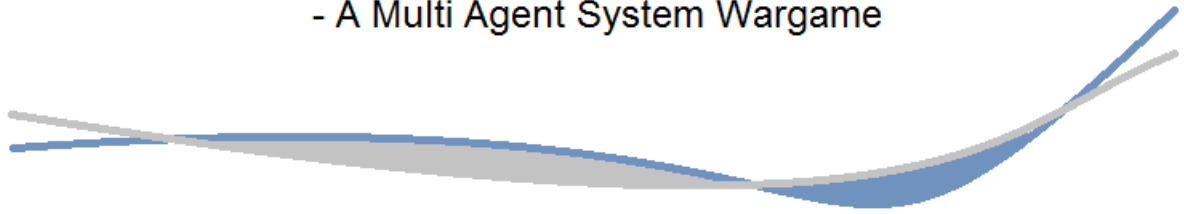# Compiler and Language Development

- 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

Number of copies:

Number of pages:

Number of appendices:

Completed: 27. May 2011

Synopsis:

In this project we develop an agent oriented language and a high-level to high-level compiler. Our language can control agents in a multi agent wargame.

The language we develop (called *MASSIVE*) is optimized for the wargame we develop. This becomes evident from a use case where we make a programming example in *MASSIVE*. Also a goal was to make the language more simple, for example by reducing the number of types.

—— conclusion ——

Preface

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

The goal of this project is to acquire knowledge about fundamental principles of programming languages and techniques for description and translation of languages in general. Also a 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 goal by designing and implementing a small language for controlling a multi agent system in the form of a wargame, which we call *MASSIVE* - **M**ulti **A**gent **S**ystem **S**imulation **I**n **V**irtual **E**nvironment. We are using Visual Studio and C#, because we have used these tools in earlier semesters and are used to the C# syntax.

The report is written i LaTeX, and we have used Google Docs and TortoiseSVN for revision control.

Source code examples in the report is represented as follows:

```
1  if (spelling.ToLower().Equals(spellings[i]))
2    {
3      this.kind = i;
4      break;
5    }
```

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 there we introduce the project, and gain some background knowledge. 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, an in this report we have focus on multi agent systems. In this project we are developing a language and compiler to generate code for a multi agent system. This leads to our problem statement:

How can a programming language and compiler, optimized to control logics of a multi agent wargame, be developed? How can one demonstrate this optimization?

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

In the next part, *Tools*, we describe building block of languages and compilers.

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

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

In the *Epilogue* part 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 entire group of agents, to achieve a predefined goal or condition.

In order to achieve this, a number of mechanisms are needed. First of all, need agents to be able to make decisions. This could be done randomly, however, for obvious reasons this would not produce very realistic results. 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 a higher state.[10]

One example could be a robot with a censor that feeds a binary input, 1 if it is warm and 0 if it is cold. If it is cold, the robot would be a the state "cold", which would have a lower value than the state "warm". If the robot then had the possibility to warm the room, it could decide to do this, in order to return to the state "warm", which is better because it has a higher value.

Another way to implement goals is be 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.[10], [4], [7]

1.1 Agent Oriented Languages

Creating a MAS using traditional programming language can be rather difficult and require both programming skills and time, In order to overcome this problem, people have started to develop languages specifically designed to create MASes and MAS-environments, these languages are called Agent Oriented Languages (AOL), and what they all have in common is added abstraction. Agent Oriented Languages ease the process of creating a MAS and therefore more people can create a MAS.

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

Chapter 2

Existing Environments

A lot of multi agent systems have already been developed, and we will take a look at a few of them, to get an idea of how others have designed multi agent systems.

2.1 Net-Logo

NetLogo is a popular and widespread environment for programming in a MAS language, developed by a man called Uri Wilensky in '99, who is currently working at the Northwestern University [9].

NetLogo features a very easy programming language for both creating Agents and defining Environments, and also provides relatively easy ways of manipulating the cosmetics of the MAS-simulation. NetLogo has the advantages that even though the programming language is very easy to learn and use, it is also very feature rich, and can create MAS's 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.

```
1  to setup
2    clear−all
3    ask patches
4      [ set pcolor (random colors) ∗ 10 + 5
5          if pcolor = 75   ;; 75 is too close to another color so
                change it to 125
6              [ set pcolor 125 ] ]
```

```
7    reset−ticks
8  end
9
10 to go
11   ask patches [ set pcolor [pcolor] of one−of patches ]
12   tick
13 end
14
15
16 ; Copyright 1997 Uri Wilensky. All rights reserved.
17 ; The full copyright notice is in the Information tab.
```

NetLogo Source code 2.1: This is a NetLogo sorce code example

This very small code example will, together with the NetLogo GUI, create
the elaborate simulation shown in 2.1. All of this simulation data is saved in
NetLogos custom file format, so that they can easily be run by someone else.
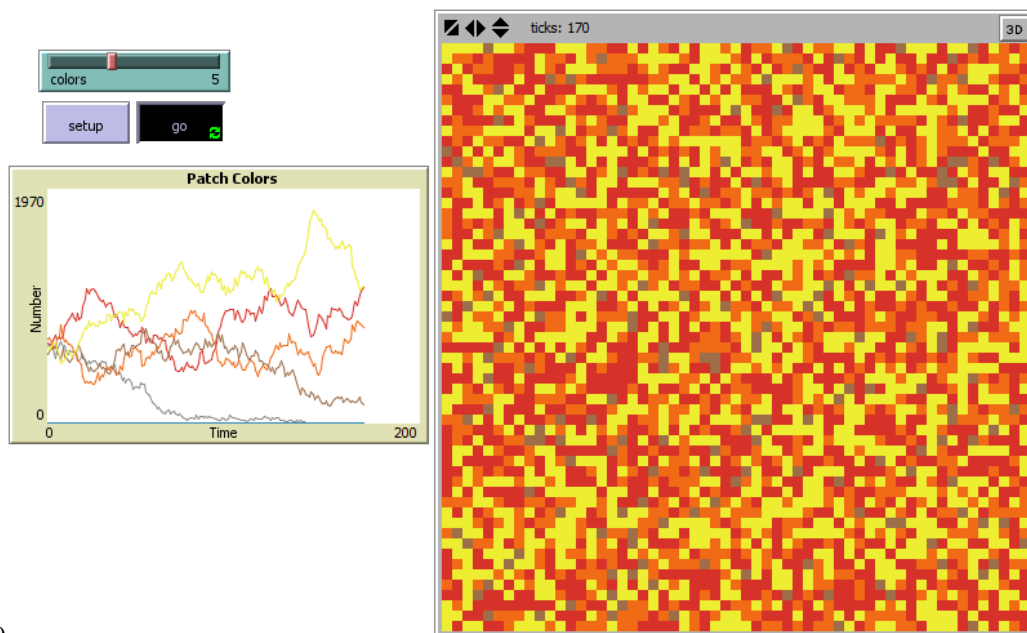
*0.9*0.9

Figure 2.1: Simple Netlogo Simulation

Chapter 3

Goal of the product

Our goal is to create a language that can express agents in some shape or form, we call this an Agent Oriented Language (AOL). Before that can be done however, it needs to be clarified what type of system we would like to build. We are not creating a MAS like the ones demonstrated2, but rather a form of wargame.

The programmer should, using our programming language, be able to express agents and predefine behaviors for these. The player of the game should then be able to choose whether to use the predefined behavior, or take control of the agent himself. Both the programmer and the player should be able to define the behavior of an agent when it come close to other hostile agents.

3.1 Rules

Our wargame scenario should be very simple;

- The game is turn-based

- The game is played on a grid-like map

- Each agent can move three grid-points in each turn

- A higher ranked agent have a higher chance of winning

- Agents fight when they are standing on the same grid-point

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

```
1  function gameRound()
2  {
3    gameFrame();
4    EndTurn();
5  }
```

Source code 3.1: Game Round

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

```
1  for(i = 0; i <= 3; i = i + 1)
2  {
3    //Check for encounters
4    foreach(agent a in agents)
5    {
6      if(a is within bounderies of another agent)
7      {
8        a.RemoveAllMovements();
9        a.encounter.Compile();
10     }
11   }
12
13   //Random movement if non
14   foreach(agent a in agents)
15   {
16     if(a has no movement)
17     {
18       agent moveToAgent = getRandomAgent();
19       a.MoveToAgent(moveToAgent);
20     }
21   }
22
23   //Check if the list is empty
24   if(moveAgents contains no items)
25     return;
26
27   //Update agent positions
28   foreach(agent a in agents)
29   {
30     if(a.team == currentteam)
31     {
32       foreach(agent moveAgent in moveAgents)
33       {
34         a.CalculateNextPosition();
35         if(a.NextPosition.IsInBounds())
36         {
37           a.MoveAgent();
38         }
39         if(a.IsAtEndPosition())
```

```
40            {
41                moveAgents.Remove(a);
42            }
43          }
44      }
45    }
46
47
48    //Check for collision
49    for(agentCount = 0; agentCount < agnets.TotalAgents;
         agentCount++)
50    {
51      foreach(agent a in agents)
52      {
53        if(a.CollideWithAgentOnOtherTeam())
54        {
55          if(a.Roll > CollidedAgent.Roll)
56          {
57            agents.Remove(CollidedAgent);
58          }
59          else
60          {
61            agents.Remove(a);
62          }
63        }
64      }
65    }
66 }
```

Source code 3.2: Game Frame

```
1  if(only team 1 has agents)
2  {
3    Team 1 wins!
4  }
5
6  ...
7
8  else if(only team n has agents)
9  {
10   Team n wins!
11 }
12
13 else
14 {
15   switchTurn();
16 }
```

Source code 3.3: 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[8]. 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 is based on reference [1]. Furthermore we describe the grammar and semantics of our language, MASSIVE, and how the language is used.

Chapter 4

Language Components

4.1 Grammar

In this project we use BNF and EBNF notation to describe our language, and those will be outlined in this section.

BNF (Backus-Naur Form) is a formal notation technique used to describe the grammar of a context-free language [5]. There are several variations of BNF, for example Augmented Backus-Naur Form (ABNF[1]) and Extended Backus-Naur Form (EBNF). EBNF is used to describe the grammer of the language developed in this project [1].

The EBNF is a mix of BNF and regular expressions (REs, see table 4.1), and thereby it 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.

---

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

	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	$X|Y$	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 [1]. X and Y are arbitrary REs, and t is any terminal symbol.

Here is 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, ...**

Left Factorization

Given that we have choises on the form $AB \mid AC$, where A, B and C are arbitrary extended REs, then we can replace these alternatives by 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 a 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 15.2.

4.2 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 [2].

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 $S \in$ 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.

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

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

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 15.

5.2 Semantics

The transition rules for the MASSIVE language are operational semantics written in bigstep notation. See section 4.2 for more theory on semantics.

Here we will be describing the transition rules for some of the transitions in MASSIVE. 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.

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

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

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:

$$\frac{env_v \vdash \langle S_2, sto \rangle \rightarrow sto'}{env_v \vdash \langle \texttt{if (b) } \{S_1\}\texttt{else } \{S_2\}, sto \rangle \rightarrow sto'}$$

(IF-ELSE-FALSE)

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

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.

(ADD-AGENT-SQUAD)

$$\frac{env_v \vdash \langle s, a, sto \rangle \rightarrow s', sto'}{env_v \vdash \langle \texttt{s.add(a) }, sto \rangle \rightarrow s', sto'}$$

This transitions uses an agent a and a squad s, and adds a to s, which leads to both s and sto being altered.

5.3 Usage of the MASSIVE Language

MASSIVE language is made for the specific purpose of making data for a wargame in the form of xml. To start using MASSIVE one need to learn some basics of the language; functions, loops, assigning values to variables, and statements. The first thing one needs to define when writing a program in MASSIVE is the main function. This is done by writing `Main()`. Then one can start writing the program inside the '' ''. There are 2 different loops in our language, the for-loop and the while-loop. The while-loop is written the following way:

```
1  while (/* Some expression */)
2  {
3      /* Some code */
4  }
```

Source code 5.1: While-loop

The for-loop can be written in the following way:

```
1  for (num i = 0; /* Some Expression */; i++)
2  {
```

```
3      /* Some code */
4   }
```

Source code 5.2: For-loop

Assigning values is an essential part of MASSIVE language, and can be done as long as the assigned value matched the datatype selected.

```
1   num count = 42;
```

Source code 5.3: Variable assignment

In MASSSIVE language we have some default classes one can use, these can be assigned using the following code:

```
1   new agent testAgent([name as string], [rank as num]);
2   new squad testSquad([name as string]);
3   new team testTeam([name as string], [color as hex code as string
        ]);
4
5   testSquad.Add(testAgent
6   testTeam.Add(testAgent);
```

Source code 5.4: Object assignment

There are 2 statements in MASSIVE language, the `if`-statement and the `else`-statement. The `else`-statement can only be used if it follows an `if`-statement:

```
1   num testNumber = 10;
2
3   if(testNumber = 20)
4   {
5       /* Some Code */
6   }
7
8   if(testNumber = 10)
9   {
10      /* Some code */
11  }
12  else
13  {
14      /* Some code */
15  }
```

Source code 5.5: Statements

When all the code has been written it can be run through the compiler, and it will generate an XML-file with the data entered.

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 descipe our semantics, which has a one to one mapping of the transistion.

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 first chapter we explain core concepts and ideas as to how to compile written code into executeable code. After that we outline our implementation of the compiler. Further more we descripe the graphical user interface to our MAS 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, or C# into Java.

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

6.1 Compilers

A compiler is basically a translator, typically capable of translating a language with a high level of abstraction (high level language), into a language that has a low level of abstraction (low-level language). 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, other then that there are no disadvantages to the approach.

A basic compiler can be broken down to three simple steps, which are illustrated in 6.1.
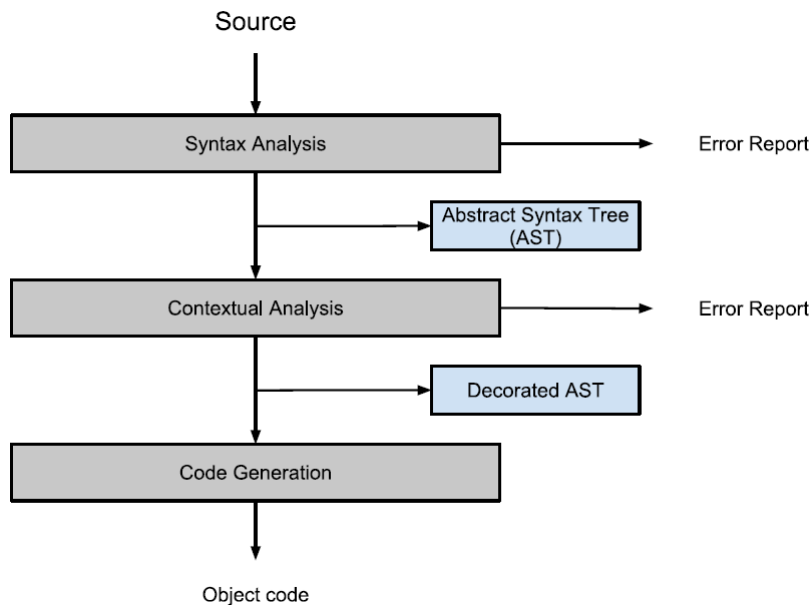
Figure 6.1: Illustration of the general structure of the compiler components.

6.2 Interpreters

An interpreter is also a translator, but where the compiler has to translate the entire input before the results can be used, 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.,

Because of this people would normally say that an Interpreter is best used when the program does not have to be run a great many times, or when the program in under development, and a compiler is best used when releasing large scale distributions of program.

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 OP-ERATOR 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 an equivalent token for [1]. 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 nonterminal symbols.

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

1. The lexical grammar is expressed in EBNF 4.1.

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 currect character to scan.
   (b) `take()`, which compares the current character to an expected character.
   (c) `takeIt()`, which updates the current character to the next character in the string.
   (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 4.1 and see the full implementation of the grammar in the appendix 15.

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 [1]. 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

---

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

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.

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 [1].
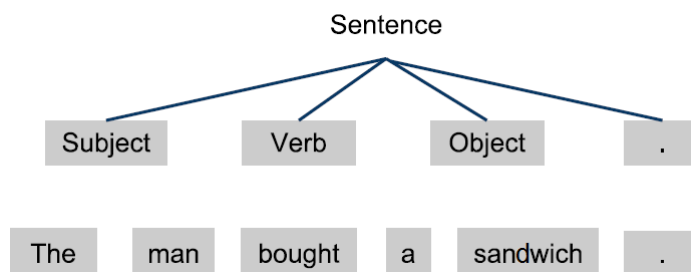
Figure 6.2: The first step for the parser is to decide what to apply ind the root node. Here it has only one option: "Sentence ::= Subject Verb Object."

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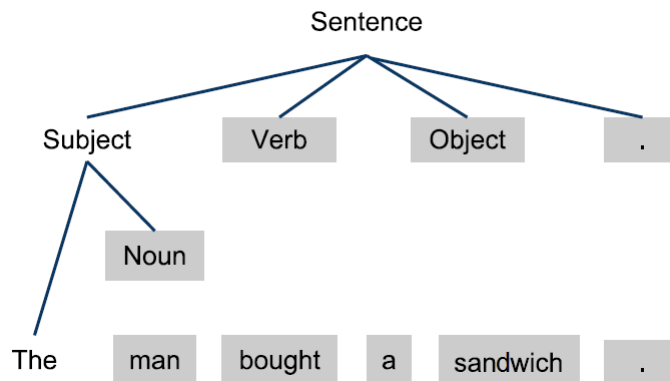
Figure 6.3: In the second step the parser looks at the stub to the left. Here the correct production rule is: "Subject ::= **The** noun".

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".
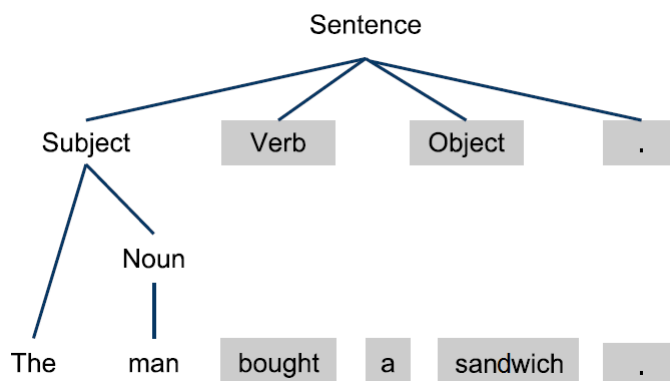
Figure 6.4: In third step the noun-stub is concidered, and the production rule becomes: "Noun ::= man".
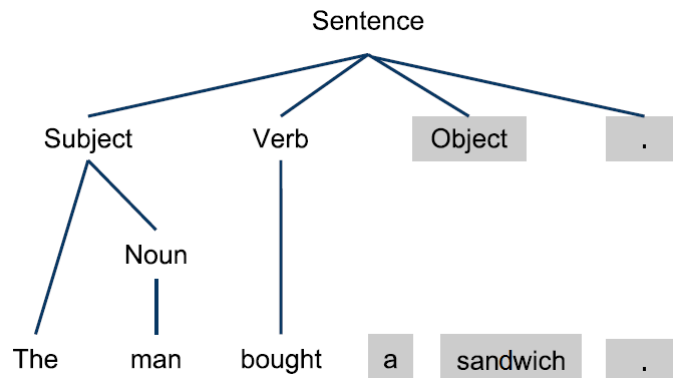
Figure 6.5: In fourth step the verb-stub is concidered, and the production rule becomes: "Verb ::= bought".
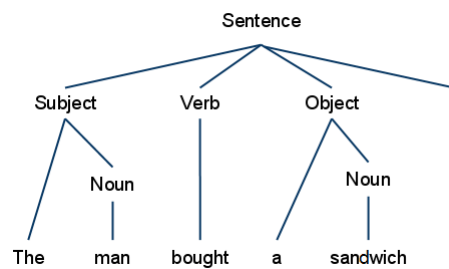
Figure 6.6: Here is the final syntax tree when the parser is done.

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 we have implemented the AST.

6.5 Decoration

Decoration refers to decorating the abstract syntax tree. Basically, up until this point we have only checked the structure of the code we are compiling, and decoration is the part where we do checks for validating the code it self. These are checks like type checks and scope checks.

To do this, we need a way of traversing the AST, while applying a lot of different logic to the various nodes in it. To this end, we utilize the visitor pattern.

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 advantageous because 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. We do not know what the type of each command is, we only know that there is a command object. 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, we will look at code from our own compiler. Say we are running through all the commands in a block.

```
1  foreach (Command c in block.commands)
2    {
3      c.visit(this, arg);
4    }
```

Source code 6.1: Here is the code that makes sure every command in a block is 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.

```
1  public class ForCommand : Command
2      {
3          ...
4          public override object visit(Visitor v, object arg)
5          {
6              return v.visitForCommand(this, arg);
7          }
8      }
```

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.

```
1  internal override object visitForCommand(ForCommand forCommand,
       object arg)
```

```
2        {
3            IdentificationTable.openScope();
4
5            // visit the declaration, the two expressions and the
                block.
6            forCommand.CounterDeclaration.visit(this, arg);
7            forCommand.LoopExpression.visit(this, arg);
8            forCommand.CounterExpression.visit(this, arg);
9
10           forCommand.ForBlock.visit(this, arg);
11
12           IdentificationTable.closeScope();
13
14           return null;
15       }
```

Source code 6.3: The visitForCommand function.

6.6 Code Generation

Code generation can be tricky, but because we are compiling to C#, we are utilizing the underlying memory management in C#, making the task much easier, and we won't expand on memory management for this reason. Code generation is therefor only a matter of printing the correct code.

A great tool for doing this is code templates. Code templates are recipes for what 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).

Chapter 7

Implementation of Compiler

7.1 Making the Scanner

The scanner is an algorithm that converts an input stream of text into a stream of tokens and keywords. The first method of the scanner is a big switch created to sort the current word according to the token starters (which can be found in appendix 15.2). E.g. if the first character of a word is a letter, the word is automaticly 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.

```
1  public Token(int kind, string spelling, int row, int col)
2          {
3              this.kind = kind;
4              this.spelling = spelling;
5              this.row = row;
6              this.col = col;
7
8              if (kind == (int)keywords.IDENTIFIER)
9              {
10                 for (int i = (int)keywords.IF_LOOP; i <= (int)
                       keywords.FALSE; i++)
11                 {
12                     if (spelling.ToLower().Equals(spellings[i]))
13                     {
14                         this.kind = i;
15                         break;
16                     }
17                 }
```

```
18                    }
19              }
```
Source code 7.1: The token method with overloads.

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

```
1   public static string[] spellings =
2           {
3               "<identifier>", "<number>", "<operator>", "<string>"
                    , ";", ":", "(", ")", "=", "{", "}",
4               "if", "else", "for", "while", "bool", "new", "main",
                     "team", "agent", "squad", "coord", "void",
5               "actionpattern", "num", "string", "true", "false", "
                    ,", ".", "<EOL>", "<EOT>", "<ERROR>"
6           };
```
Source code 7.2: The string array spellings.

This is the same for operators and digits, if the current word being read is an operator, the scanner builds the operator. If the operator is a boolean operator e.g. "<", ">", "<=", ">=", "==", 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 therefor contain both a single number og a number containing one punktuation.

Every time the "scan()" method is called, the scanner checks if there is anything which should not be implemented in the token list, e.g. 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 to make 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, that is called

whenever a corresponding token is checked. Each of these methods then generate their own subtree that is added to the AST.

```
1  public AST parse()
2          {
3                  return parseMainblock();
4          }
```

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

```
1  private AST parseMainblock()
2          {
3                  Mainblock main;
4
5                  accept(Token.keywords.MAIN);
6                  accept(Token.keywords.LPAREN);
7                  Input input = (Input)parseInput();
8                  accept(Token.keywords.RPAREN);
9                  main = new Mainblock(parseBlock());
10                 accept(Token.keywords.EOT);
11
12                 main.input = input;
13
14                 return main;
15         }
```

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 isn't one, the parser returns an error saying what line the error was on, and which token did not match an expected token.

7.3   The Abstract Syntax Tree

The Abstract Syntax Tree (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. This syntax tree will for eksample parse the source code:

```
1  Main  (  400  )
2  {
3     new  Team  teamAliens("Aliens",  "#FF0000");
4     new  Agent  agentAlice("Alice",  5);
5
6     teamAliens.add(agentAlice);
7  }
```

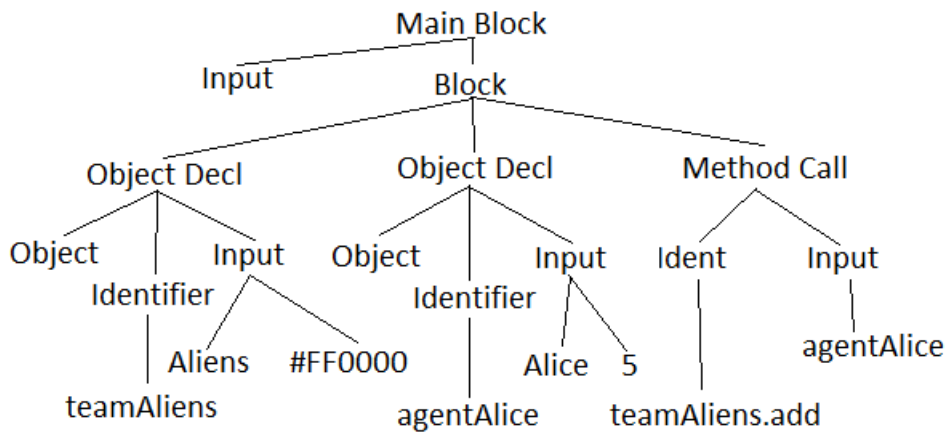Source code 7.5: Source code example.

To the AST:

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

The AST can be printed by a pretty printer to give a better overview of the compiled source code. In the MAS compiler, the pretty printer prints all completed parses in the windows console, the MAS pretty printer indents whenever a new branch is added. The source code above will be printed:
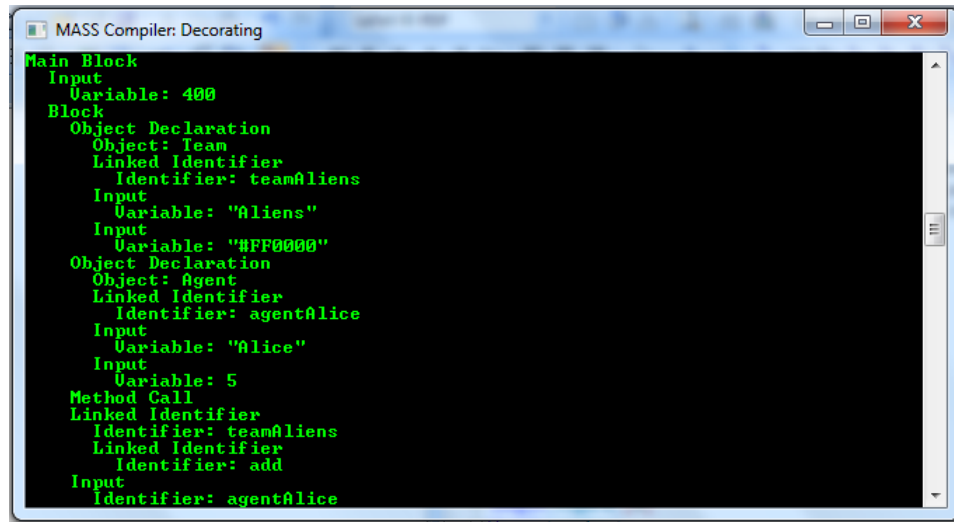
Figure 7.2: Example of the AST compiled with the MAS compiler.

7.4 Decoration

The decoration of the abstract syntax tree can be a large task, depending on how many checks that need to be done. To accomodate this task, 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.

Therefor, this is where type safety is enforced in the compiler. This works simply by taking the type of the variable's token and comparing it to the values it is being used with.

The scope checking is a little more complex. We want to make sure that variables are not used outside of 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 vari-

ables that are accessible from the current scope, so long as the scopes are updated correctly.

```
1  internal override object visitBlock(Block block, object arg)
2      {
3          IdentificationTable.openScope();
4          ...
5          IdentificationTable.closeScope();
6
7          return null;
8      }
```

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 constructions legal in our langauge:

```
1  new Team teamAliens("Aliens", "#FF0000");
2  new Team teamRocket("Team Rocket");
3
4  new Agent agentJohn("John", 5, teamAliens);
5  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 the validity of given input, from these classes. It's therefor easy to add new overloads to existing methods and constructors, as well as completely 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 simply to check that 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 of agents, squads and teams that go unused in the code.

7.5   Code Generation

In order to print the C# code, we must again traverse the AST, and determine what code should be printed. Therefor, as with the decoration process, we use a visitor (see section 6.5.1) 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, we use code templates.

A code template is basically a recipe for how the input code should be converted into C# code. Many of our templates are printed as the code is visited by the visitor. For example will a for command first have "for (" printed, followed by a type declaration, "num i = 0;", an expression, "i < 10;" and finally an assignment command, "i = i + 1" with a parenthesis to round off.

For the methods in our language, we have a different solution though. Every class for a method or constructor, see 7.4.2, in our language must define an overload for the method PrintGeneratedCode.

```
1   public  override  string  PrintGeneratedCode(string  one ,  string  two
        )
2               {
3                   // squad  one  =  new  squad(two)
4                   return  "squad  "  +  one.ToLower()  +  "  =  new  squad("  +
                        two.ToLower()  +  ")";
5               }
```

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

In the code for the squad constructor, two strings are given as input. One is the variable name, and two is the input given as a string. It is therefor pretty straightforward to input the strings in their proper place. A more complex example is the agent constructor, which takes both a name, a rank and a team as input.

```
1   public  override  string  PrintGeneratedCode(string  one ,  string  two
        )
2               {
3                   string[]  input  =  two.Split(',');
4
5                   // agent  one  =  new  agent(two);
6                   // one.team  =  two
7                   return  "agent  "  +  one.ToLower().Trim()  +
8                     "  =  new  agent("  +  input[0].ToLower().Trim()  +
9                     ",  "  +  input[1].ToLower().Trim()  +  ");\n"  +
10                   one.ToLower().Trim()  +  ".team  =  "  +
11                   input[2].ToLower().Trim();
12              }
```

Source code 7.9: The code printed for the agent constructor taking three arguments as input.
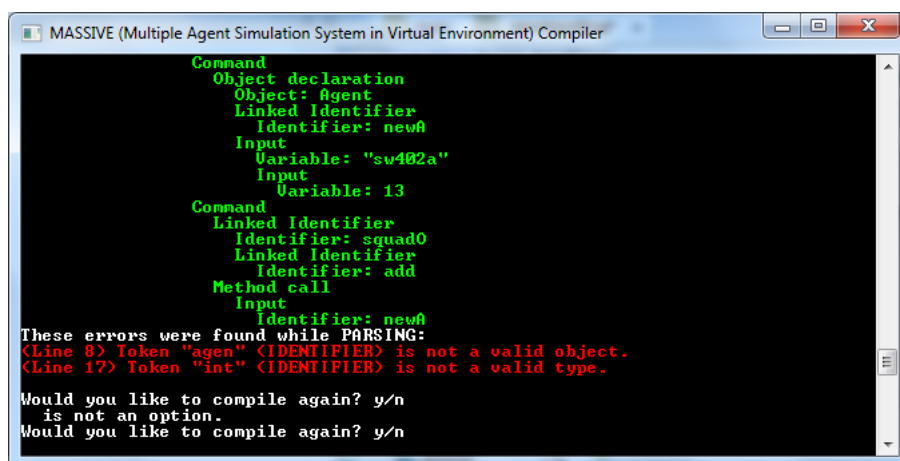
Here the input string must be split up and put in the correct places, but the method still takes the same arguments as the other overloads. These templates are what makes it possible to easily print the code for any method or constructor used in our language.

7.6 Error handling

It is very important that a programmer knows if the code he is writing is correct or not, so it's very important that the compiler tells him of any errors it encounters. Our compiler can catch errors after every parsing of the code it does, and it will also complete the parse, so it can report every error encountered in that parse.

To make the compiler easier to use, we have tried to make the error messages very descriptive. The programmer also gets a choice of whether he wants to print the compilation of the code, and error markers will be printed in the correct places if he does. 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 **??**). 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

The user interface is made as a windows form application[1]. Using Visual Studios designer tools, it is simple to make a graphical user interface with buttons, panels, and windows just the way you want.

The main idea of the design of the user interface is that it should be intuitive, which the user should not spend a lot of time figuring out what all the buttons do. Furthermore we have designed the interface so the main structure looks like other popular strategy computer games (see 14.1 and 14.2 in appendix). We have done this to make the application easy to learn how to use.

Game Start Settings

When the game is started, a dialog box is shown where one can choose the size of the *war zone*. We have chosen to have three fixed grid sizes, because of the way we draw the grid 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.
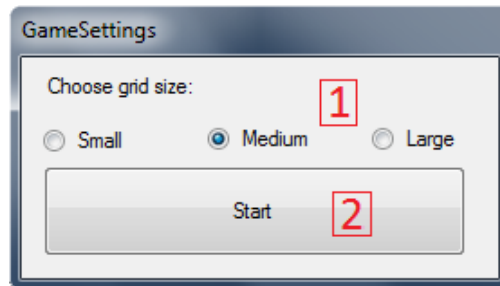
Figure 8.1: Screen shot of the game settings dialog box.

Game Interface Functions

The functions of the game interface is:

1. *War zone* - contains the grid on which the war game unfolds.

2. *Agents* - the agents of the different teams (here with a 4-player game setup).

3. *Command center* - here the user types the commands to navigate the agents around the grid.

4. *Stats field* - shows the stats of a selected agent.

5. *Agents left* - shows how many agents are left on the teams.

6. *Combat log* - contains a combat log on who killed who in fights between agents.

7. *Command list* - contains the list of available commands the user can type in the *command center*.

8. *MousePos grid* - shows the grid point of the mouse position.

9. *Execute* button - executes the typed in command in the *command center*.

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.
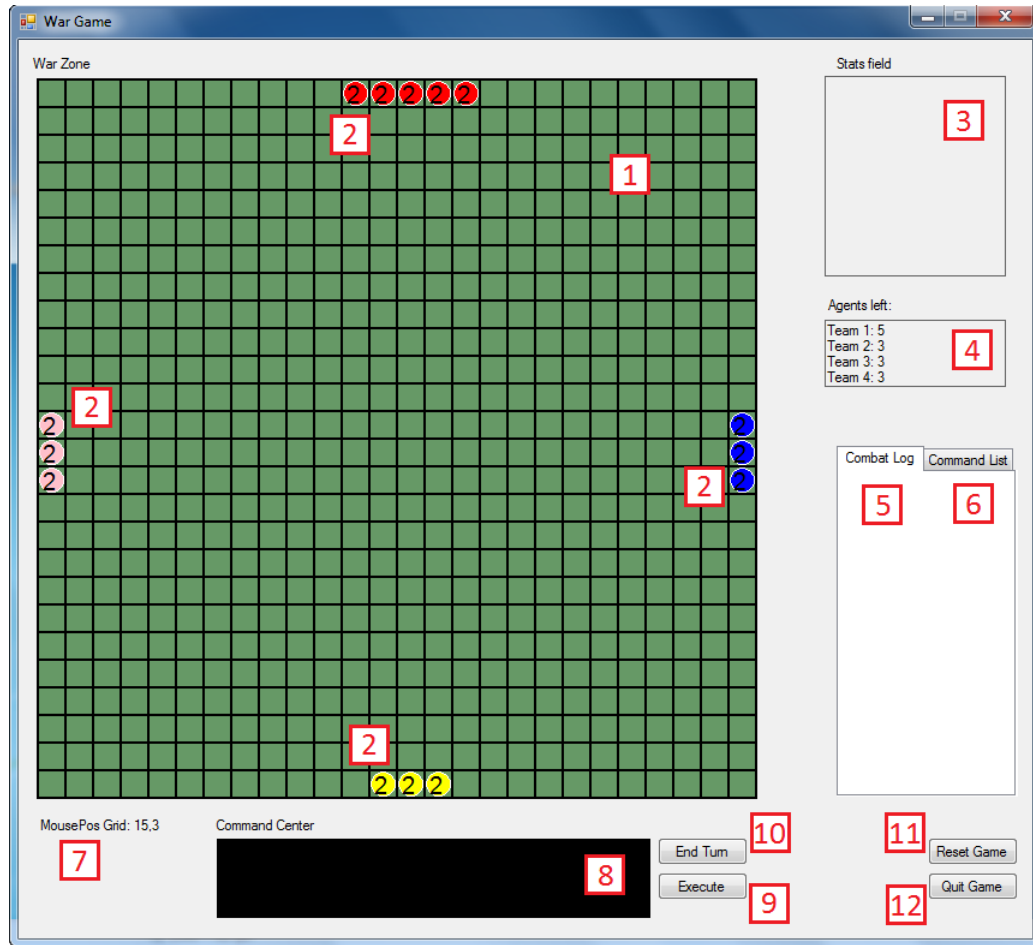
Figure 8.2: Screen shot of the game interface.

Drawing the Grid and Agents

The program make use of GDI+ [3] to draw the grid (the war zone) on the screen. A usercontrol is added to eliminate the flickering GDI+ normally creates on windows forms, this is done with the help of double buffering. We only use GDI+ graphics inside the usercontrol DBpanel, and make sure we draw things in the correct order, as we draw the pixels untop of each other. The first thing drawn is the background, which in our case is green, with the black gridlines on top of it, to create the game grid. Next the agents are drawn, one after the another. The agent's start posistions are calculated by the following algorithm:

```
int it1 = (Grids / 2) − (agentsOnTeam1 / 2);
```

44

```
2                int it2 = (Grids / 2) − (agentsOnTeam2 / 2);
3                int it3 = (Grids / 2) − (agentsOnTeam3 / 2);
4                int it4 = (Grids / 2) − (agentsOnTeam4 / 2);
5                foreach (Agent a in agents)
6                {
7                    Point p = new Point();
8                    if (a.team.ID == 1)
9                    {
10                        p = getGridPixelFromGrid(new Point(it1, 0));
11                    }
12                    else if (a.team.ID == 2)
13                    {
14                        p = getGridPixelFromGrid(new Point(Grids −
                             1, it2));
15                    }
16                    else if (a.team.ID == 3)
17                    {
18                        p = getGridPixelFromGrid(new Point(it3,
                             Grids − 1));
19                    }
20                    else if (a.team.ID == 4)
21                    {
22                        p = getGridPixelFromGrid(new Point(0, it4));
23                    }
24
25                    a.posX += p.X;
26                    a.posY += p.Y;
27
28                    if (a.team.ID == 1)
29                    {
30                        it1++;
31                    }
32                    else if (a.team.ID == 2)
33                    {
34                        it2++;
35                    }
36                    else if (a.team.ID == 3)
37                    {
38                        it3++;
39                    }
40                    else if (a.team.ID == 4)
41                    {
42                        it4++;
43                    }
44                }
```

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

It is the start location for each team. If the grid is 13 "grids" wide and team one consists 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, 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 wants to operate on.
The state, indicates which state the unit should execute the command, e.g. the encounter command waits untill there is an enemy unit in its parameter.
The option, identifies the coordinate or direction the unit should go to, e.g. the option up would move the unit one square up.

Some of the most simple commands in the action interpreter would be the "move" commands, e.g. "12 move 1,2" would move the unit with the 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 its in range of an enemy unit.

8.1.1 Contextual Analysis & Code Generation

The contextual analysis is the decoration of the AST, which is done by traversing the AST with the visitors. In the contextual analysis, code generation is build in, since there is no need to parse the AST more than once, when all information used by the move function 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").

```
1  if (object.ReferenceEquals(
2    single_Action.selection.GetType(),
3    new squadID().GetType()
4    ))
5    {
6    // set arg to null if its an id.
7    visitCodeGen_MoveSquad(single_Action, null);
8    }
```

Source code 8.2: Example of the determination of the identifier in the visitors, this part identifies SquadID.

When the squad has been identified the decorator calls the visitCode-Gen_MoveSquad method and moves all agents in the squad.

```
1  squad squad;
2  // If arg is null, the selection is an ID.
3  if (arg == null)
4    {
5      squadID select = (squadID)single_Action.selection;
6      Token selectToken = select.num;
7      squad = Lists.RetrieveSquad(Convert.ToInt32(selectToken.
          spelling));
8    }
9  else
10      {
11      Identifier ident = (Identifier)single_Action.selection;
12      squad = Lists.RetrieveSquad(ident.name.spelling);
13    }
14
15    foreach (agent a in squad.Agents)
16    {
17      visitCodeGen_MoveOption(a, single_Action.move_option);
18    }
```

Source code 8.3: Snippit of the identification of the units in a squad.

The visitCodeGen_MoveOption method, analyse the stance and the option. If the stance is encounter instead of move, a string with the agent, the agents name, the stance move and the option.

```
1  // If the stance is an encounter call the add encounter function
      .
2  if (move_Option.stance == (int)Stance.Stances.ENCOUNTER)
3    {
4      Functions.addEncounter(_agent, _agent.name + " move " +
          token.spelling);
```

```
5        return ;
6    }
```

Source code 8.4: Code snippit, when the encounter stance is chosen instead of move.

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

```
1  object moveOption = move_Option.dir_coord.visit(this, null);
2
3  // If there was no actionpattern with this name, Exception.
4  if (moveOption == null || !object.ReferenceEquals(moveOption.
      GetType(), new actionpattern().GetType()))
5    {
6        throw new InvalidMoveOptionException("The actionpattern was
            invalid!");
7    }
8  actionpattern ap = (actionpattern)moveOption;
9
10  // If the stance is an encounter call the add encounter function
       .
11  if (move_Option.stance == (int)Stance.Stances.ENCOUNTER)
12    {
13        Functions.addEncounter(_agent, _agent.name + " move " + ap.
            name);
14        return ;
15    }
16
17  foreach (string s in ap.actions)
18    {
19        ActionInterpet.Compile(s, _agent);
20    }
21  return ;
```

Source code 8.5: The method moving a unit if, the move option is an action pattern.

Compilers and interpreters are two types of translators, where a compiler has to translate the entire input before the result can be used. An interpreter

runs one instruction at a time from the input, thus enabling it to start utilizing the input when it is received.

The scanner produce a stream of tokens which it has recognized in the source program. The parser then recognize the phrase structure of the token stream.

We have implemented...

Part IV

Discussion

In this part we discuss our project. We describe a use case of the MAS-SIVE language and from that we show how our language has lived up to its purpose. Also we list some advantages and disadvantages of the MASSIVE language versus other object oriented programming languages, i.e. C#, and finally we conclude on the project as a whole.

Chapter 9

Language Development

9.1 Compiler language

We decided early on to develop our compiler in C#, because it is a language we have a lot of experience with, and the object oriented paradigm is helpful in developing a compiler that uses an abstract syntax tree. 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 [6], where new data is created and the alterations are applied to, so reference types are of no concern.

Chapter 10

MASSIVE Language

10.1 OOP

10.2 MASSIVE

10.3 C# vs MASSIVE

Source code 10.1: C# ActionPattern code example

```
1   Main(400)
2   {
3   new ActionPattern ap("Action1");
4   ap.add("unit move up");
5   ap.add"unit move left");
6   ap.add("unit move up");
7   }
```

Source code 10.2: MASSIVE ActionPattern code example

Source code 10.3: C# Teams code example

```
1   Main(
```

Source code 10.4: MASSIVE Teams code example

Source code 10.5: C# Agent code example

53

```
1   Main(400)
2   {
3   new Agent derp(
4   }
```

Source code 10.6: MASSIVE Agent code example

Source code 10.7: C# Squad code example

Source code 10.8: MASSIVE Squad code example

10.4   Use Case

Chapter 11

Conclusion

tail...

Part V

Epilogue

In this part we come with suggestions on how the language, compiler and environment could be further improved in future work.

Chapter 12

Future Work

The purpose of the compiler is to provide data that can be used in our wargame. Currently this is achieved by compiling our language into C# code, which then produces XML data when run. A more efficient way of doing it could be by compiling straight to XML, so a seperate file with C# would not have to be generated, compiled and run.

Currently, our compiler and wargame are also seperate, and the two could be integrated further by building them into one program, making for a more consistent experience. This would allow us to skip XML generation, and generate data directly to the wargame.

Other improvements could be made to the language it self. For example are action patterns very limited in functionality right now, and introducing language constructs that would allow for conditional movements could make a big difference. Allowing users of the language to define their own encounters, like what would happen if an agent met an agent with three times as much rank, would also be a big improvement.

tail...

Part VI

Appendix

Chapter 13

Appendix

Chapter 14

Other Games

Figure 14.1: Screen shot of the game user interface in Red Alert 2.

Figure 14.2: Screen shot of the game user interface in Command and Conquer 3.

Chapter 15

Full Implemented Grammar

15.1   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

15.2 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]] ::= /

15.3 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

15.4   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

Bibliography

[1] Deryck F. Brown David A. Watt. Programming language processors in java. Book, 2000.

[2] Hans Hüttel. Transitions and trees. Book, 2010.

[3] Microsoft. Gdi+. Website, 2010. Date seen: 25. mar. 11.

[4] Jürgen Dix & Amal El fallah Seghrouchni Rafael H Bordini, Mehdi Dastani. Multi-agent programming. PDF, 2009. Chapter 1 & 2.

[5] Michael Sipser. Introduction to the theory of cumputation - second edition, international edition. Book, 2006. Chapter 2.

[6] Unknown. Why haskell matters. Website, 2011. http://www.haskell.org/haskellwiki/Why_Haskell_matters - Date seen: 15. may 11.

[7] José M Vidal. Fundamentals of multiagent systems. PDF, 2010. Chapter 1.

[8] Uri Wilensky. Netlogo. Website, 1999-2011. URL: http://ccl.northwestern.edu/netlogo/ - Date seen: 02. mar. 11.

[9] Uri Wilensky. Netlogo. Website, 2011. http://ccl.northwestern.edu/uri/ - Date seen: 13. may 11.

[10] Kevin Leyton-Brown Yoav Shoham. Multiagent systems. PDF, 2009, 2010. Chapter 1.