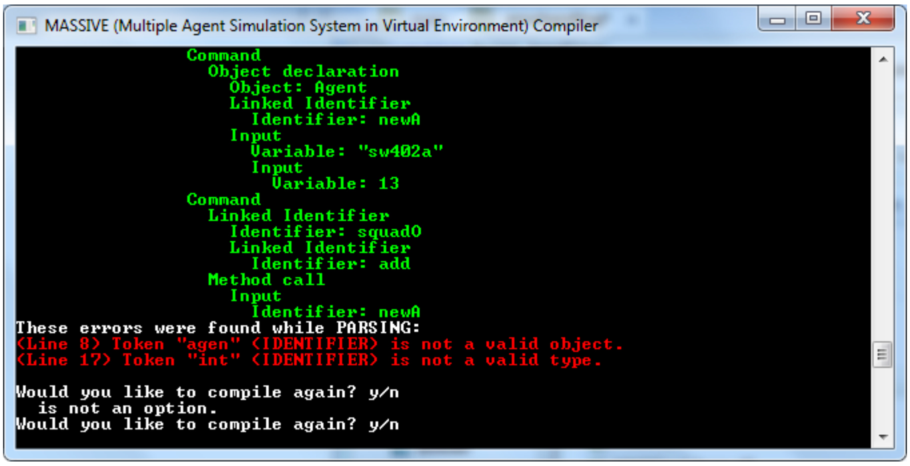


Figure 7.3: An example of how the compiler handles errors.

CHAPTER 8

Graphical User Interface and Action Language

The user interface is made as a windows form application[1]. Using Visual Studio's designer tool, it is simple to make a graphical user interface with buttons, panels, and windows.
The main idea of the user interface design, is that it should be intuitive, so the user should not spend a lot of time figuring out what all the buttons do. We have designed the interface so the main structure looks like other strategy computer games.

Game Start Settings

When the game is started, a dialog box is shown where one can choose the size of the *war zone*. The war zone has three fixed grid sizes, because of the way the grid is drawn, see 8.
The functions of the dialog box is:

1. *Small, Medium, Large* radio buttons - select one to choose the grid size.

2. *Start* button - starts the game.

[1]graphical application programming interface, included in the .NET Framework.

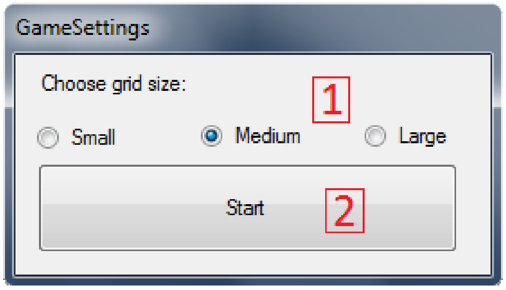


Figure 8.1: Screen shot of the game settings dialog box.

Game Interface Functions

The functions of the game interface are as follows:

1. *War zone* - contains the grid on which the wargame unfolds.
2. *Agents* - the agents of the different teams (here with a 4-player game setup).
3. *Stats field* - shows the stats of a selected agent.
4. *Agents left* - shows how many agents are left on each team.
5. *Combat log* - contains a combat log on who killed who in fights between agents.
6. *Command list* - contains the list of available commands the user can type in the *command center*.
7. *MousePos grid* - shows the grid point of the mouse.
8. *Command center* - here the user types the commands to navigate the agents around the grid.
9. *Execute x5* button - simulates five game rounds.
10. *End turn* button - ends the turn and gives the turn to the next player.
11. *Reset game* button - sets up a new game.
12. *Quit game* button - closes the game.
13. *Simulate* button - starts a simulation, where the game starts and runs until the game is over.

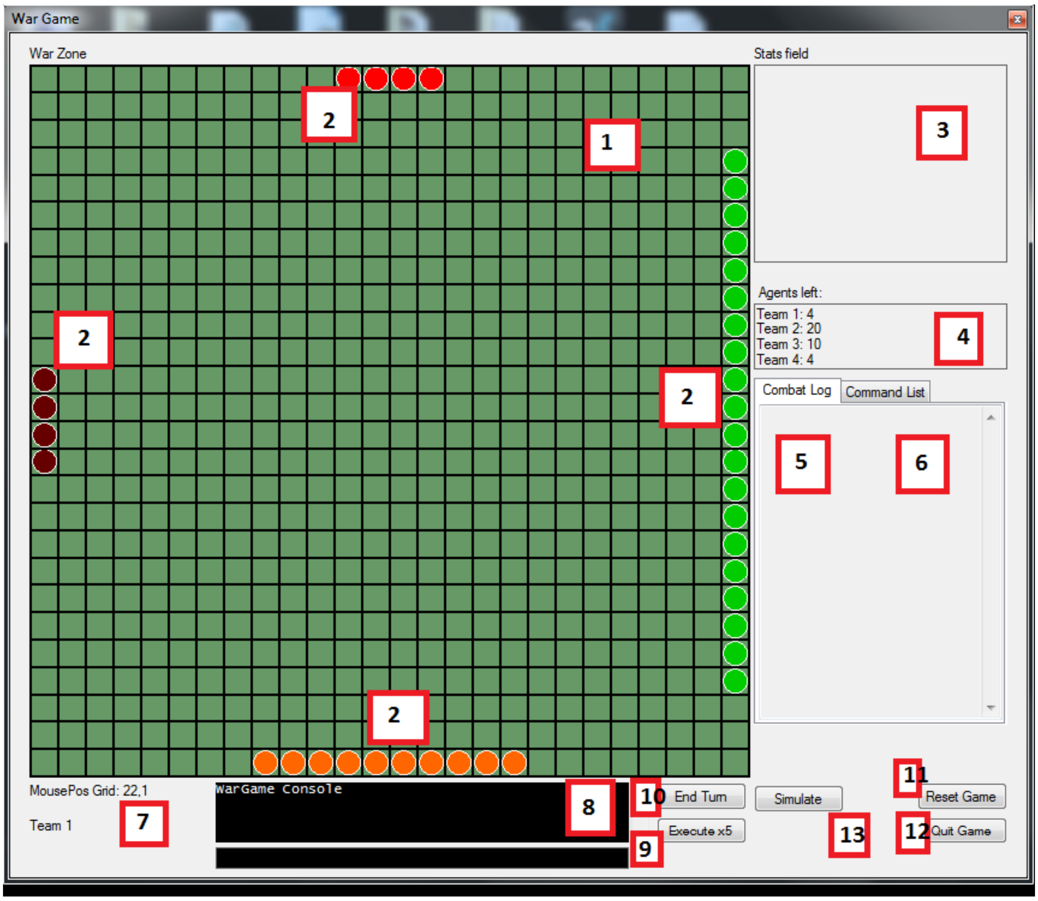


Figure 8.2: Screen shot of the game interface.

Drawing the Grid and Agents

The program make use of GDI+ [?] to draw the grid (the war zone) on the screen. A usercontrol is added to eliminate the flickering GDI+ normally creates on windows forms, which is done with the help of double buffering. GDI+ graphic is only used inside the usercontrol `DBpanel`. We have to make sure everything is drawn in the correct order, as the pixels are drawn on top of each other. The first thing drawn is the background, which in this case is green, with the black gridlines on top of it, to create the game grid. Finally the agents are drawn. The starting posistions of the agents are calculated by the following code:

```
            int it1 = (Grids / 2) - (agentsOnTeam1 / 2);
            int it2 = (Grids / 2) - (agentsOnTeam2 / 2);
            int it3 = (Grids / 2) - (agentsOnTeam3 / 2);
```

```
            int it4 = (Grids / 2) - (agentsOnTeam4 / 2);
            foreach (Agent a in agents)
            {
                Point p = new Point();
                if (a.team.ID == 1)
                {
                    p = getGridPixelFromGrid(new Point(it1, 0));
                }
                else if (a.team.ID == 2)
                {
                    p = getGridPixelFromGrid(new Point(Grids - 1, it2));
                }
                else if (a.team.ID == 3)
                {
                    p = getGridPixelFromGrid(new Point(it3, Grids - 1));
                }
                else if (a.team.ID == 4)
                {
                    p = getGridPixelFromGrid(new Point(0, it4));
                }

                a.posX += p.X;
                a.posY += p.Y;

                if (a.team.ID == 1)
                {
                    it1++;
                }
                else if (a.team.ID == 2)
                {
                    it2++;
                }
                else if (a.team.ID == 3)
                {
                    it3++;
                }
                else if (a.team.ID == 4)
                {
                    it4++;
                }
            }
```

Source code 8.1: This code snippet calculates the agents's start positions

`it` is the start location for each team. E.g. if the grid is 13 "grids" wide and team one consist of three agents, the starting position for team one will be $(13/2) - (3/2) = 6,5 - 1,5 = 5$.

8.1 Action Interpreter

The action interpreter, is the interface for all commands the user can give to the units in the GUI. It analyzes a single command at the time, and if the command is valid, it executes it directly in the GUI. A command in the action interpreter consists of three parts; *identification*, *state*, and *option*.
The *identification* identifies which unit, team, or squad the user is giving the command to.
The *state* indicates in which state the unit should execute the command, e.g. the `encounter`-command waits untill there is an enemy unit in its perimeter.
The *option* identifies the coordinate or direction the unit should go to, e.g. the option `up` would move the unit one grid up. The full list of commands is seen in figure 8.1.

Command	Result of command
`[identifier] Move [direction]`	Moves the selected agent in the selected direction
`[identifier] Move [coordinates]`	Moves the selected agent to specific grid coordiantes
`[identifier] Move [actionpattern]`	Moves the selected agent according to a specific predefined actionpattern
`[identifier] Encounter [direction]`	Moves the selected agent in a selected direction, when an opposing agent is in close
`[identifier] Encounter [coordinates]`	Moves the selected agent to a coordinate, when an opposing agent is in close
`[identifier] Encounter [actionpattern]`	Moves the selected agent according to a predefined actionpattern, when an opposing agent is in close

Table 8.1: Table with all the commands in the action language. `[identifier]` refers to *agent id, agent name, squad,* or *team.* [direction] can be *up, down, left,* and *right.* [coordinates] is a grid point, i.e. `2,3`

The commands in the action interpreter are simple. There is the `move`-commands, e.g. `12 move 1,2`, which would move the unit with ID `12` to the

coordinate `1,2`.
Furthermore the `encounter`-command can give the user the ability to do a certain sequence of movements, whenever the unit is in range of an enemy unit, e.g. `12 encounter 1,2` would move the unit with the ID `12` to the coordinate `1,2` when it is in range of an enemy unit.

8.1.1 Lexical Analysis

The scanner 6.3 is the first part of the *lexical analysis*, the scanner divides the source code, according to the Action Grammar found in appendix 13.4.
The scanner can best be described as an automaton 8.3, which accepts any combination of the terminal symbols defined as letters or digits, a number, or any keyword from the Action Grammar.
As an example would the phrase "12 move up", be analyzed as a combination of three tokens.

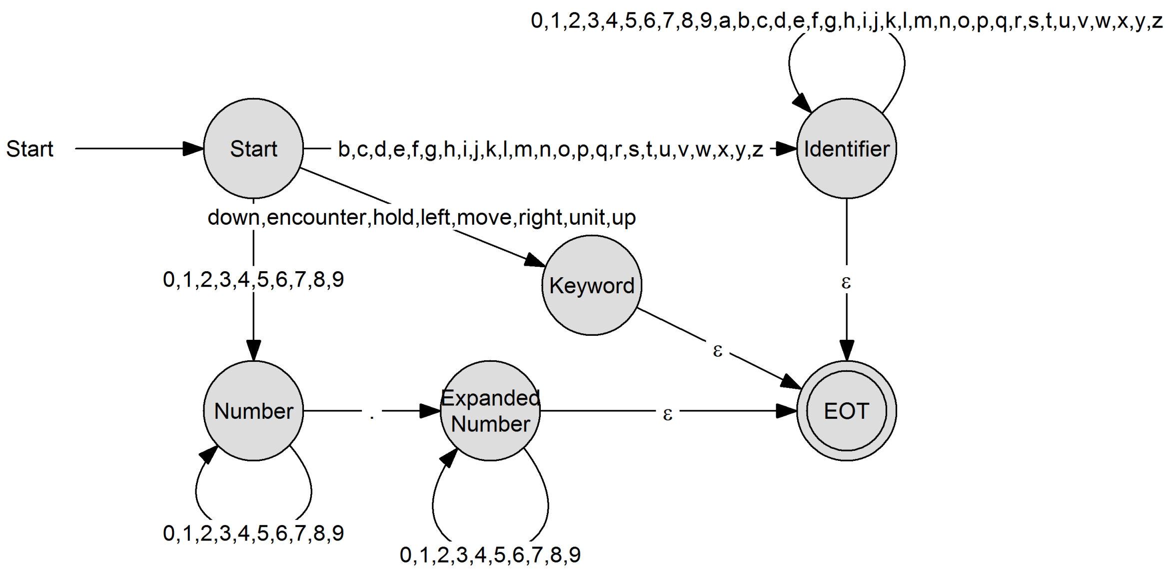


Figure 8.3: Atomaton of the Action Interpreters language.

These three tokens are stored in a list containing all tokens the scanner finds. Since this is an interpreter the scanner only analyse one command at the time.
Therefor the AST returned by the parser is really simple, the AST created from the tokens recieved by the scanner can look like 8.4.

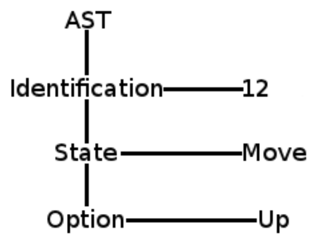


Figure 8.4: The AST parse from the stream of tokens recieved by the scanner.

The parser is overall pretty simple, since the majority of the parser is the three switches, identification, state, and option. An example of a switch is the identification of the identifier, which parses the first part of the command. SquadID and agentID, has an equivilent TeamID and SquadID, these classes are placeholders used by the visitor to determine which kind of identifier it is when decorating.

```
case (int)Token.keywords.SQUAD:
case (int)Token.keywords.S:
  // Accepts the token, since its either S or Squad.
    acceptIt();
    squadID squad = parseSquadID();
    return squad;
case (int)Token.keywords.NUMBER:
  // If only a number has been selected, parse it as an Agent ID
    .
  agentID agentNum = parseagentID();
    return agentNum;
case (int)Token.keywords.IDENTIFIER:
    // If the identification is an identifier, treat it as one.
    Identifier ident = parseIdentifier();
  return ident;
```

Source code 8.2: Example of how the parser identifies the identifier.

8.1.2 Contextual Analysis & Code Generation

The contextual analysis is the decoration of the AST, which is done by traversing the AST with the visitors, see 6.5.1 about visitors. Code generation is the last methods of the contextual analysis visitors, since there is no need to parse the AST more than once, when all information used by the move functions is given cronologically.

The first part of the decoration is to verify the identification of the command. To verify the identification the decorator finds the unit or units the user wants to move, e.g. the user gives the command `squad 1 move down`. The parser then determines that the identifier `1` is a squad, and stores its token as a SquadID. The decorator then searches for the squad identifier in the squad list, and calls the move method to execute the action `move down`.

```
if (object.ReferenceEquals(
  single_Action.selection.GetType(),
  new squadID().GetType()
  ))
  {
  // set arg to null if its an id.
  visitCodeGen_MoveSquad(single_Action, null);
  }
```

Source code 8.3: Example of the determination of the identifier in the visitors, this part identifies SquadID.

When the squad has been identified the decorator calls the `visitCodeGen_MoveSquad` method and moves all agents in the squad.

```
squad squad;
// If arg is null, the selection is an ID.
if (arg == null)
  {
    squadID select = (squadID)single_Action.selection;
    Token selectToken = select.num;
    squad = Lists.RetrieveSquad(Convert.ToInt32(selectToken.
        spelling));
  }
else
    {
    Identifier ident = (Identifier)single_Action.selection;
    squad = Lists.RetrieveSquad(ident.name.spelling);
  }

  foreach (agent a in squad.Agents)
  {
    visitCodeGen_MoveOption(a, single_Action.move_option);
  }
```

Source code 8.4: Code snippet of the identification of the units in a squad.

The `visitCodeGen_MoveOption` method analyze the state and the option. If the state is `encounter` instead of `move`, the function `addEncounter` is called with the parameters `currentAgent` (current agent object), and a string containing the agents name, the state `move`, and the option.

```
// If the state is an encounter call the add encounter function.
if (move_Option.state == (int)State.States.ENCOUNTER)
  {
    Functions.addEncounter(_agent, _agent.name + " move " +
        token.spelling);
    return;
  }
```

Source code 8.5: Code snippet showing what happens when the encounter state is chosen instead of move.

If any of the directions have been chosen as the option, the agent will be moved one coordinate in the direction.
Furthermore if an actionpattern is chosen the action interpreter calls itself recursivly, and adds the agent who is going to be moved, along with the actionpattern as the overload. This will interpret the action and instead of the `unit`-keyword, insert the agent instead.

```
object moveOption = move_Option.dir_coord.visit(this, null);

// If there was no actionpattern with this name, Exception.
if (moveOption == null || !object.ReferenceEquals(moveOption.
    GetType(), new actionpattern().GetType()))
  {
    throw new InvalidMoveOptionException("The actionpattern was
        invalid!");
  }
actionpattern ap = (actionpattern)moveOption;

// If the state is an encounter call the add encounter function.
if (move_Option.state == (int)State.States.ENCOUNTER)
  {
    Functions.addEncounter(_agent, _agent.name + " move " + ap.
        name);
    return;
  }

foreach (string s in ap.actions)
  {
    ActionInterpet.Compile(s, _agent);
  }
return;
```

Source code 8.6: The method moving a unit if the `move`-option is an actionpattern.

Compilers and interpreters are two types of translators. A compiler has to translate the entire input before the result can be used and an interpreter runs one instruction at a time from the input, thus enabling it to start utilizing the input before it is done translating.

The scanner produces a stream of the tokens it has recognized in the source program. The parser then recognizes the phrase structure of the token stream, and produces an abstract syntax tree.

To decorate the AST, we use the visitor pattern. There is a visitor for type and scope checking, one for input validation, one that checks for unused variables, and one for code generation. This design pattern is specifically used for traversing data structures and executing operations on objects without adding the logic to that object beforehand.

Our compiler can catch errors after every parsing of the code. This is convenient for the programmer when he needs to debug the code he writes.

The user interface is implemented using Windows forms and C#.

Part IV

Discussion

In this part we discuss our project. We describe a use case of the MASSIVE language. Also a comparison of the advantages and disadvantages of the MASSIVE language versus other object oriented programming languages, i.e. C#, and finally a conclusion of the project.

CHAPTER 9

Language Development

9.1 Compiler language

The MASSIVE compiler is developed in the C# language, which is because C# is an object oriented language and therefore provides a good base to create classes and objects used in the compiler. E.g. the abstract syntax trees.
Java provides the same base as an object oriented language, but due to developing the GUI in C# in Visual Studio, it is convenient to create the MASSIVE compiler in the same programming language.
There were problems managing reference types in C# though. Reference types are the kinds of objects that when created refer to an existing object in memory rather than creating a new instance of the object.
Several bugs occurred due to difficulty in anticipating when something is a referenced type as opposed to a seperate object.
It might therefor have been beneficial to develop the compiler in a language like Haskell, which uses the functional paradigm. This is because purely functional languages do not allow side effects in their functions, meaning that existing data is not altered. Haskell is one such language [**?**], where new data is created and the alterations are applied to, so reference types are of no concern.

CHAPTER 10

MASSIVE Language

In this report we illustrate how we design and implement the agent oriented language, MASSIVE. During this chapter we demonstrate a working simulation with a use case, and compare the agent oriented language to object oriented code (C#). Furthermore we discuss the advantages and disadvantages of the MASSIVE language.

10.1 Use Case

This use case demonstrates how to write a wargame scenario in the MASSIVE language, how to compile it, and how to run it.

The first thing to do is to write the MASSIVE code for creating the agents and alike. In the example two teams are created, these teams are called "Disco" and "Kman". Agents are added to each team and an action pattern is defined, for later use when running the wargame simulation.

```

/* Initializes the game */
Main ()
{

  // Creates team Disco.
  new team teamDisco("Disco", "#FF6600");
  num totalDiscos = 10;
  for ( num i = 0; i < totalDiscos; i = i + 1)
```

```
{
  num a = 0;
  if ( i < totalDiscos-1 )
  {
    a = 1;
  }
  else
  {
    a = 21-totalDiscos;
  }

  new Agent newAgent("Stue", a);
  teamDisco.add(newAgent);
}

new team teamKman("Kman", "#660000");
new squad squadNabs("noobs");
new squad squadRevo("Revolution");

for(num i = 0; i < 4; i = i + 1)
{
  num a = 0;
  if(i =< 1)
  {
    a = 2;
  }
  if(i >= 2)
  {
    a = 8;
  }

  new Agent newAgent("Kman", a);
  teamKman.add(newAgent);

  if (i <= 1)
  {
    squadNabs.add(newAgent);
  }
  if (i => 2)
  {
    squadRevo.add(newAgent);
  }
}

// Moves used in the actionPatterns.
string moveUp = "unit move up";
string moveDown = "unit move down";
string moveLeft = "unit move left";
string moveRight = "unit move right";
```

```

// Creates the action pattern Patrol Low.
// Patrols the lower part of the game area.
new actionpattern patrolLow("PatrolLow");
patrolLow.add(moveUp);
patrolLow.add("unit move 25,24");
patrolLow.add(moveUp);
patrolLow.add("unit move 0,23");
patrolLow.add(moveDown);

}
```

Source code 10.1: MASSIVE code example

When compiling the code the compiler warns that there are unused variables (see 10.1). This will not produce any fatal compilation errors, but it is just a notification to the programmer. However if any fatal errors are found while compiling the MASSIVE compiler will break the compilation and print the errors.

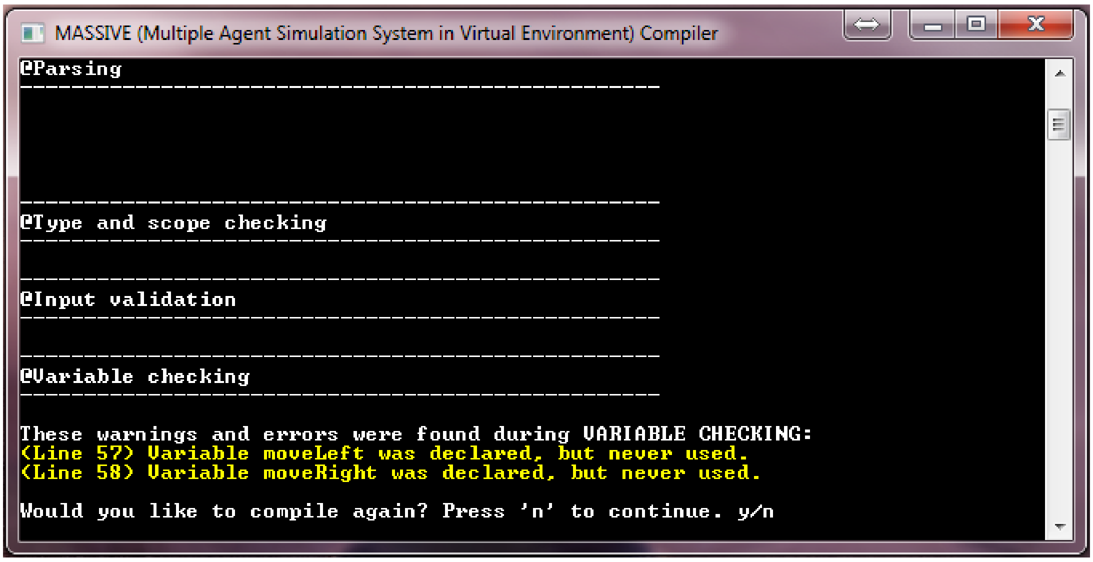


Figure 10.1: The MASSIVE compiler warning of unused variables.

The MASSIVE compiler provides the programmer with the oppurtunity to recompile the code, which provides the programmer with a way of correcting code containing errors without terminating the compiler everytime it fails. After a succesfull compilation a file named "MASSIVECode.cs" is created. The purpose of creating the .cs file is to be able to compile it with the help of the C# compiler. The compilation of the .cs file generates an

.exe file, which create the actual data output in XML format. The XML-data is then loaded by the MASSIVE battlefield application. When running the MASSIVE battlefield application, the user is given the choice to determin the grid size 10.2).

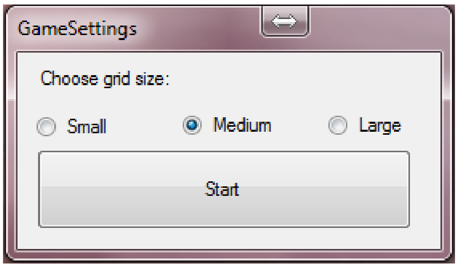


Figure 10.2: Choosing the size of the game-grid

The user will be presented with the actual simulation (see 10.3). Here the user will have the oppertunity to instruct the agents to use the action pattern defined in the previous code example or any of the movement commands, as shown in 10.3.

When the MASSIVE battlefield application is running, the user is able to either give commands to the agents, end the current teams turn, run 5 sequal turns, or simulate the wargame untill a winner is found. The figure 10.4 is the result of the command "team 2 move patrolLow" followed by an "Execute x5".

10.2 Comparison

This section compares how a wargame can be build using C# or by using the MASSIVE compiler. We will take a look at some of the pros and cons by using C# aswell as the pros and cons using the MASSIVE language.

10.3 C#

The MASSIVE language is being compared to C# which is an object orientated language. This was done due to both the compiler and the GUI is written in C#. C# does not have a built in multi agent orientated function or enviroment, which means it is required for the programmer to build the multi agent wargame from scratch. Building a multi agent wargame in C# would require the programmer to have programming knowledge, in the MASSIVE language there are less features, which should make it more simple for

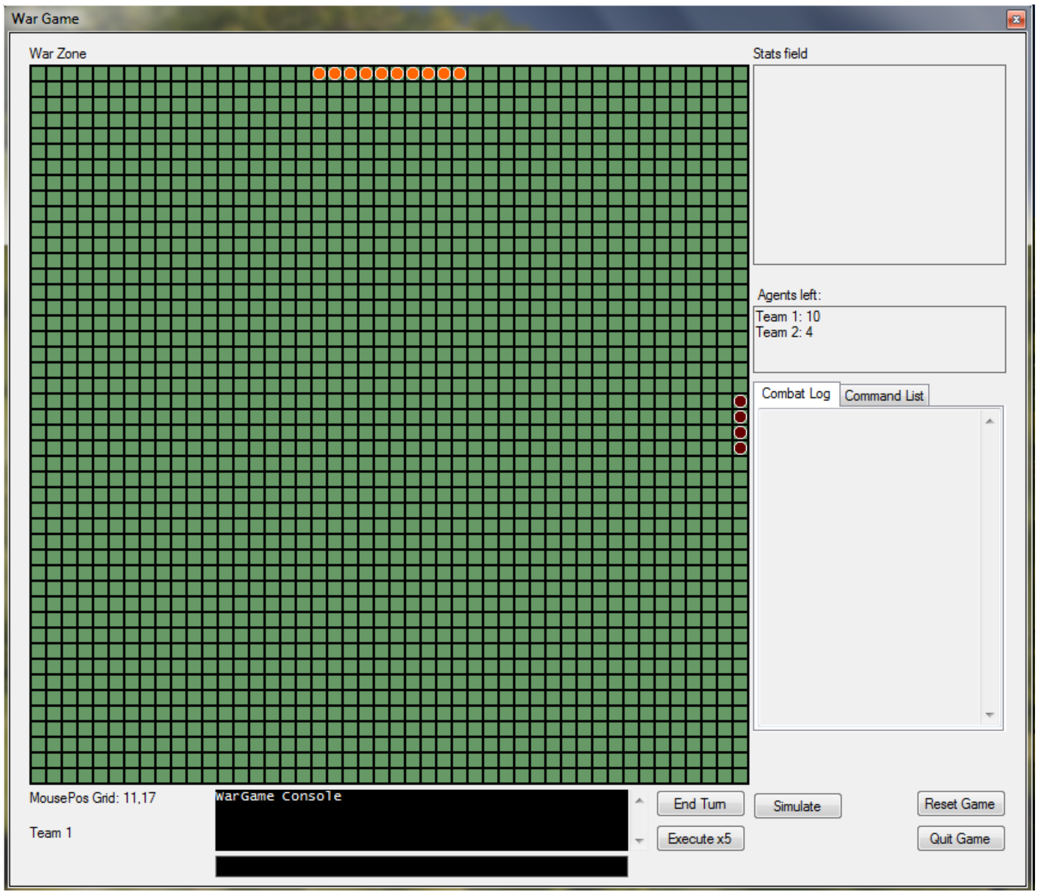


Figure 10.3: The simulation running with the input instructing som of the agents to use an actionpattern

first time users.

Pros

- No limits, you can create all the features you want.

Cons

- No existing multi agent enviroment.
- No existing multi agent types.
- No existing multi agent functions.

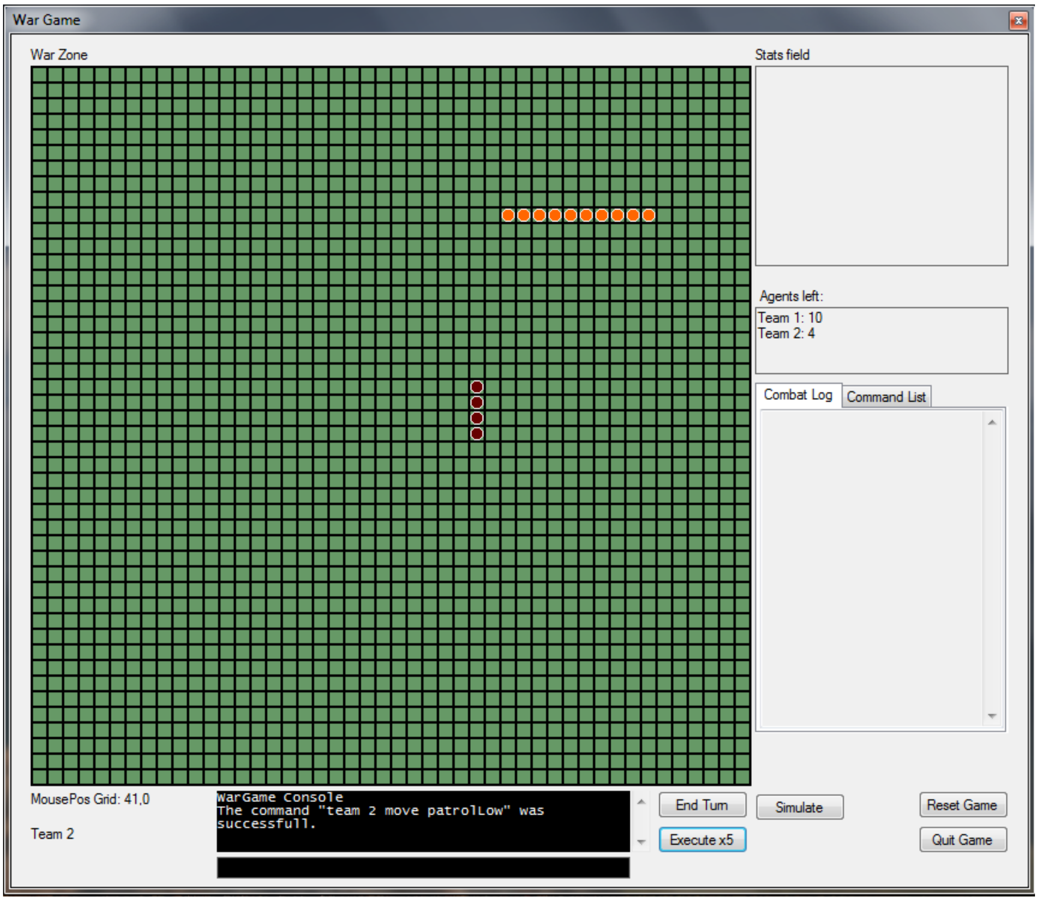


Figure 10.4: The result of the command "team 2 move patrolLow" followed by an "Execute x5".

10.4 MASSIVE

MASSIVE is a agent orientated language, which contains premade enviroment and functions for creating agents, squads, teams, and actionpatterns, which means that one does not have to build these functions themselves and it is therefore simpler to simulate a wargame in. The MASSIVE language does not support functions, therefore limiting the programmer to use the build in functions. As the MASSIVE language is case insensitive the programmer does not have to think about writing in upper or lower case characters.

Pros

- Simple to simulate a wargame.
- Premade enviroment.

- Premade types for agent, squad, team, and actionpattern.
- The language is case insensitive.

Cons

- Limited to use build-in functions.

10.5 C# vs MASSIVE

If the previous MASSIVE code example 10 was written in C# the differences would be how to delcare objects, use num instead of any number denoters, and that the language is case insensitive. Furthermore you cannot increment a number by using "++", but the programmer have to use an assignment instead. Therefore the most important difference between C# and the MASSIVE language is, that the MASSIVE language already have build-in types and functions needed by the programmer to initialize a wargame scenario.

Below is a comparison between some of the MASSIVE languages functions and the same functions written in C#. It is implicit that in the C# examples the types have been created already.

```
Main
{
new ActionPattern ap("FirstAction");
ap.add("unit move up");
ap.add("unit move left");
ap.add("unit move up");
}
```

Source code 10.2: MASSIVE ActionPattern code example

```
static void Main(string[] args)
    {
        ActionPattern AP = new ActionPattern("FirstAction");
        ap.add("unit move up");
        ap.add("unit move left");
        ap.add("unit move up");
    }
```

Source code 10.3: C# ActionPattern code example

A noticeable difference is that the programmer does not have to write the long definition to initialize the main method and ActionPattern is only written once to be declared.

```
Main
{
  new team teamDisco("Disco", "#FF6600");
  new team teamKman("Kman", "#660000");
}
```

Source code 10.4: MASSIVE Teams code example

```
static void Main(string[] args)
  {
      Team teamDisco = new Team("Disco", "#FF6600");
      Team teamKman = new Team("Kman", "#660000");
  }
```

Source code 10.5: C# Teams code example

Again the programmer dont have to write as much code, because the MASSIVE language does not require that the programmer is able to write object oriented types and objects.

CHAPTER 11

Conclusion

In this project a language called MASSIVE is developed. The purpose of MASSIVE is to control agents in a multi agent wargame. In order to implement this language, a compiler is also developed.

The language is limited to create agents, teams, squads, and actionpatterns for a wargame, because the purpose is to optimize the process of programming multi agent wargame scenarios. MASSIVE is simpler than for instance C#, since MASSIVE does not have the same amount of features.

MASSIVE comes with constructs for both agents, teams, squads and actionpatterns, allowing for new instances of these to easily be created. MASSIVE also comes with a few methods for easier manipulation of the data, making for more concise code, because the user does not have to define any custom constructs.

A second language has also been developed, designed only to control the agents in real time when running the wargame, which is implemented via an interpreter.

It is evident that MASSIVE is more optimized for programming multi agent wargame scenarios than C#. This is seen from the amount of code needed to prepare a wargame scenario in either language, as seen in section 10.

11.1 Future Work

11.1.1 Design Improvements

The design of a language is always difficult to improve because it is very subjective whether or not a language is well designed or not. However, a wider range of features could be implemented to provide the users with more advanced or easier to use features.

One of these features could be the addition of a for-each statement. This could for example be used to go through agents who are in a specific team or to go through the squads in a specific team.

Another feature which could be implemented is to allow the programmer to decide where the created team, squad or agent should start on the battlefield. This would provide better ability to simulate different battle scenarios.

To make the language more of a simulation and less of a game, it could also be implemented that the actionpatterns are not limited to the encounter and move functionality. These functionalities could be expanded by adding more states to the action interpreter.
A feature which enables the programmer to compare the current unit with units in its vicinity and act accordingly could also be implemented, this would give the agents a more independent and life-like behavior.

11.1.2 Implementation improvements

Besides improving the language design, the current implementation could also be improved. As previously described, the purpose of the compiler is to provide data that can be used in a wargame environment. Currently this is achieved by compiling the MASSIVE language into C# code, which then produces XML data. A more efficient way of doing it is by compiling straight to XML, so a separate file with C# does not have to be generated, compiled and run. This will also cause the MASSIVE compiler to be more efficient when it comes to memory usage, since it relies on the C# compiler, which is build for the more complicated language C# and therefor wont be optimized for the MASSIVE language, which for instance does not require the possibility to create new classes.

Currently, the compiler and wargame are two separate programs, however to make the user-experience more consistent the compiler and wargame can be integrated further by merging them into one program. This allows the compiler to skip the XML generation, and generate data directly to the wargame. By doing so it can provide the user the opportunity to create custom actionpattern and states contrary to being locked to the default ones.

Part V

Appendix

CHAPTER 12

Appendix

CHAPTER 13

Full Implemented Grammar

13.1 BNF - Initialize

Imperative:

type ::= *num* | *string* | *bool*
identifier ::= letter | identifier letter | identifier digit
letter ::= a | A | b | B | c | C | d | D | e | E | f | F | g | G | h | H | i | I | j | J | k | K | l | L | m | M | n | N | o | O | p | P | q | Q | r | R | s | S | t | T | u | U | v | V | w | W | x | X | z | Z
token ::= = | *num* | *string* | *bool* | ; | *new* | . | *Team* | *Agent* | *Squad* | *actionPattern* | *Coordinates* | (|) | , | | | *void* | *if* | *while* | *for* | *true* | *false* | *Main* | + | - | / | * | < | > | <= | >= | == | *else*

actual-string ::= "chars"
chars ::= char | char chars
char ::= Any unicode
boolean ::= *true* | *false*
number ::= digits | digits.digits
digits ::= digit | digit digits
digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
object ::= *Team* | *Agent* | *Coordinates* | *Squad*
operator ::= + | - | / | * | < | > | <= | >= | ==
becomes ::= =

variable ::= number | actual-string | boolean

mainblock ::= Main ( ) block
block ::= commands

commands ::= command ; | command ; commands
command ::= declaration | method-call | if-command | while-command | for-command | assign-command

assign-command ::= identifier becomes variable | identifier becomes expression

while-command ::= *while* (expression) block
if-command ::= *if* (expression) block | *if* (expression) block *else* block
for-command ::= *for* (type-declaration ; expression ; expression) block

expression ::= parent-expression | numeric-expression
parent-expression ::= (numeric-expression)
numeric-expression ::= primary-expression operator primary-expression | primary-expression operator-expression | parent-expression operator primary-expression | parent-expression operator expression
primary-expression ::= number | identifier | boolean

declaration ::= object-declaration | type-declaration
object-declaration ::= *new* object identifier (input)
type-declaration ::= type identifier becomes type

method-call ::= identifier (input) | identifier . method-call
input ::= variable | identifier | input, variable | input, identifier | ε

comment ::= // Any unicode eol | /* Any uni-code */

actionPattern-declaration ::= *actionPattern* identifier action-block
action-block ::= action
action ::= actual-string eol

13.2 Starters

starters[[letter]] ::= a | A | b | B | c | C | d | D | e | E | f | F | g | G | h | H | i | I | j | J | k | K | l | L | m | M | n | N | o | O | p | P | q | Q | r | R | s | S | t | T | u | U | v | V | w | W | x | X | z | Z
starters[[type]] ::= n | N | s | S | b | B
starters[[identifier]] ::= starters[[letter]]
starters[[token]] ::= starters[[type]]| ; | . | , | starters[[object]] | (|) | | | v | V | i | I | f | F | m | M | starters[[operator]]

starters[[string]] ::= "
starters[[chars]] ::= starters[[char]]
starters[[char]] ::= any unicode
starters[[bool]] ::= t | T | f | F
starters[[num]] ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
starters[[digit]] ::= starters[[num]]
starters[[digits]] ::= starters[[num]]

starters[[object]] ::= t | T | a | A | c | C | s | S

starters[[operator]] ::= + | - | / | * | < | > | =

starters[[object-declaration]] ::= n | N
starters[[type-declaration]] ::= starters[[type]]
starters[[actionPattern-declaration]] ::= a | A

starters[[input]] ::= starters[[letter]] | starters[[num]] | ε

starters[[method-call]] ::= starters[[letter]]

starters[[while-command]] ::= w | W
starters[[if-command]] ::= i | I
starters[[for-command]] ::= f | F

starters[[expression]] ::= starters[[primary-expression]]
starters[[primary-expression]] ::= starters[[letter]]
starters[[single-command]] ::= starters[[while-command]] | starters[[if-command]] | starters[[for-command]]
starters[[command]] ::= starters[[letter]] | starters[[block]] | starters[[num]]
starters[[commands]] ::= starters[[command]]
starters[[block]] ::=

starters[[mainblock]] ::= m | M

starters[[comment]] ::= /

13.3 EBNF - Initialize

type ::= *num* | *string* | *bool*
identifier ::= letter (letter | digit)* letter ::= a | A | b | B | c | C | d | D | e | E | f | F | g | G | h | H | i | I | j | J | k | K | l | L | m | M | n | N | o | O | p | P | q | Q | r | R | s | S | t | T | u | U | v | V | w | W | x | X | z | Z
token ::= = | *num* | *string* | *bool* | ; | *new* | . | *Team* | *Agent* | *Squad* | *actionPattern* | *Coordinates* | (|) | , | | | *void* | *if* | *while* | *for* | *true* | *false* | *Main* | + | - | / | * | < | > | <= | >= | == | *else*

actual-string ::= "chars"
chars ::= char (char)*
char ::= Any unicode
boolean ::= *true* | *false*
number ::= digits | digits.digits
digits ::= digit (digit)*
digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
object ::= *Team* | *Agent* | *Coordinates* | *Squad*
becomes ::= =
operator ::= + | - | / | * | < (=)+ | > (=)+ | = (=)+
variable ::= number | actual-string | boolean

mainblock ::= Main ( ) block
block ::= commands

commands ::= (command ;)*
command ::= declaration | method-call | if-command | while-command | for-command | assign-command

assign-command ::= identifier becomes (variable | expression)

while-command ::= *while* (expression) block
if-command ::= *if* (expression) block (*else* block)+
for-command ::= *for* (type-declaration ; expression ; expression) block

expression ::= parent-expression | numeric-expression

parent-expression ::= (numeric-expression)
numeric-expression ::= (primary-expression | parent-expression)+ operator (primary-expression | expression)+
primary-expression ::= number | identifier | boolean

declaration ::= object-declaration | type-declaration
object-declaration ::= *new* object identifier (input)
type-declaration ::= type identifier becomes (variable | expression)

method-call ::= (identifier .)* identifier (input)
input ::= (variable | identifier (, variable | , identifier)*)+

comment ::= // Any unicode eol | /* Any uni-code */

actionPattern-declaration ::= *actionPattern* identifier action-block
action-block ::= action
action ::= actual-string eol

13.4 Action Grammar

Declarative:

action ::= single-action EOL
selection ::= ID | identifier

ID ::= Agent ID | Squad ID | Team ID
Agent ID ::= num | *AGENT* num | *A* num
Squad ID ::= *SQUAD* num | *S* num
Team ID ::= *TEAM* num | *T* num

single-action ::= selection action-option move-option

action-option ::= *MOVE* | *ENCOUNTER*

move-option ::= *UP* | *DOWN* | *LEFT* | *RIGHT* | *HOLD* | coordinate | ActionPattern Name
coordinate ::= num , num

num ::= digits | digits.digits
digits ::= digit | digit digits

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

identifier ::= letter | identifier letter | identifier digit

token ::= *IDENTIFIER* | *MOVE* | *ENCOUNTER* | *HOLD* | *UP* | *DOWN* | *LEFT* | *RIGHT* | *EOL*