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:
SW4, 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 will develop a multi agent system formed as a wargame. To do that we use C# to make the wargame environment, and then we make our own language and compiler that will generate code to control the agents in the wargame.

method, experimentation, conclusion - to come...

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 to achieve these goal by designing and implementing a small language for controlling a multi agent system in the form of a wargame. 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

Contents

Part I

Introduction

In this chapter there will be an introduction to multi agent systems, agent oriented languages and existing multi agent environments.

Chapter 1

Multi Agent System

The purpose of a Multi Agent System (MAS) is to create scenarios in which independent and intelligent agents make self-interested 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 you 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. This goal can be defined in a lot of different ways, one of which is to make valued states that the agents strive to be in, the higher the value, the better the state.

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.

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.

sectionNet-Logo NetLogo is a popular and widespread MAS, developed by a man called Uri Wilensky in '99, who is currently working at the Northwestern University. [5] NetLogo features a very easy programming language for creating both Agents and 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.

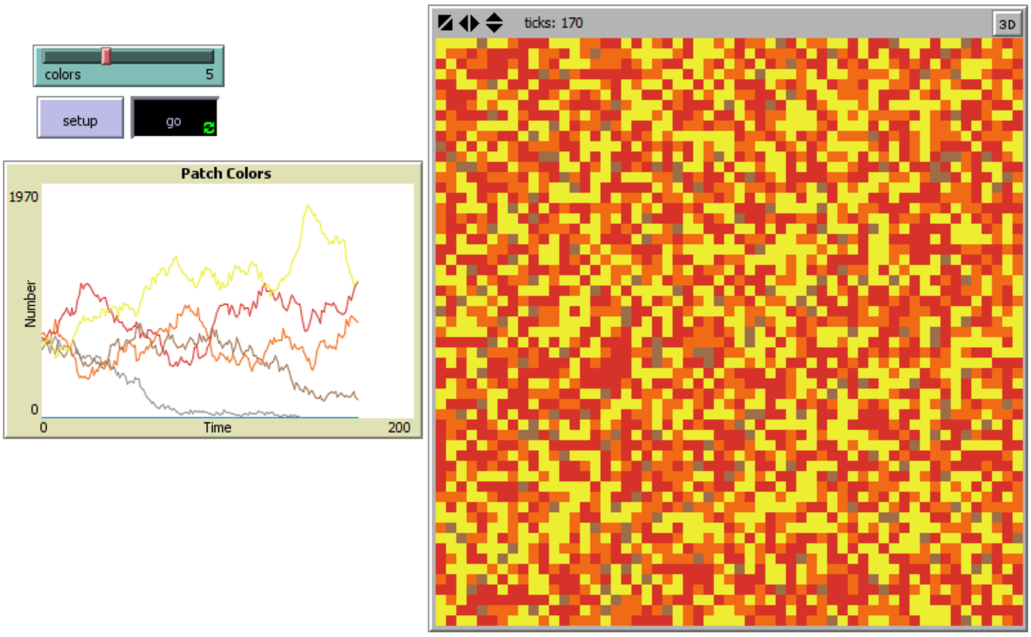
The code shown in the code-snippet 18 will generate a simple test with color-mixing, to simulate passing of genes.

```
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 1997 Uri Wilensky. All rights reserved.
; 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

Agent Oriented Language

Creating all of these MAS' and the environment to go with them, using a traditional programming language, can, however, be rather difficult and tiresome. The need for programming skill both limits the amount of people able to create a MAS and prolongs the amount of time required from people who have the necessary skills.

In order to overcome this problem, people have started to develop languages specifically designed to create MAS' and MAS-environments, these languages are called Agent Oriented Languages (AOL), and what they all have in common is added abstraction.

tail - we have introduced... *In this chapter there will be a description of the concept of our wargame.*

Chapter 4

Wargame Scenario

The game can have up to four players, where some can be set to move in predefined patterns, so that one can get a feeling of playing against an artificial intelligence.

When the application is started, the user chooses the size of the battle area, and the agents and teams are set up by the input of the generated XML-file 4.0.1.

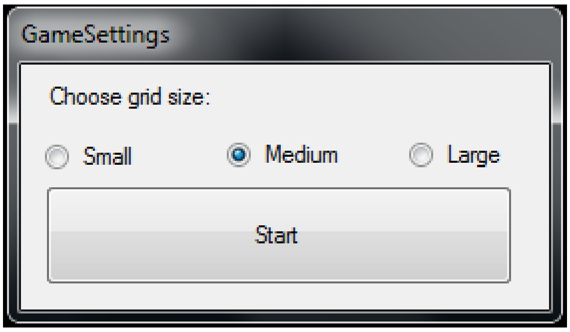


Figure 4.1: The player can choose between 3 diffrent sizes, small(13), medium(26) and large(46)

The user can type his commands in the *command center*, when the user is done giveing all his commands for his round, he can press the *End Turn* button to make his moves and end his round.

The moves available for the user to make is *up*, *down*, *left* and *right* (one coordinate at a time), and it is also possible to move several grids with an agent, if you select the agent and type the coordinates you want the agent to move to.

When a collision between agents from opposing teams occur, a random function is called, deciding which agent wins the fight, favoring the unit with the highest rank.

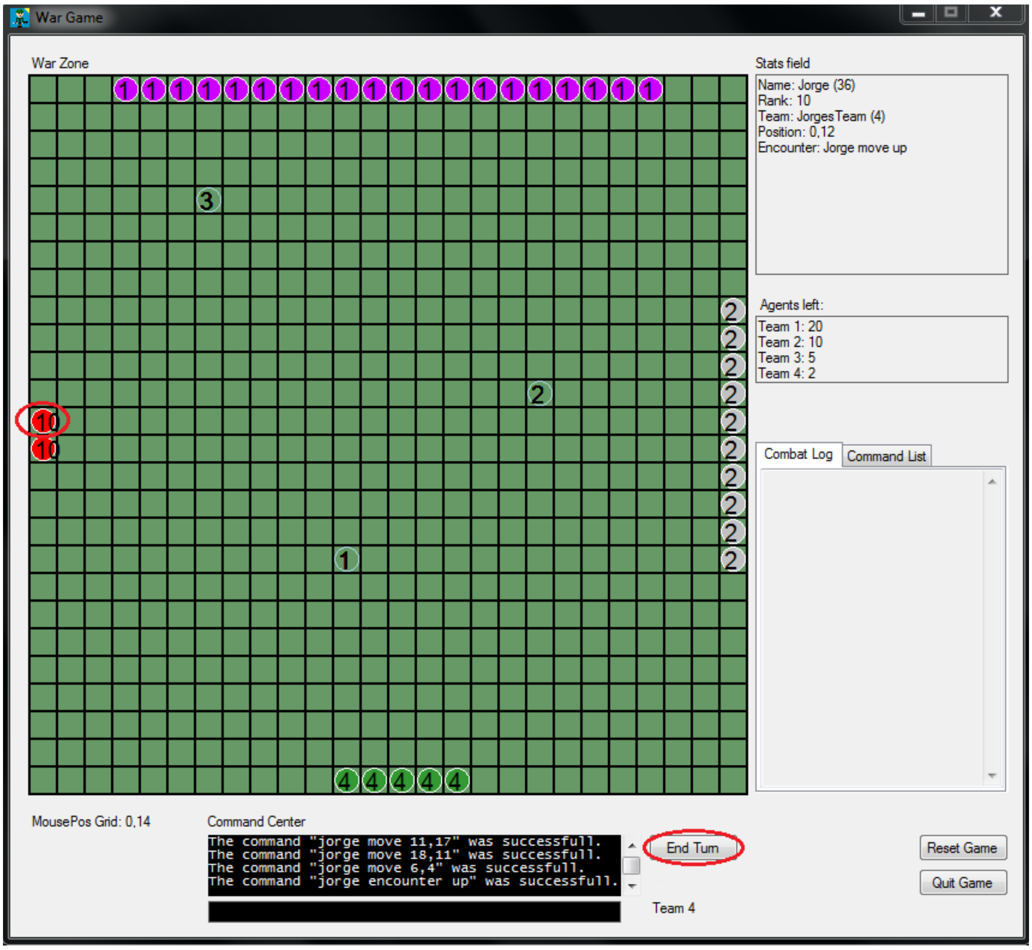


Figure 4.2: Example of an execution of a move-by-coordinate command. The user type the command `pink1 MOVE 14,3` and press *Execute*. For more about the commands, see appendix 19.0.6.

4.0.1 Agents and Action Patterns

The XML-document, which is loaded when the game initialized, sets the up the teams with number of players, division into squads, and rank of the agents. Also the action patterns are set by the XML-document.

tail - in this chapter we outlined...

Chapter 5

Compilers and Interpreters

Header...
tail...

Chapter 6

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?

Part II

Tools

Chapter 7

Language Components

In this chapter 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].

7.1 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, se table 7.1), 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	$X \vert Y$	any string generated either by X or Y
iteration	X^*	any string generated either by X or Y
grouping	(X)	any string generated by X

Table 7.1: 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 to 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, the we can replace this with an equivalent EBNF production rule: $N ::= X(Y)^*$. These two rules is 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 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 uncomplicated, then this specific substitution might simplify the grammar as a whole.

Starter Sets

The starter set of an regular expression X (*starters[[X]]*) is the set of terminal symbols that can start a string generated by X.

7.2 Semantics

The semantics of a programming language is a mathematical notation that explains langauge behavior. It defines the meaning of all the elements in a language [2].

Tail...

Chapter 8

Compiler Components

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 this chapter we will explain core concepts and ideas as to how to compile written code into executeable code. This section will describe how the compiler components can be implemented, and there may be some differences between this and the way the components are actually implemented in this project.

A basic compiler can be broken down to three simplet steps which are illustrated in **??**.

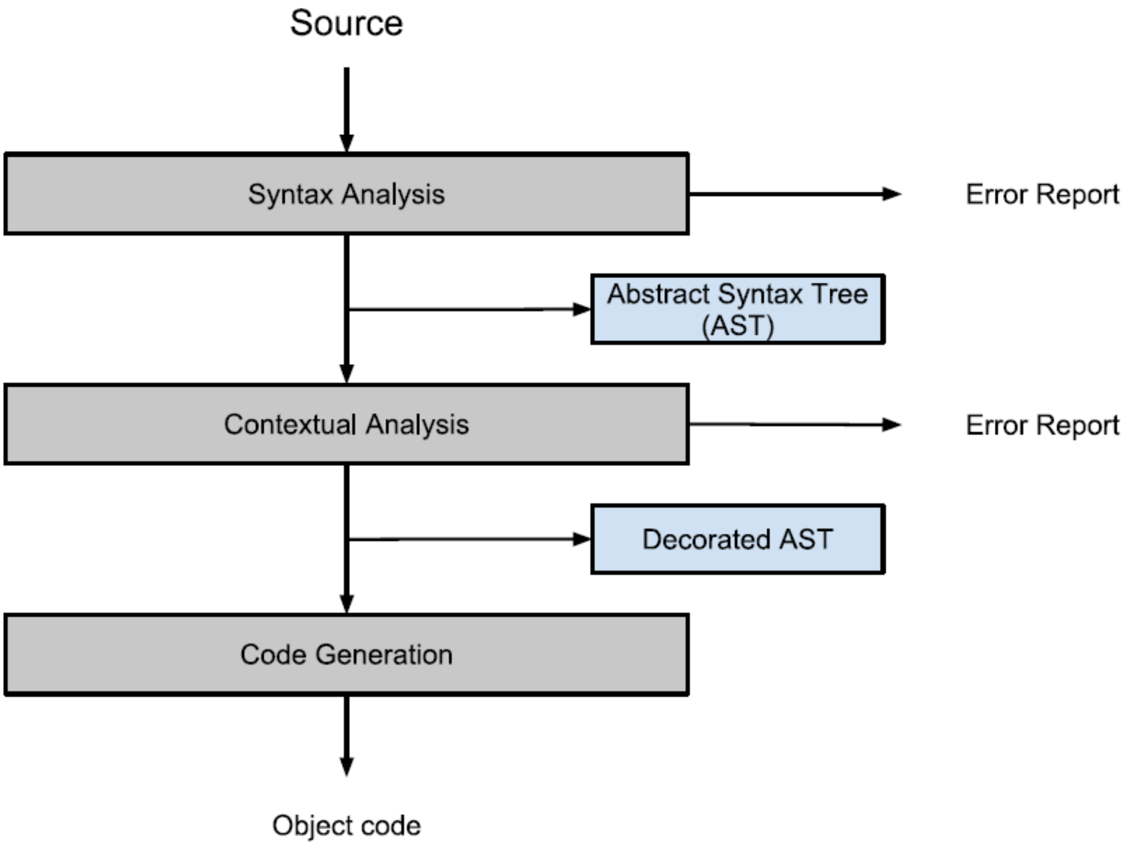


Figure 8.1: Illustration of the general structure of the compiler components.

8.1 Scanner

The scanner has the purpose to recognize tokens in the source program. This process is called *lexical analysis* and is a part of the *syntactic analysis*.

The terminal symbols are individual characters, which are put together to form the tokens [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 7.1.

2. Then there is for each EBNF production rule $N ::= X$ made a transcription to a scanning method `scanN`, where the body is determined by X.

3. The scanner has the following list of variables and methods: private variable `currentChar`, private methods `take` and `takeIt`, the private method `scanN` in (2) is improved to record token's spelling and kind, and last, a public method `scan`, which scans the combination 'Separator* Token', discarding the separator and returning the token.

See more about the BNF and EBNF notation in section 7.1 and see the full implementation of the grammar in the appendix 19.

8.2 Parser

The scanner 8.1 has the purpose to recognize tokens, and that leads to recognizing the input string and 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*. 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 consider the terminal symbols of a string, from left to right, and constructs its AST from top to bottom (from root node to terminal node).

8.2.1 Data Representation

Here is an example of how the *top-down* parsing algorithm works.

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

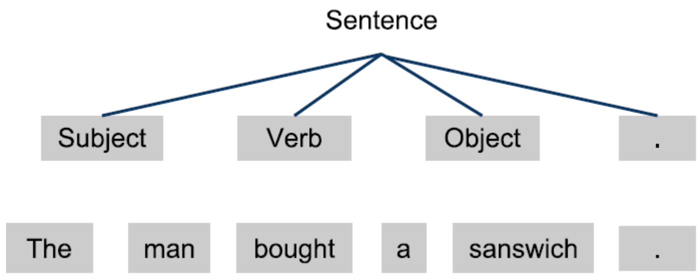


Figure 8.2: The first step for the parser is to decide what to apply ind the root node. Here it has only one option: "Sentence ::= Subject Verb Object."

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.

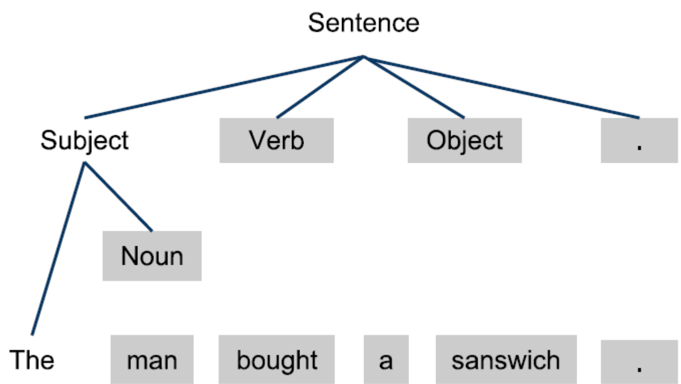


Figure 8.3: In the second step the parser looks at the stub to the left. Here the correct production rule is: "Subject ::= **The** noun".

The parser chooses the production rules by examining the next input terminal symbol. If the terminal symbol in figure 8.3 had been "A" then it would have chosen the production rule: "Subject ::= **A** noun".

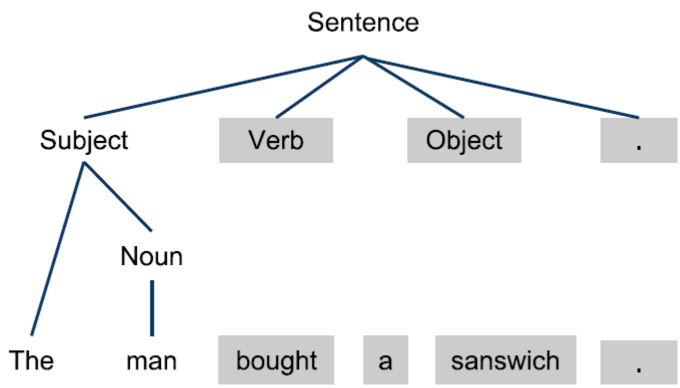


Figure 8.4: In third step the noun-stub is concidered, and the production rule becomes: "Noun ::= man".

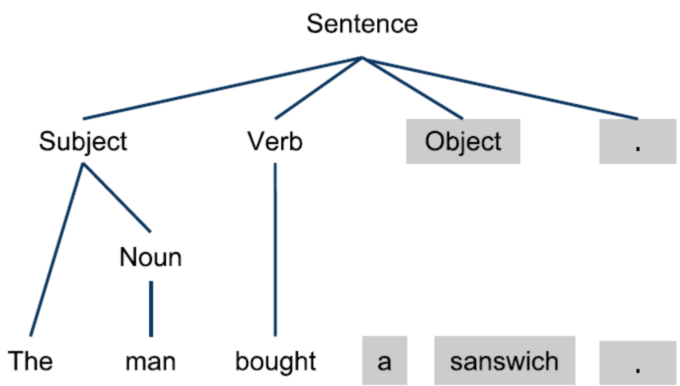


Figure 8.5: In fourth step the verb-stub is concidered, and the production rule becomes: "Verb ::= bought".

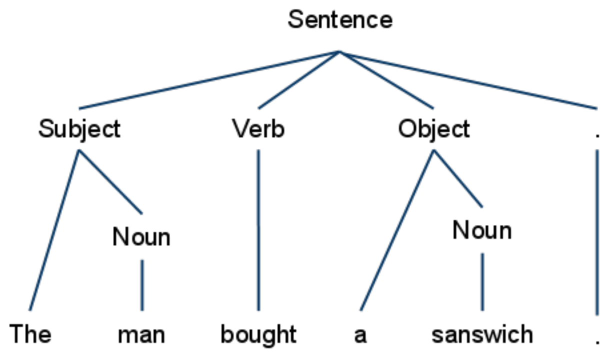


Figure 8.6: Here is the final syntax tree when the parser is done.

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 11.3 on how we have implemented the AST.

8.3 Code Generation

tail...

Part III

Implementation

Chapter 9

Language Documentation

header...

9.1 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 and floating points, 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 19.

9.2 Semantics

The semantics for the MAS language are operational semantics written in bigstep notation. The language encompasses:

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

9.3 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([someNumber])` where `someNumber` is the maximum unitpoints, which are then divided equally between each team. 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 9.1: While-loop

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

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

Source code 9.2: 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 9.3: 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 9.4: 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 9.5: 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.

tail...

Chapter 10

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 18.1 and 18.2 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 10.
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.

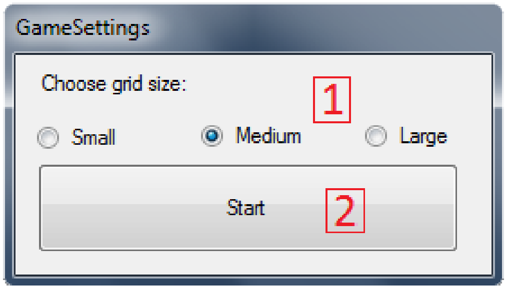


Figure 10.1: Screen shot of the game settings dialog box.

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.

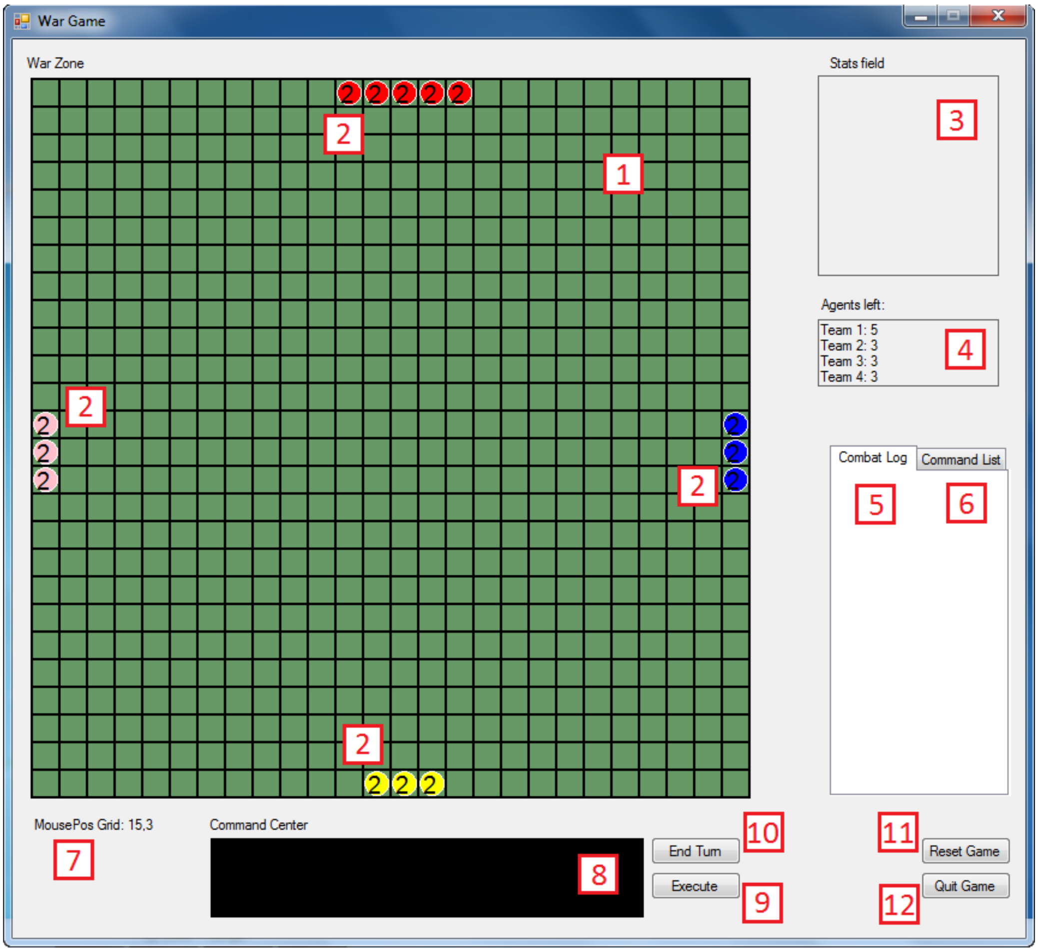


Figure 10.2: Screen shot of the game interface.

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);
```

```
            int it2 = (Grids / 2) - (agentsOnTeam2 / 2);
            int it3 = (Grids / 2) - (agentsOnTeam3 / 2);
            int it4 = (Grids / 2) - (agentsOnTeam4 / 2);
            foreach (Agent a in agents)
            {
                Point p = new Point();
                if (a.team.ID == 1)
                {
                    p = getGridPixelFromGrid(new Point(it1, 0));
                }
                else if (a.team.ID == 2)
                {
                    p = getGridPixelFromGrid(new Point(Grids -
                        1, it2));
                }
                else if (a.team.ID == 3)
                {
                    p = getGridPixelFromGrid(new Point(it3,
                        Grids - 1));
                }
                else if (a.team.ID == 4)
                {
                    p = getGridPixelFromGrid(new Point(0, it4));
                }

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

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

Source code 10.1: 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.

10.1 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.

Scanner

Parser

10.1.1 Visitors

Code Generation

Chapter 11

Compiler Components

header - in this chapter..

11.1 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 19.0.4. 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.

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

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

```
                }
            }
        }
```

Source code 11.1: 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.

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

Source code 11.2: 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.

11.2 Making the Parser

The parser8.2 takes the stream of tokens and keywords generated by the scanner and builds an abstract syntax tree (AST)8.2.1 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.
see section 11.3 for details on how the AST looks.

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

Source code 11.3: This is the main parsing method, which parses a mainblock and returns it as the AST.

```
private AST parseMainblock()
        {
            Mainblock main;
            switch(currentToken.kind)
            {
                case (int)Token.keywords.MAIN:
                    acceptIt();
                    accept(Token.keywords.LPAREN);
                    accept(Token.keywords.RPAREN);
                    main = new Mainblock(parseBlock());
                    accept(Token.keywords.EOT);
                    return main;
                default:
                    // Error message
                    accept(Token.keywords.ERROR);
                    return null;
            }
        }
```

Source code 11.4: 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.

11.3 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 8.2, creates a syntax tree. This syntax tree will for eksample parse the source code:

```
Main ( 400 )
{
  new Team teamAliens("Aliens", "#FF0000");
  new Agent agentAlice("Alice", 5);

  teamAliens.add(agentAlice);
}
```

Source code 11.5: Source code example.

To the AST:

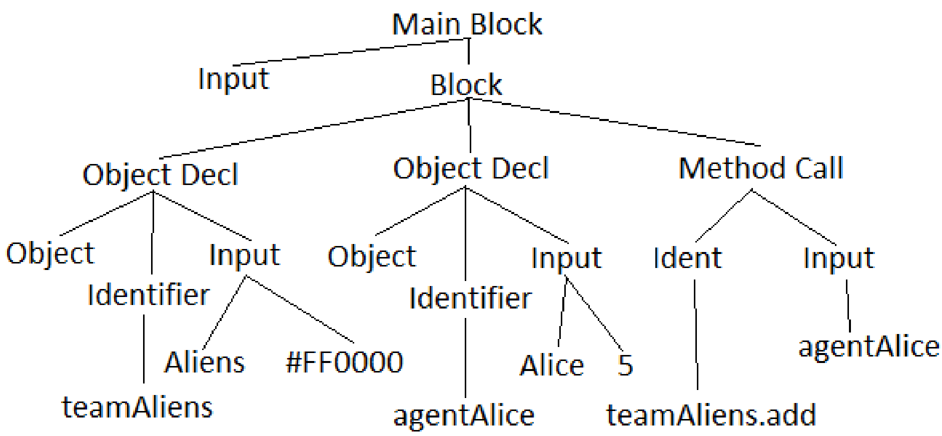


Figure 11.1: Example of the AST compiled from the source code above.

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:

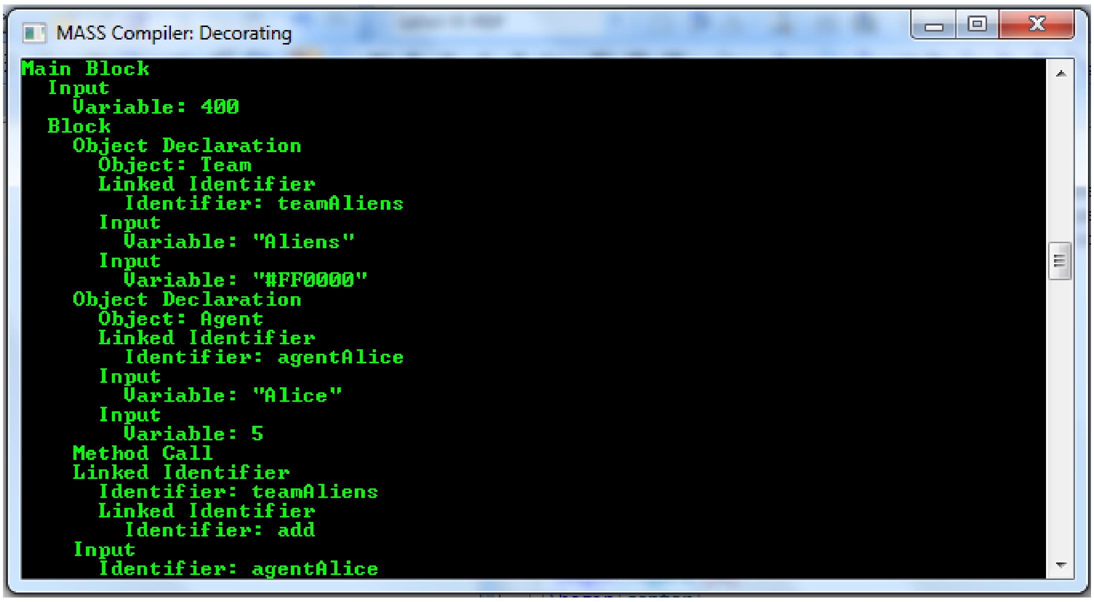


Figure 11.2: Example of the AST compiled with the MAS compiler.

11.4 Visitors

Once the parser has created the abstract syntax tree, there are still many checks that need to be done, such as type checks and input validation. This process is known as decoration, and refers to decorating the abstract syntax tree.
To do this, we utilize the visitor pattern, a design pattern 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 contains a number of commands. We do not know what type of each command is though, we only know that there is a command object. When that object is "‘visited"’, the code is automatically redirected to the correct function based on the type of the object that is visited.

As an example, say we are running through all the commands in a block.

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

```
}
```

Source code 11.6: 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 then say that the next command is a for-loop (which inherits from the Command class), the visit command will lead to the visitForCommand function being called.

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

Source code 11.7: The ForCommand class from the AST.

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

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

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

        forCommand.ForBlock.visit(this, arg);

        IdentificationTable.closeScope();

        return null;
    }
```

Source code 11.8: The visitForCommand function.

11.4.1 Decoration

The decoration of the abstract syntax tree can be a large task, depending on how many checks that need to be done. Therefor, instead of having just one visitor do all the decoration, we have several.

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

```
internal override object visitBlock(Block block, object arg)
    {
        IdentificationTable.openScope();
        ...
        IdentificationTable.closeScope();

        return null;
     }
```

Source code 11.9: A block is visited, and the scope is opened and closed respectively.

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:

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

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

Source code 11.10: 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.

11.4.2 Code Generation

The final visitor is the CodeGenerationVisitor. This visitor 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. Most of our

sub conclusion - in this chapter we have made...

Part IV

Discussion

header...

Chapter 12

Language Development

Chapter 13

Advantages in the MAS Language

Chapter 14

Usability

Chapter 15

Conclusion

tail...

Part V

Epilogue

header...

Chapter 16

Future Work

tail...

Part VI

Appendix

Chapter 17

Appendix

Chapter 18

Other Games

Figure 18.1: Screen shot of the game user interface in Red Alert 2.

Figure 18.2: Screen shot of the game user interface in Command and Conquer 3.

Chapter 19

Full Implemented Grammar

19.0.3 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

19.0.4 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]] ::= /

19.0.5 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

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

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