Part I

Implementation

In order to give the reader a top-down understanding of our product, we find that it is very important that the reader understands basic concepts of compiling. In the first chapter we explain core concepts and ideas as to how to compile written code into executeable code. After that we outline our implementation of the compiler. Further more we descripe the graphical user interface to our MAS environment.

Compiler Components

There are a number of different kind of language processors, however, we focus on the ones important to our project: Translators. A translator is exactly what it sounds like; it is a program that translates one language into another, this being Chinese into English, or C# into Java.

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

1.1 Compilers

A compiler is basically a translator, typically capable of translating a language with a high level of abstraction (high level language), into a language that has a low level of abstraction (low-level language). This could for example translate the language C into runnable machine code. A compiler has the defining property that it has to translate the entire input before the result can be used, however, it will then be run at full machine speed. If the input is very large it may take quit a while to finish translating, other then that there are no disadvantages to the approach.

A basic compiler can be broken down to three simple steps, which are illustrated in 1.1.
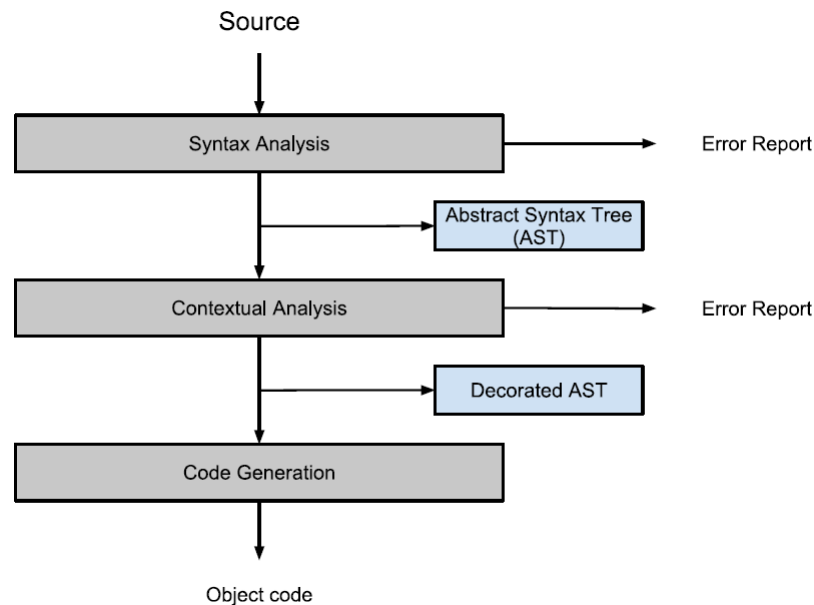
Figure 1.1: Illustration of the general structure of the compiler components.

1.2 Interpreters

An interpreter is also a translator, but where the compiler has to translate the entire input before the results can be used, the interpreter runs one instruction at a time from the input, thus enabling it to start utilizing the input right when it receives it. This boosts the time it will initially take to start running the output, but reduces the speed at which it can be run.,

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

1.3 Scanner

The purpose of the scanner is to recognize tokens in the source program. Tokens are abstractions of the code, and the scanner simplifies the code by recognising a string as a token. For example a + is recognised as an OP-ERATOR token. This process is called *lexical analysis* and is a part of the *syntactic analysis*.

Terminal symbols are the individual characters in the code, which the scanner reads and creates an equivalent token for [**?**]. 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 **??**.

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

3. The scanner needs the following variables and methods:

   (a) `currentChar`, which holds the currect character to scan.

   (b) `take()`, which compares the current character to an expected character.

   (c) `takeIt()`, which updates the current character to the next character in the string.

   (d) `scanN()`, as seen in step 2, though improved so it records the kind and spelling of the token as well.

   (e) `scan()`, which scans the combination 'Separator* Token', discarding the separator and returning the token.

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

1.4 Parser

The scanner 1.3 produces a stream of tokens. This stream provides an abstraction of the original input, and is used in determining the phrase structure, which is the purpose of the parser [**?**]. We strive to make the language unambiguous[1] to avoid the complication an ambiguous sentence would bring.

There are two basic parsing strategies, *bottom-up* and *top-down*, both of which produce an abstract syntax tree (AST). An AST is a representation of the phrase structure of the code, where the tokens found by the scanner are

---

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

turned from a list into a tree, as defined by the structure of your grammar. We will here expand on the *top-down* strategy, because that is what we have implemented.

The *top-down* parsing algorithm is characterized by the way it builds the AST. The parser does not *need* to make an AST, but it is convenient to describe the parsing strategy by making the AST. The *top-down* approach considers the terminal symbols of a string, from left to right, and constructs its AST from top to bottom (from root node to terminal node).

1.4.1 Data Representation

Here is an example of how the *top-down* parsing algorithm works, demonstrated with an AST [**?**].
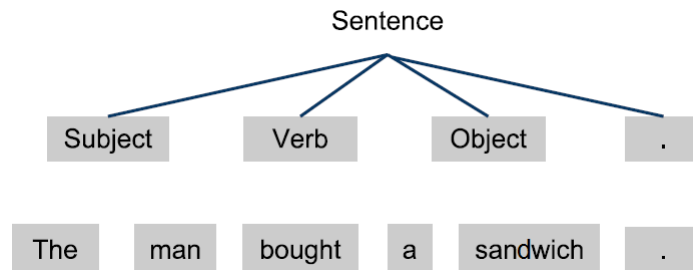

Figure 1.2: The first step for the parser is to decide what to apply ind the root node. Here it has only one option: "Sentence ::= Subject Verb Object."

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.
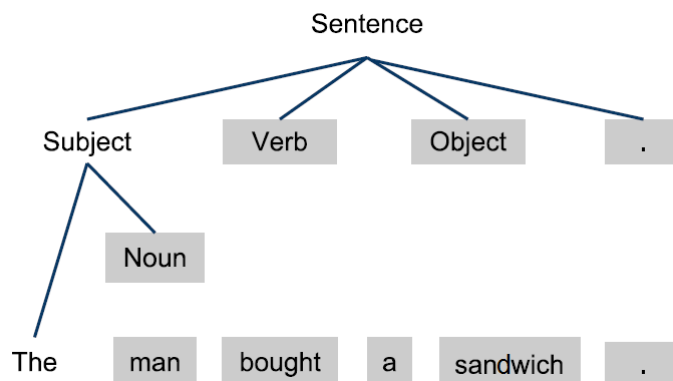
Figure 1.3: In the second step the parser looks at the stub to the left. Here the correct production rule is: "Subject ::= **The** noun".

The parser chooses the production rules by examining the next input terminal symbol. If the terminal symbol in figure 1.3 had been "A" then it would have chosen the production rule: "Subject ::= **A** noun".
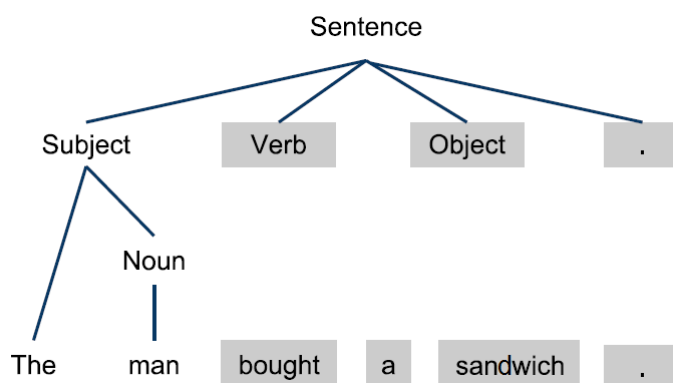

Figure 1.4: In third step the noun-stub is concidered, and the production rule becomes: "Noun ::= man".
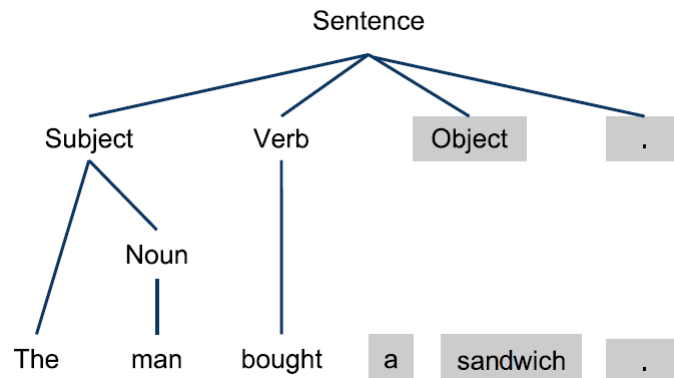
Figure 1.5: In fourth step the verb-stub is concidered, and the production rule becomes: "Verb ::= bought".
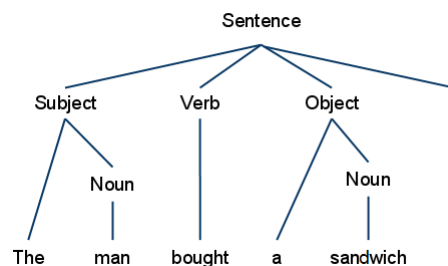

Figure 1.6: Here is the final syntax tree when the parser is done.

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 2.3 on how we have implemented the AST.

1.5 Decoration

Decoration refers to decorating the abstract syntax tree. Basically, up until this point we have only checked the structure of the code we are compiling, and decoration is the part where we do checks for validating the code it self. These are checks like type checks and scope checks.
To do this, we need a way of traversing the AST, while applying a lot of different logic to the various nodes in it. To this end, we utilize the visitor pattern.

1.5.1 Visitor Pattern

This design pattern is specifically used for traversing data structures and executing operations on objects without adding the logic to that object beforehand.
Using the visitor pattern is advantageous because we do not need to know the structure of the tree when it is traversed. For example, every block in the code contains a number of commands. We do not know what the type of each command is, we only know that there is a command object. When that object is "visited", the visitor is automatically redirected to the correct function based on the type of the object that is visited.
As an example, we will look at code from our own compiler. Say we are running through all the commands in a block.

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

Source code 1.1: Here is the code that makes sure every command in a block is visited.

This is done from within a visitor class, so "this" refers to an instance of the visitor. The reason the visitor is sent as input, is so all the visit functions can be kept in that visitor, and multiple visitors with different functionality can be used. If say, the next command is a for-loop (which inherits from the Command class), the visit function will lead to the visitForCommand function being called.

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

Source code 1.2: The ForCommand class from the AST.

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

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

9

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

Source code 1.3: The visitForCommand function.

1.6   Code Generation

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

A great tool for doing this is code templates. Code templates are recipes for what code should be written under the current circumstances, which makes the visitor pattern well suited for this task as well (see section 1.5.1).

CHAPTER 2

Implementation of Compiler

2.1 Making the Scanner

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

11

```
13                        {
14                            this.kind = i;
15                            break;
16                        }
17                    }
18                }
19            }
```

Source code 2.1: The token method with overload

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

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

Source code 2.2: The string array spellings.

The structure of the `Token` method applies for operators and digits as well. If the current word is an operator, the scanner builds the operator. If the operator is a boolean operator i.e. "<", ">", "<=", ">=", "==", 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 therefore 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, i.e. 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, which makes it easier to go back and forth in the list of tokens.

2.2 Making the Parser

The parser (see section 1.4) takes the stream of tokens and keywords generated by the scanner, and builds an abstract syntax tree (see section 1.4.1) from it, while also checking for grammatical correctness. To accomodate all the different tokens, each token has a unique parsing method, which is called whenever a corresponding token is checked. Each of these methods then generate their own subtree which is added to the AST.

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

Source code 2.3: This is the main parsing method, which parses a mainblock and returns it as the AST.

```
private AST parseMainblock()
        {
                Mainblock main;

                accept(Token.keywords.MAIN);
                accept(Token.keywords.LPAREN);
                Input input = (Input)parseInput();
                accept(Token.keywords.RPAREN);
                main = new Mainblock(parseBlock());
                accept(Token.keywords.EOT);

                main.input = input;

                return main;
        }
```

Source code 2.4: 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 is no semicolon, the parser returns an error together with the line number and token which did not match an expected token.

2.3   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 1.4, creates a syntax tree. This syntax tree will for eksample parse the source code:

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

Source code 2.5: Source code example.
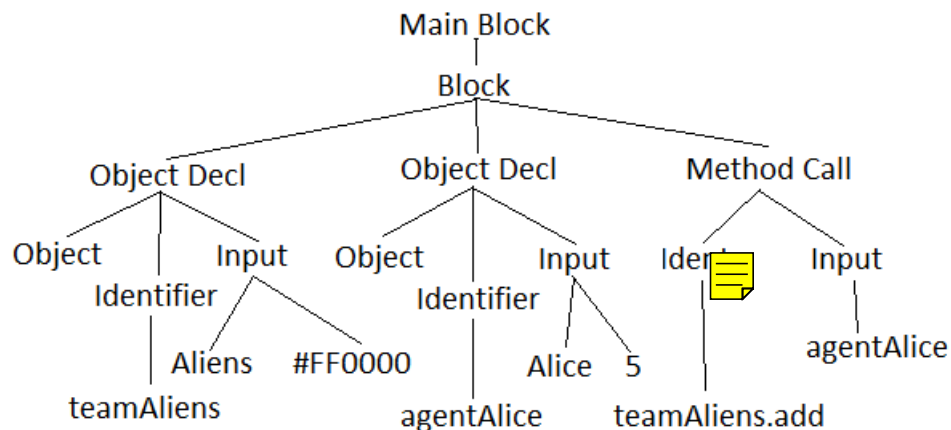
To the AST:

Figure 2.1: Example of the AST compiled from the source code above.

The AST can be printed by a pretty printer[1] to give a better overview of the compiled source code. In the MASSIVE compiler, the pretty printer prints all completed parses in the windows console. The MASSIVE pretty printer indents whenever a new branch is added. The source code above will be printed as seen in figure 2.2.
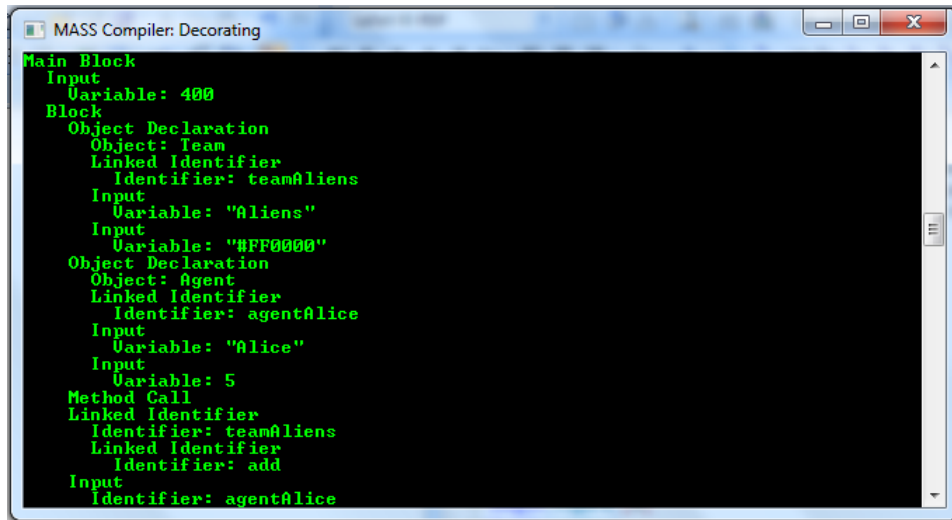
---

[1]A method for printing ASTs.

Figure 2.2: Example of the AST compiled with the MAS compiler.

2.4 Decoration

To decorate the AST we use the visitor pattern (see section 1.5.1) with several different visitors handling different parts of the task.

2.4.1 Type and scope checking

The first one is the `TypeAndScopeVisitor` which visits every node of the abstract syntax tree, and checks if the types and scopes of variables in the code are correct. Therefore this is where type safety is enforced in the compiler. This works by taking the type of the variable's token and comparing it to the values it is being used with.

In the scope checking we want to make sure that variables are not used outside 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.

```
1  internal override object visitBlock(Block block, object arg)
2      {
```

```
3            IdentificationTable.openScope();
4            ...
5            IdentificationTable.closeScope();
6
7            return null;
8        }
```

Source code 2.6: A block is visited, and the scope is opened and closed respectively.

2.4.2 Input validation

The second decoration visitor is the `InputValidationVisitor`. The job of this visitor is to make sure that all methods and constructors in the language recieve the proper input, depending on the available overloads. The overloads in our language represent the option for methods and constructors to work with different inputs. For example are all the following declaration are legal in our langauge:

```
1   new Team teamAliens("Aliens", "#FF0000");
2   new Team teamRocket("Team Rocket");
3
4   new Agent agentJohn("John", 5, teamAliens);
5   new Agent agentJane("Jane", 5);
```

Source code 2.7: Examples of overloads.

Every overload of every method and constructor in the language is handled as a class of its own in the compiler. The compiler then takes the information it needs, to determine if the given input i valid, from these classes. It is therefore possible to add new overloads to existing methods and constructors, as well as add new methods and constructors, because you only need to create a new class for it and initialize it.

2.4.3 Variable Checking

The `VariableVisitor` is the third visitor, and its job is to check if the variables that are declared are also used. While this will catch every unused variable, the main reason for the creation of this visitor is to catch unused objects, so the compiler can warn about unused agents, squads and teams.

2.5 Code Generation

In order to print the C# code, we must again traverse the AST, and determine what code should be printed. Therefor, as with the decoration process, we use a visitor (see section 1.5.1) to accomplish this. This visitor is the CodeGenerationVisitor and is responsible for printing out the correct C# code, such that it can be compiled an run without errors. To accomplish this, we use code templates.

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

For the methods in our language, we have a different solution though. Every class for a method or constructor, see 2.4.2, in our language must define an overload for the method PrintGeneratedCode.

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

Source code 2.8: The code printed for the squad constructor.

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

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

17

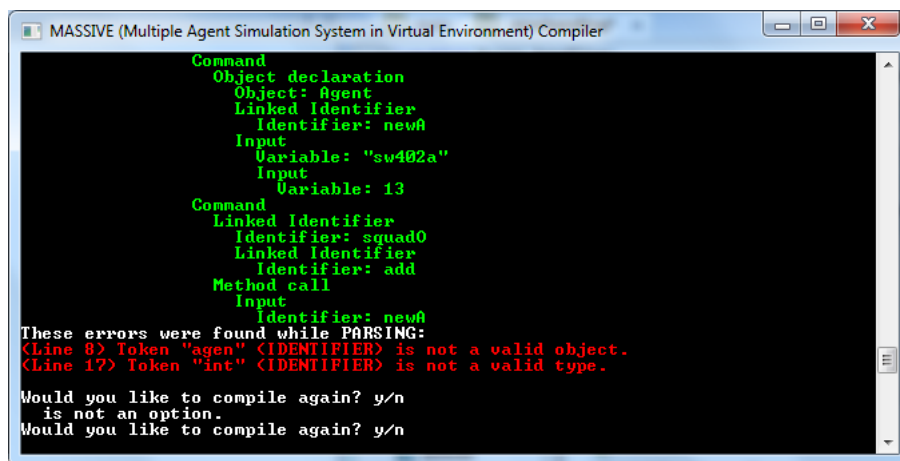Source code 2.9: The code printed for the agent constructor taking three arguments as input.

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

2.6   Error handling

It is very important that a programmer knows if the code he is writing is correct or not, so it's very important that the compiler tells him of any errors it encounters. Our compiler can catch errors after every parsing of the code it does, and it will also complete the parse, so it can report every error encountered in that parse.
To make the compiler easier to use, we have tried to make the error messages very descriptive. The programmer also gets a choice of whether he wants to print the compilation of the code and error markers will be printed in the correct places if he does. We have also made it such that the programmer can recompile his code once he has corrected any errors without restarting the compiler.
The are also warning messages, but these only occur during the variable check (see section 2.4.3). The programmer can choose to either recompile or continue with the current compilation when a warning has been found.

Figure 2.3: An example of how the compiler handles errors.

18

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 **??** and **??** 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 3.
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.
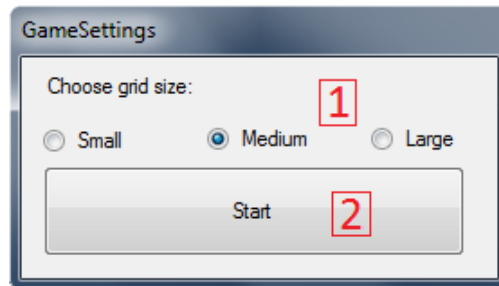
Figure 3.1: Screen shot of the game settings dialog box.

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

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

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

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

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

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

9. *Execute x5* button - executes the typed in command in the *command center*, and ends the turn. This cycle is done 5 times for all teams.

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.

13. *Simulate* button - starts a simulation, where the game starts and runs until the game is over.
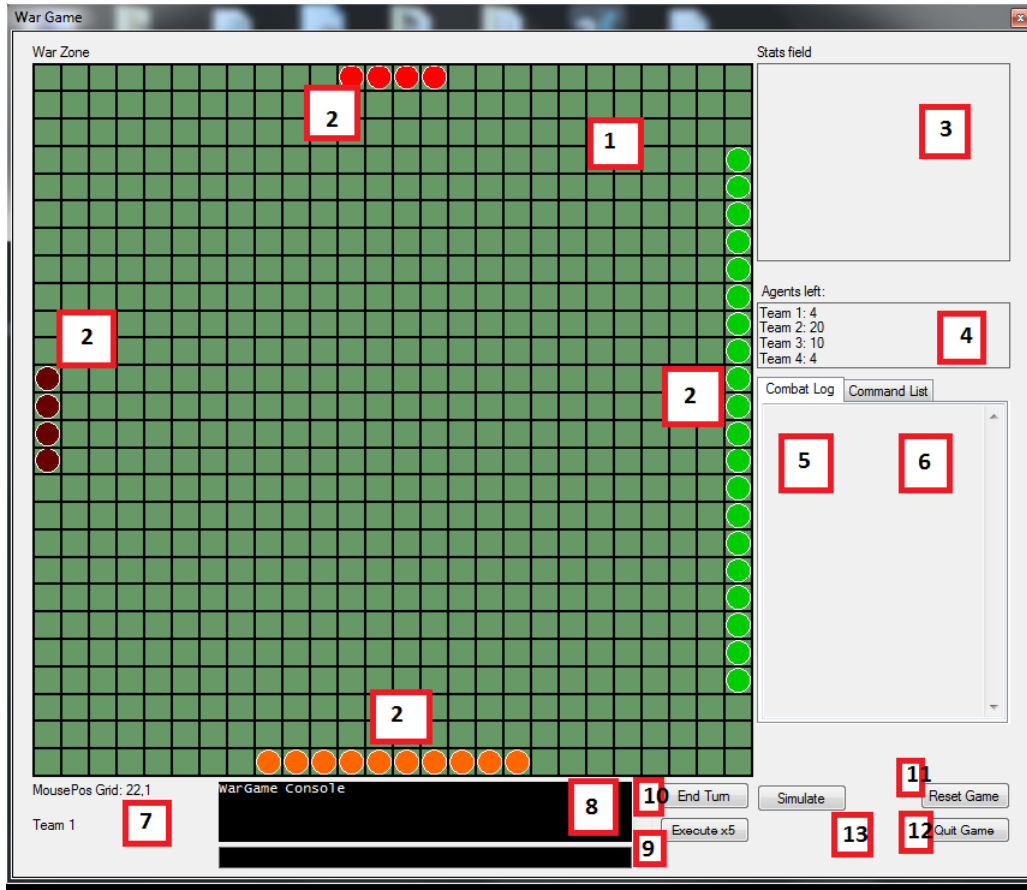

Figure 3.2: Screen shot of the game interface.

Drawing the Grid and Agents

The program make use of GDI+ [**?**] 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:

```
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   }
```

Source code 3.1: 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.

3.1 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.

3.1.1 Contextual Analysis & Code Generation

The contextual analysis is the decoration of the AST, which is done by traversing the AST with the visitors. In the contextual analysis, code generation is build in, since there is no need to parse the AST more than once, when all information used by the move function is given cronologically The first part of the decoration, is to verify the identification of the command.
To verify the identification, the decorator finds the unit or units the user wants to move. E.g. the user gives the command "squad 1 move down".
The parser then determines that the identifier "1" is a squad, and stores its token as a SquadID. The decorator then searches for the squad identifier in the squad list, and calls the move method to execute the action ("move down").

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

Source code 3.2: Example of the determination of the identifier in the visitors, this part identifies SquadID.

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

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

Source code 3.3: Snippit of the identification of the units in a squad.

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

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

```
5        return ;
6    }
```

Source code 3.4: Code snippit, when the encounter stance is chosen instead
of move.

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

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

Source code 3.5: The method moving a unit if, the move option is an action
pattern.

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

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

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

We have implemented...