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

Chapter 1

Introduction

header - in this chapter we will introduce...

1.1 Motivation

1.2 Tools

tail - we have introduced...

Chapter 2

The Wargame

header - in this chapter we will outline...

2.1 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 3

Compiler Components

header...

3.1 Scanner

3.2 Parser

3.3 Data Representation

tail...

Chapter 4

Implementation

header - in this chapter..

4.1 Grammar

4.2 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.

4.3 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 4.4.

```
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.

4.4 The Abstract Syntax Tree

4.5 Code Generation

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

Chapter 5

Discussion

header...

5.1 Usability

tail...

Chapter 6

Epilogue

header...

6.1 Conclusion

6.2 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.