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 small language to control the logics of a multi agent system.

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 **ref. to study regulation**.
We are going 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 going to use Visual Studio and C#, because we have used these tools in earlier semesters and are used to the C# syntax.

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

Source code examples in the report is represented as follows:

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

Contents

Part I

Introduction to MAS

Chapter 1

Introduction

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

1.1 Multi Agent System

1.2 Agent Orientet Language

1.3 Existing Environments

tail - we have introduced...

Part II

Tools

Chapter 2

Language Components

Header...

2.1 Language Grammar

2.2 Semantics

Tail...

Chapter 3

Compiler Components

In order to give the reader of the Report 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 try to 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 is actually implemented in this project.

A basic compiler can be broken down to three simplet steps which are illustrated in 3.

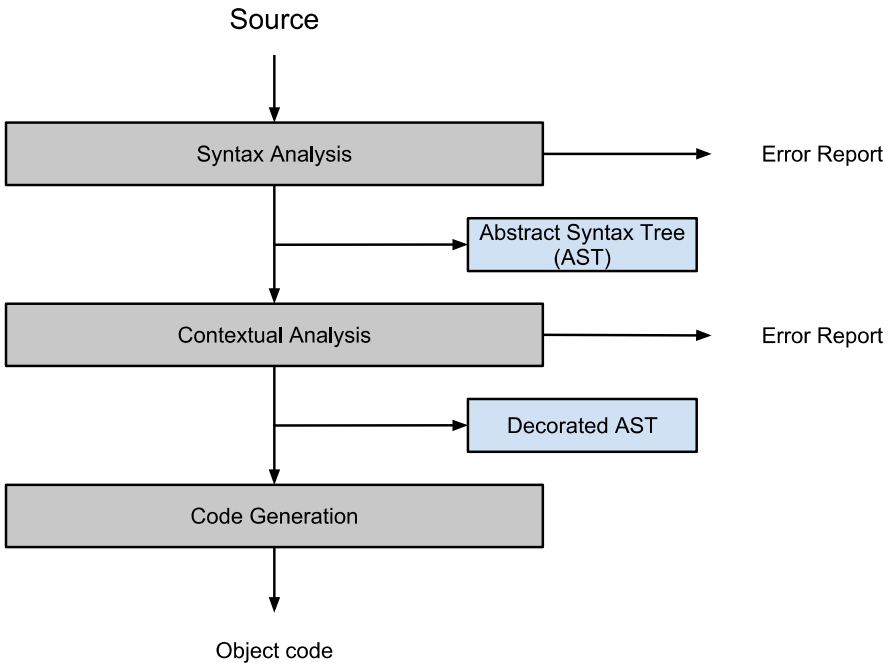

Figure 3.1: Illustration of the compiler components.

3.1 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 [**?**]. 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 3.1.1.
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.

3.1.1 BNF and EBNF

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

ENBF is...

3.2 Parser

3.3 Data Representation

tail...

Part III

Implementation

Chapter 4

The Wargame

header - in this chapter we will outline...

4.1 Wargame Scenario

The war game is initialized and the number of agents on the teams is chosen by the user. The first user types the first command, and clicks the button *Execute* to execute the command. When the user is done making his draws, he ends his turn by pressing the *End Turn* button. 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 make several moves 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, which decides which agent wins the fight, favoring the unit with the highest rank.

tail - in this chapter we outlined...

Chapter 5

Implementation of Compiler Components

header - in this chapter..

5.1 grammar

The grammatics is the heart of a programming language. It is what difines the rest of the compilerbase and what every aspect of language is made of. It is important when building the grammar for a language that one is clear of what every aspect of the grammar does. It is important that the language is not ambiguous, as this would could lead to misunderstanding in compile-time, and make wrong code. To define the grammatics of a programming language, one needs to define the very basics of the language. First one must define which things should be allowed with the language, and which should not. One of the things one should start defining is the diffrent types of the language. In our language we choose three types; num, string and bool. These will 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 type if the language.

In the grammar one need to define 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 diffrent things can be nested. A general program written in our language must consist of a mainblock, where every-

thing else is containt within. The mainblock will be made up by the keyword Main, followed by the two brackets '(' ')', followed by a block. The block consists of a starting bracket '' some commands and then a end 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 A.2.

5.2 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.

5.3 Making the Scanner

The scanner is an algorithm, which converts the string of text, the input, to a compilation 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 **??**. E.g. if the first character of a word is a letter, the word is automaticly assigned as an identifier and a string with the word is created.
When an identifier is saved as a Token, the Token class searches for any keyword, that would be able to match the exact string, e.g. if the string spells the word "'for"', the Token class changes the string to a **for** token.

```
1 public Token(int kind, string spelling, int row, int col)
2         {
3             this.kind = kind;
4             this.spelling = spelling;
5             this.row = row;
6             this.col = col;
7 
8             if (kind == (int)keywords.IDENTIFIER)
```

```
9               {
10                  for (int i = (int)keywords.IF_LOOP; i <= (int)
                        keywords.FALSE; i++)
11                  {
12                      if (spelling.ToLower().Equals(spellings[i]))
13                      {
14                          this.kind = i;
15                          break;
16                      }
17                  }
18              }
19          }
```

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

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

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.

5.4 Making the Parser

The parser is what takes the compilation of tokens and keywords generated by the scanner and builds an abstract syntax tree (AST) 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 of the AST, and returns that subtree, so it can be added to the AST.
For details on how the AST works, see section 5.5.

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

```
1  private AST parseMainblock()
2          {
3              Mainblock main;
4              switch(currentToken.kind)
5              {
6                  case (int)Token.keywords.MAIN:
7                      acceptIt();
8                      accept(Token.keywords.LPAREN);
9                      accept(Token.keywords.RPAREN);
10                     main = new Mainblock(parseBlock());
11                     accept(Token.keywords.EOT);
12                     return main;
13                 default:
14                     // Error message
15                     accept(Token.keywords.ERROR);
16                     return null;
17             }
18         }
```

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 kind. 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.

5.5 The Abstract Syntax Tree

5.6 Code Generation

5.7 The Graphical User Interface

The user interface is made as a windows form application. With Visual Studios designer tools it is simple to make a nice design with buttons, panels, and windows just the way you want.
The main idea of the design of the user interface is that there should be only a few buttons, 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 A.1 and A.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 5.7.
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.

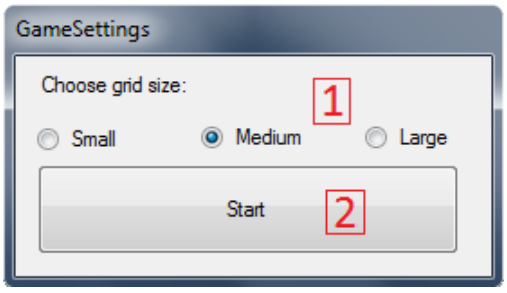

Figure 5.1: Screen shot of the game settings dialog box.

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.

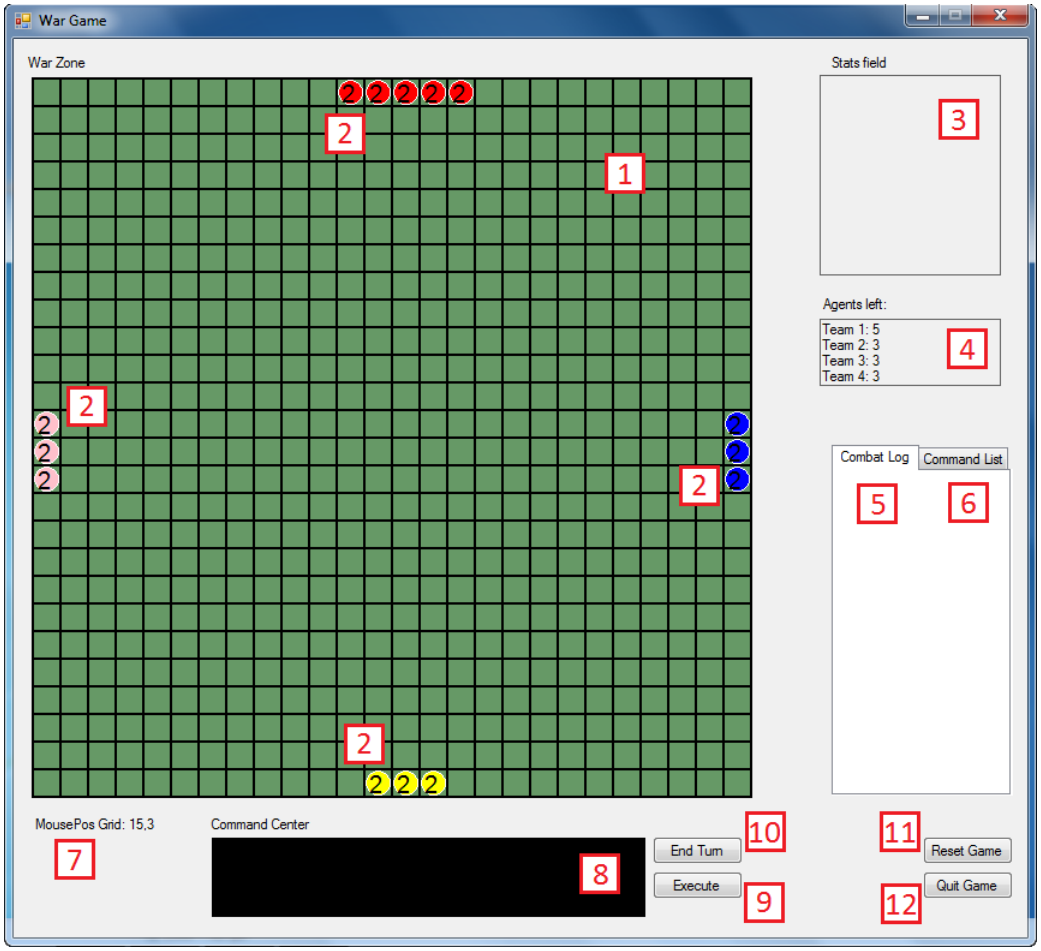

Figure 5.2: Screen shot of the game interface.

Drawing the Grid and Agents

The drawing of the grid and agents, is done using GDI+ [?] which makes it possible to draw graphic 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 agents start posistions are calculated by the following algorithem:

```
1                int it1 = (Grids / 2) - (agentsOnTeam1 / 2);
```

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

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.

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

Part IV

Epilogue

Chapter 6

Discussion

header...

6.1 Usability

tail...

Chapter 7

Epilogue

header...

7.1 Conclusion

7.2 Future Work

tail...

Appendix A

Appendix

A.1 Other Games

Figure A.1: Screen shot of the game user interface in Red Alert 2.

Figure A.2: Screen shot of the game user interface in Command and Conquer 3.

A.2 Full Implemented Grammar

A.2.1 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

A.2.2 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]] ::= /

A.2.3 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

A.2.4 Action Grammar

Declarative:

action ::= single-action EOL
identifier ::= Agent ID | Agent Name
Agent ID ::= num
single-action ::= identifier move-action

move-action ::= *MOVE*(move-option)
move-option ::= *UP* | *DOWN* | *LEFT* | *RIGHT* | *HOLD*
coordinate ::= num , num

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

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

Bibliography

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