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**imulation**S**ystem **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:

```
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 a programming language and compiler, optimized to control agents of a multi agent wargame, be developed?

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.[10]

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.[10], [3], [6]

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 system, we will take a look on NetLogo.

2.1 NetLogo

NetLogo is a widespread environment for programming a MAS. NetLogo developed by Uri Wilensky in 1999, at the Northwestern University [9].

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. [7]

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.

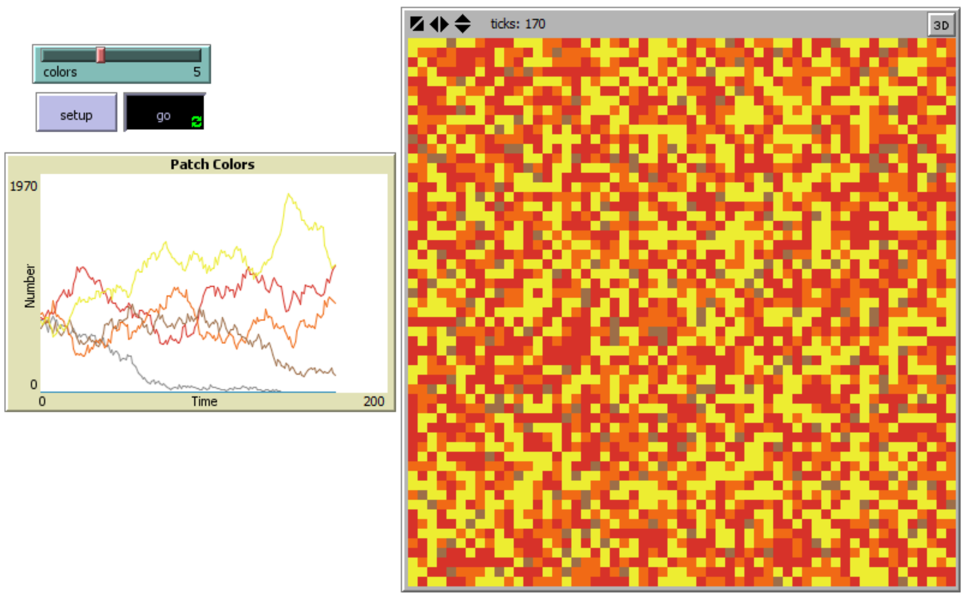


Figure 2.1: Simple Netlogo Simulation

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 game is played on a grid.
- 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.

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.

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 3.4: 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 3.5: 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 3.6: 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 [4]. 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 12.2.

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 $n \in \text{Num}$.
- Variables $v \in \text{Var}$.
- Arithmetic expressions $a \in \text{Aexp}$.
- Boolean expressions $b \in \text{Bexp}$.
- 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.

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

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

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 | digits.digits.

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

digits ::= digit | 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 12.

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.

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

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

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:

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

if $env_v, sto \vdash b \rightarrow ff$

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:

```
while(/* Some expression */)
{
    /* Some code */
}
```

Source code 5.1: While-loop

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

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

```
{
    /* Some code */
}
```

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.

```
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:

```
new agent testAgent([name as string], [rank as num]);
new squad testSquad([name as string]);
new team testTeam([name as string], [color as hex code as string
    ]);

testSquad.Add(testAgent
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:

```
num testNumber = 10;

if(testNumber = 20)
{
    /* Some Code */
}

if(testNumber = 10)
{
    /* Some code */
}
else
{
    /* Some code */
}
```

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 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

Discussion

In this part we discuss our project. We descripe a use case of the MASSIVE 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 6

Language Development

6.1 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 [5], where new data is created and the alterations are applied to, so reference types are of no concern.

CHAPTER 7

MASSIVE Language

In this report we illustrates how we designed and implemented the agent oriented language, MASSIVE. During this chapter we demonstrates 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.

7.1 Use Case

In this use case we demonstrate how to write a mini-game in our language, how to compile it and how to play it.

The first thing one needs to do is to write some MASSIVE code. In 71 is examples of code, however, there are features of the langauge that are not being used in this example. For a full code referernce please check **??** HEJ MED DIG. In the example two teams are created called "Disco" and "Kman", agents are added to them and at the end a simple action pattern is defined, later to be used when running the simulation.

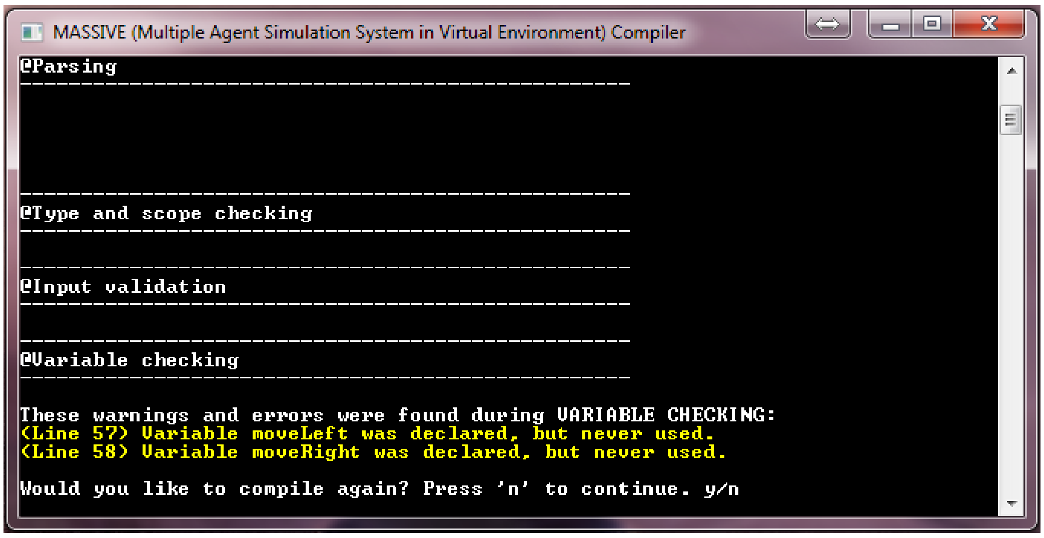
```

/* Initializes the game with the properties
Maximum Units = 400 */
Main ( 400 )
{

  // 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 7.1: MASSIVE code example

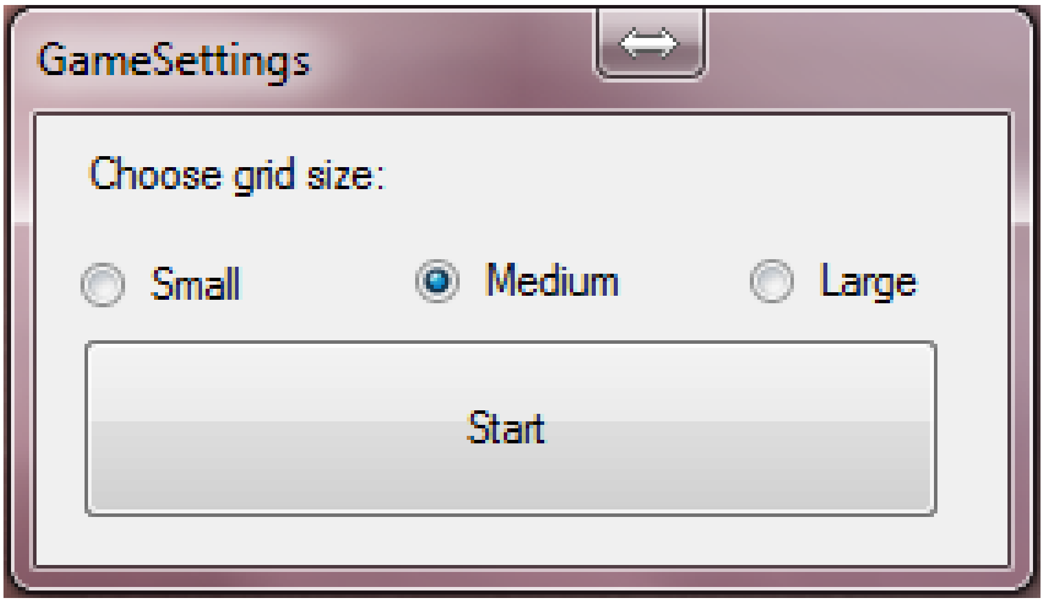
When compiling this code the compiler warns that there are unused variable (see 7.1). We will disregard this for the purpose of this use case, however, if there were serious faults in the code the compiler would warn you the same manner and maybe even refuse to compile if the faults were serious enough.


MASSIVE (Multiple Agent Simulation System in Virtual Environment) Compiler

```
@Parsing
--------------------------------------------------




--------------------------------------------------
@Type and scope checking
--------------------------------------------------

--------------------------------------------------
@Input validation
--------------------------------------------------

--------------------------------------------------
@Variable checking
--------------------------------------------------

These warnings and errors were found during VARIABLE CHECKING:
(Line 57) Variable moveLeft was declared, but never used.
(Line 58) Variable moveRight was declared, but never used.

Would you like to compile again? Press 'n' to continue. y/n
```

Figure 7.1: The MASSIVE compiler warning of unused variables.

The compiler will happily compile the code again if that option is selected, which provides the programmer with an easy way of correcting erroneous code. After a succesfull compilation a file named "MASSIVECode.cs"and "MASSIVECode.exe" will have been created. The only purpose of creating the cs-file is allowing the programmer to have a look at the code our compiler generates. The cs-file will have been compiled into the exe-file wich is run automatically. This exe-file creates the actual data output in XML format, which is then run by the MASSIVE simulator, and the user of the simulator is given a choice of how large the game grid will be (see 7.2).


GameSettings

Choose grid size:

Small [✓ Medium] Large

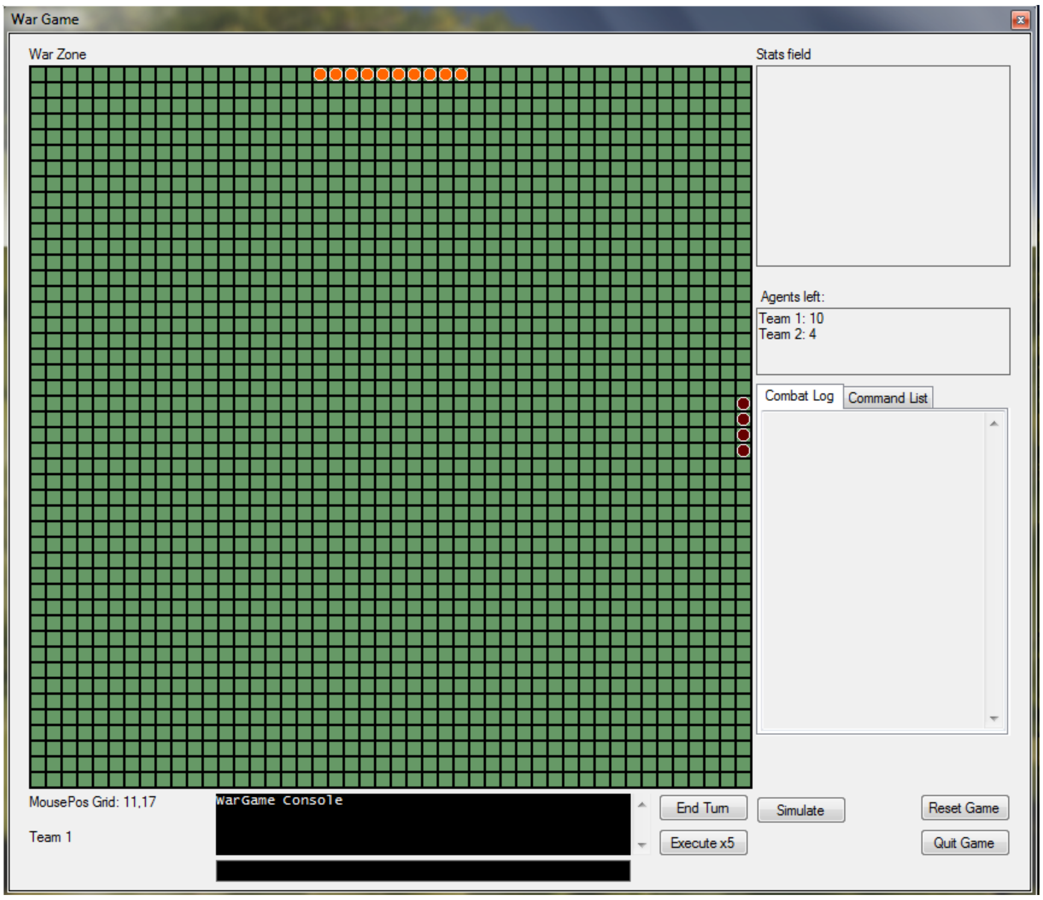
Figure 7.2: Choosing the size of the game-grid

Upon choosing "large", the user will bee presented with the actual simulation (see 7.3). Here he will have the oppertunity to instruct the agents to use the action pattern defined in 71, as shown in 7.3.

At this point the user is presented with a choice; He can either press "Simulate" to let the simulation run to an end without any interaction, or he can choose to run the game turn-by-turn and control the agents as the game progresses. We see the result of this simulation in 7.4.

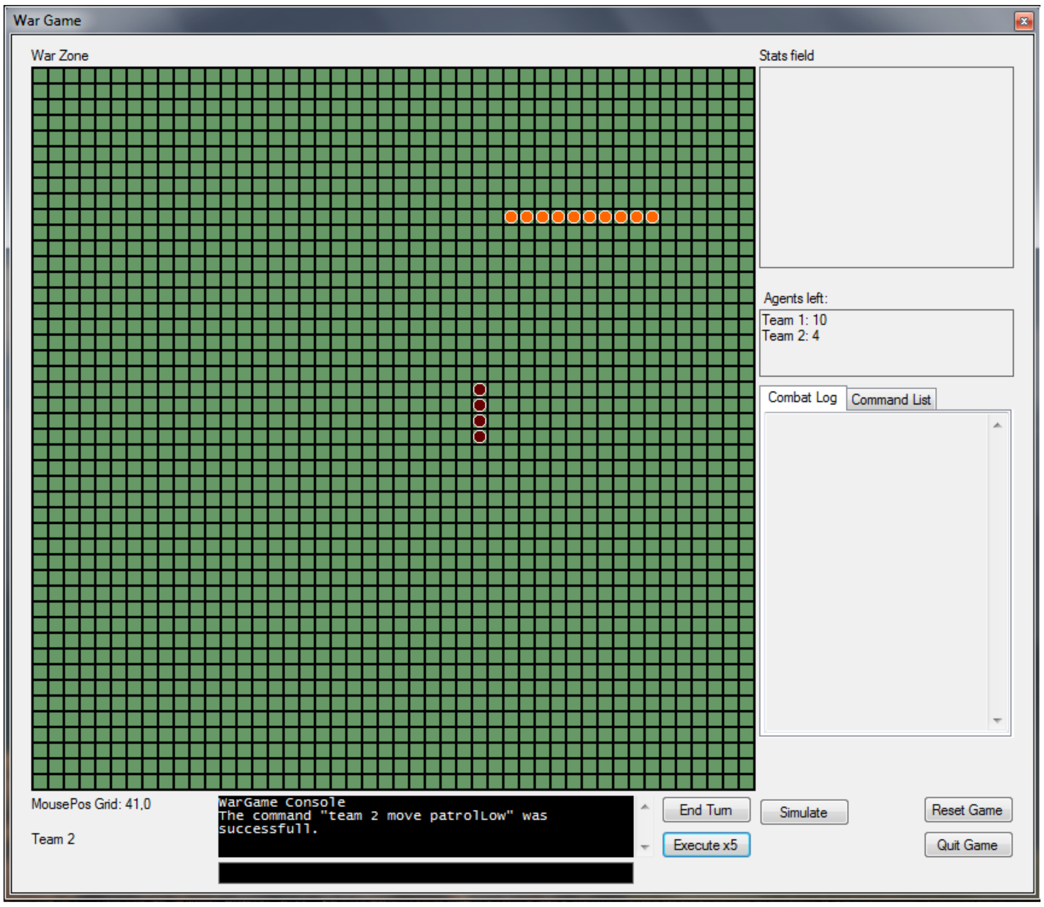
7.2 Comparison

This section is about how to make a multi agent wargame in C# compared to our own language MASSIVE. We will take a look on some of the pros and cons by using C# to build a multi agent wargame, aswell as the pros and cons while using MASSIVE. We will then compare C# and MASSIVE to examine which language is the best to build a multi agent wargame.



Team 1: 10
Team 2: 4

Figure 7.3: The simulation running with the input instructing som of the agents to use an actionpattern



```
Team 1: 10
Team 2: 4
```

```
WarGame Console
The command "team 2 move patrolLow" was
successfull.
```

Figure 7.4: The result of the use case simulation in MASSIVE

7.3 C#

C# is a object orientated language(OOP) that we have decided to compare MASSIVE with, we decided to use C# to compare with because both our compile and enviroment are written in this language. C# do not have built in functions or enviroment multi agent orientated, which means it is required for the programmer to build the multi agent wargame from scratch. To build a basic multi agent wargame in C# you need to make constructors for agent and teams, furthermore you will also need to create functions for agents and teams, which might be movement, attack, and attack functions. At last you will need to create an enviroment where the agents and teams will simulate a wargame. However building a multi agent wargame in C# enables you to create all the features you want in a wargame simulation.

Using C# you can declare objects and use them later.
Pros

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

Cons

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

7.4 MASSIVE

MASSIVE is a agent orientated language(AOL) which contains premade enviroment and functions for creating agents, squads, teams, and actionpatterns, which means that you do not have to make this yourself. It is relative fast to simulate a wargame in MASSIVE, since all the function you need is already made. You cannot declare new functions in MASSIVE which limit you to only use the built in functions. MASSIVE is not case sensitve. When declaring an object in MASSIVE you are forced to declare it with properties.

Pros

- Relative fast to simulate a wargame.
- Premade enviroment.
- Premade constructors for agent, squad, team, actionpattern.

- Type and functions are not case sensitive.

Cons

- Limited to the languages functions.

7.5 C# vs MASSIVE

In this section we will compare C# and MASSIVE, we assume that we have already created constuctors, functions, and an enviroment for the C# code.

```
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 7.2: C# ActionPattern code example

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

Source code 7.3: MASSIVE ActionPattern code example

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

Source code 7.4: C# Teams code example

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

Source code 7.5: MASSIVE Teams code example

```
static void Main(string[] args)
  {
    for(int i = 0; i < 4; i++)
    {
      if(i < 2)
      {
        Agent newAgent = new Agent("Stue", i+1);
        teamDisco.add(newAgent);
      }
      else
      {
        Agent newAgent = new Agent("Kman", i-1);
        teamKman.add(newAgent);
      }
    }
  }
```

Source code 7.6: C# Agent code example

```
Main
{
  for(num i = 0; i < 4; i = i + 1)
  {
    if(i < 2)
    {
      new Agent newAgent("Stue", i+1);
      teamDisco.add(newAgent);
    }
    else
    {
      new Agent newAgent("Kman", i-1);
      teamKman.add(newAgent);
    }
  }
}
```

Source code 7.7: MASSIVE Agent code example

```
static void Main(string[] args)
  {
    Squad squadNabs = new Squad("noobs");
    Squad squadRevo = new Squad("Revolution");
    squadNabs.add(newAgent);
    squadRevo.add(anotherAgent);
  }
```

Source code 7.8: C# Squad code example

```
Main
{
  new Squad squadNabs("noobs");
  new squAd squadRevo("Revolution");
  squadNabs.Add(newAgent);
  squadRevo.add(anotherAgent);
}
```

Source code 7.9: MASSIVE Squad code example

In the above code examples you can see how one would generate teams, agents, squads, and actionspattern using C# and MASSIVE. The structure of the two languages are every much alike, the only visible differences are how to declare objects, num instead of int, type and functions are not case sensitive, and that you cannot increment by using "++". The importan difference cannot be seen in the code examples above, because the code examples only show how you call functions and perform for loops, but it can been seen in the time ... The importan difference between C# and MASSIVE is that you have to create your own environment, constructors and functions when using C# which takes a long time compared to MASSIVE where you quickly can make a wargame simulation since all the needed functions is already made for you.

CHAPTER 8

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 creating agents, teams, squads, and actionpatterns for a wargame, because the purpose is to optimize the process of programming multi agent wargame scenarios. MASSIVE is easier to start using than for instance C#, since MASSIVE does not have the same amount of features, and is therefore easier to get an overview of.

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 7.

tail...

Part IV

Epilogue

CHAPTER 9

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.

Part V

Appendix

CHAPTER 10

Appendix

CHAPTER 11

Other Games

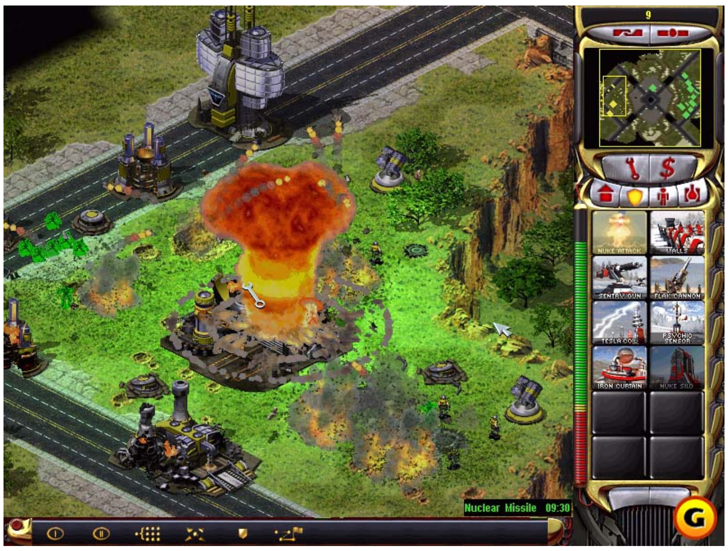


Figure 11.1: Screen shot of the game user interface in Red Alert 2.

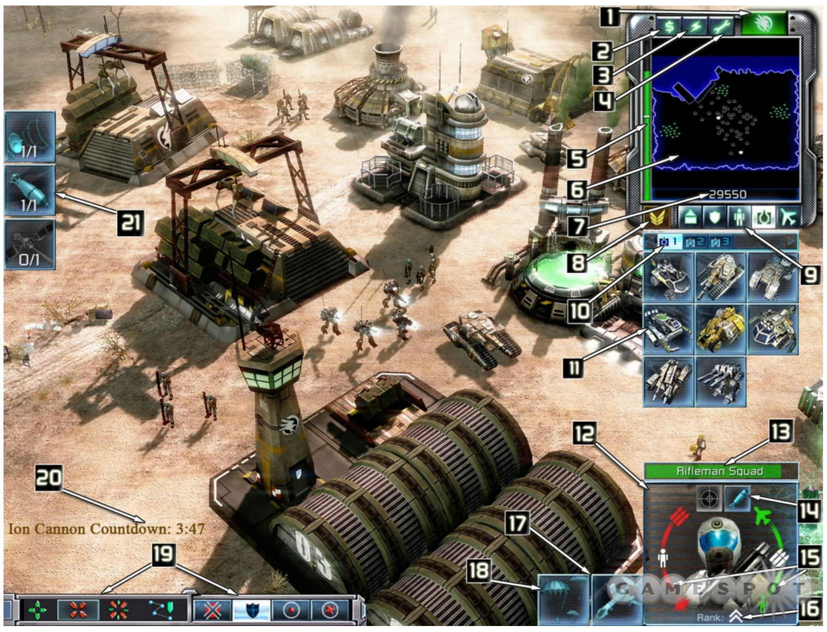


Figure 11.2: Screen shot of the game user interface in Command and Conquer 3.

CHAPTER 12

Full Implemented Grammar

12.1 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

12.2 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]] ::= /

12.3 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

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

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

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

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

[7] Uri Wilensky. Netlogo. Website, 1999-2011. URL: http://ccl.northwestern.edu/netlogo/models/index.cgi - Date seen: 23. mar. 11.

[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.