|  |  |
| --- | --- |
|  | |
| **U.V. 6.1: “Animo”**  **CI – 2019 – SPID – SLS** | |
|  | IETA Huseyin KIZIL  huseyin.kizil@ensta-bretagne.org  Supervisor: Jean Christophe Le Lann |

# Acknowledgment

I would like to pay tribute to M. Jean-Christophe Le Lann under whose direction this project has been conducted. The theoretical knowledge he provided for the proper understanding of the subject was essential. In addition, the technical expertise of M. Le Lann was an extremely helpful. Without his assistance, this project would not have been carried out correctly.

# Abstract

The current project tackles the uncovered topic of log file. Analyzing log files composed of several thousands of lines is a tedious work. Firstly, the log files are not necessarily readable. Then, much information is unnecessary. The objective of this study is to develop a tool which will analyze log file and generate animation according to the user will. Hence, the challenges are to develop two Domain Specific Languages and a compile. The first DSL set rules defining how log files must be structured. The second DSL set rules defining how the user must describe the object of the log file and how those objects will be animated. The compiler, Animo, parses and analyzes the two files provided by the user then generates output code. The output code depends entirely on the instructions written in the object file by the user.

Table of contents

[Introduction 5](#_Toc3130050)

[1 Presentation of the problem 5](#_Toc3130051)

[1.1 Log files 5](#_Toc3130052)

[1.2 State of art 6](#_Toc3130053)

[1.3 Action plan 7](#_Toc3130054)

[1.3.1 Domain Specific Language 8](#_Toc3130055)

[1.3.2 Compiler 8](#_Toc3130056)

[2 Domain Specific Languages 9](#_Toc3130057)

[2.1 Grammar – EBNF 9](#_Toc3130058)

[2.2 Log DSL 10](#_Toc3130059)

[2.1.1. Grammar of the log language 10](#_Toc3130063)

[2.1.2. Lexicon of the log language 11](#_Toc3130064)

[2.2. Object DSL 11](#_Toc3130065)

[2.2.1. Grammar of the object language 11](#_Toc3130066)

[2.2.2. Lexicon of the object language 12](#_Toc3130067)

[2.3. Semantic rules 12](#_Toc3130068)

[3. Compiler 14](#_Toc3130069)

[3.1. Compiler theory 14](#_Toc3130070)

[3.1.1. Compiler history 14](#_Toc3130071)

[3.1.2. Operating principles of a compiler 15](#_Toc3130072)

[3.2. Compiling process 16](#_Toc3130073)

[3.2.1. Lexical analysis 16](#_Toc3130074)

[3.2.2. Syntactic analysis 17](#_Toc3130075)

[3.2.3. Contextual analysis 17](#_Toc3130076)

[3.2.4. Code generation 18](#_Toc3130077)

[3.3. Implementation 18](#_Toc3130078)

[3.3.1. Implementation of a scanner 18](#_Toc3130079)

[3.3.2. Implementation of a parser 19](#_Toc3130080)

[3.3.3. Implementation of an abstract syntax tree 20](#_Toc3130081)

[3.3.4. Visitor pattern 20](#_Toc3130082)

[3.3.5. Implementation of the contextual analyzer 21](#_Toc3130083)

[3.3.6. Implementation of code generator 21](#_Toc3130084)

[4. Results 22](#_Toc3130085)

[5. Alternative solution 22](#_Toc3130086)

[6. Project management 23](#_Toc3130087)

[Conclusion 25](#_Toc3130088)

[Annex 26](#_Toc3130089)

[Bibliography 27](#_Toc3130090)

# Introduction

The log files are made to keep track of the history of a system, a server, a software, a mobile phone or of a communication. Those log files are composed of lines. Each line describes an event which occurred at a specific time. Those lines contain information of the event. A log file is composed of more than thousands of lines. Hence, a human being will spend an inordinate amount of time to analyze such files. The purpose of the current project is to develop a tool to provide a solution to animate the log files. Indeed, visualization would permit users to concretely see the events how they would like to see them. Firstly, it is more pleasant to watch an animation rather than to read a long text. Secondly, it is time-saving. Thirdly, the user will be able to filter information and to focus on what he would like. To achieve this, a compiler must be implemented. The user will specify to the compiler Animo how event must be animated, and which kind of information must be taken into account. Those specifications must be made in another file (object file). The compiler, Animo, will then analyze the log file. For each event of the selected information, an animation instruction will be generated. Compilers are not similar to human brains. They cannot adapt to any kind of log syntax nor to understand the user wishes. Consequently, to be able to process log information and instructions on how to animate events, grammar structures must be set. In this way, Animo could treat information. Hence, the construction of Animo lies in three tasks: to define a specific language to which any log file will fit, to define another specific language to allow the user to define how event must be animated and construct a compiler which will analyze the log files and the object files then generate animation.

# Presentation of the problem

## Log files

Une image contenant texte

Description générée avec un niveau de confiance très élevé “LOG is the file extension for an automatically produced file that contains a record of events from certain software and operating systems. While they can contain a number of things, log files are often used to show all events associated with the system or application that created them.”[[1]](#endnote-1) Typically, log files are plain text and readable by humans. They can be applied in several domains (robotics, data, embedded systems, operating systems, servers …). An example of a log file from a web server is given below (Fig.1) [[2]](#endnote-2). Each line is dated precisely, and each line is similar to the others. A certain kind of pattern is followed: date, action and information which vary from one line to another one.

Figure 1 Log file from a web server

Extracting and processing information from a single type of log file in a specific context is feasible. A human is easily able to analyze such file. Tools can be developed to process one kind of log files and representing them in any way. A challenging task is to develop a program which would handle the treatment of any kind of log files.

The goal of the present project is to develop a language description of animated scenes which would animate any kind of log file, a generic log file analyzer. Log files could contain any kind of data: location, water level, temperature, data … The task is hence twofold: firstly, to propose a language to which the log files will have to correspond; secondly to develop a compiler to analyze files and generate animations.

## State of art

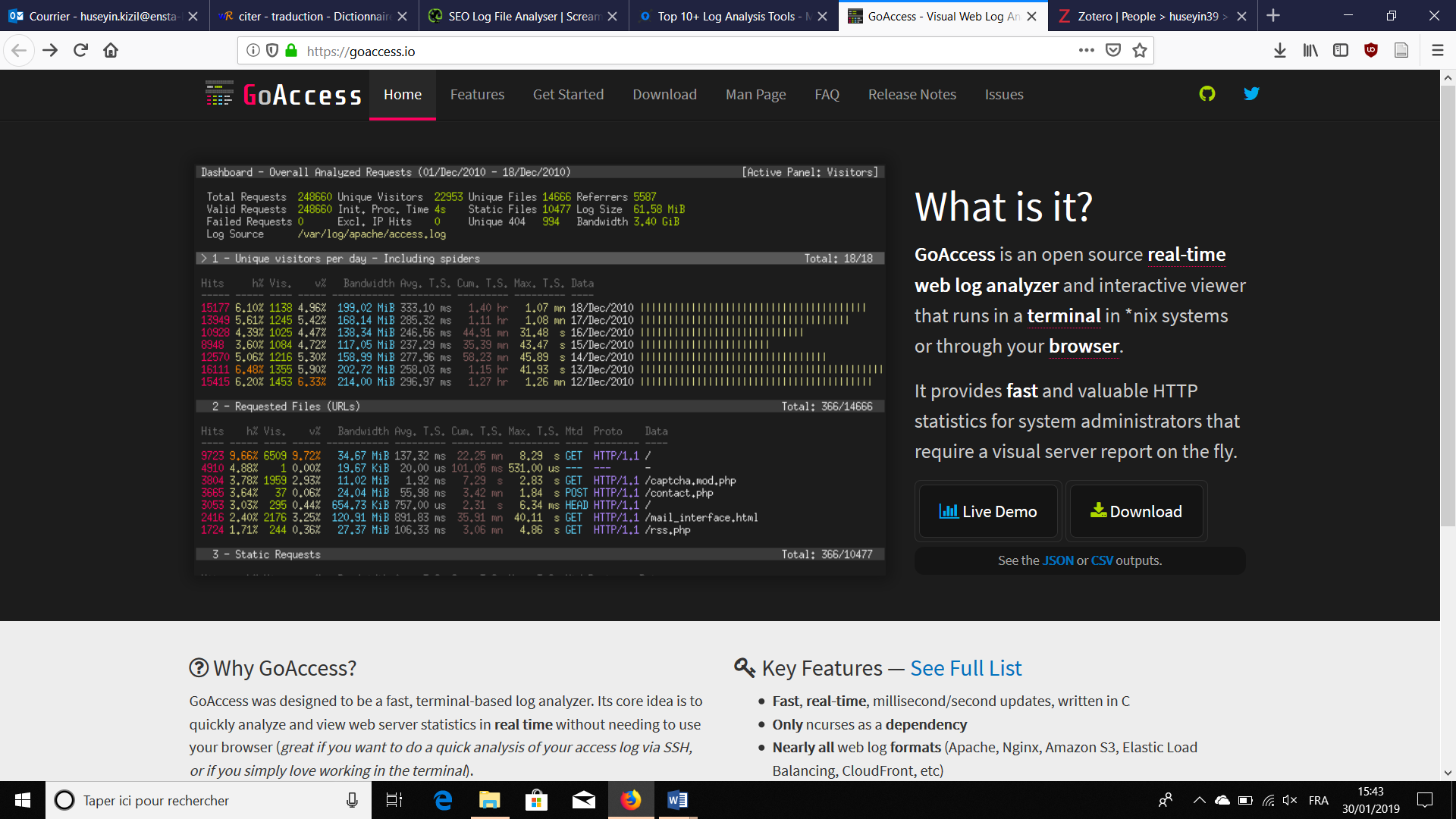
This is a developing and promising subject. Several log analyzers currently exist. However, they mostly (if not all) target web log files since the market is large and the demand is huge. Some of the best-known log file analyzers can be listed.

Figure 2 GoAcess

Firstly, there is the well-known web server analyzer GoAccess. “GoAccess is an open source real-time web log analyzer and interactive viewer that runs in a terminal in \*nix systems or through your browser. It provides fast and valuable HTTP statistics for system administrators that require a visual server report on the fly.”[[3]](#endnote-3) Its advantage is its status: open source. That means it is free and adaptable to personal use. Its developer community is quite large and responsive as well. The project has more than 8,000 stars on GitHub, that means GoAccess is maintained and popular.

Secondly, commercial log analyzers exist of course such as (Logz, [Loggly](https://www.loggly.com/) …). Their purpose is basically similar, to analyze log files coming from servers and websites. They work with technologies such as PHP, RubyOnRails, Django … for websites. Concerning operating systems, they tackle both Windows and Linux. In addition, features to analyze log files for containers (Docker) and Servers (Apache) are proposed. Finally, cloud (Amazon) logs are in the scope as well.

Thirdly, there are specific log analyzers too. DatCon/CsvView[[4]](#endnote-4) addresses this issue for a small range of drones (Phantom 3, Phantom 4, Phantom 4 Pro, Inspire 1, Spark, and Mavic Pro drones). This tool creates animations to visualize location on GoogleMap and on a graph.

At this time, there is no generic log analyzer suited to process any kind of data. This project is extremely challenging! To achieve this, a scalable model will be first produced that could be extended later.

## Action plan

The two significant objectives of Animo are: to build a comprehensive Domain Specific Language (DSL) to cover the log file domain and to construct the appropriate compiler to generate code for animation.

### Domain Specific Language

The purpose of a Domain Specific Language is to address a particular domain (log file here) compared to a General Purpose Language (GPL) such as Java or Python which proposes solutions to any kind of computational problem. HTML is a DSL for web pages for example. DSL specifies the rules to follow via a grammar. Grammars are used to describe languages by generating sentences. In the English language, for example, a sentence is considered correct if it is constructed with a subject, then a verb and possibly an object. Our topic is subject to 2 DSL.

The first one concerns the way log files have to be written so that Animo will be able to process them. Consequently, rules have been set: a log file must start with ‘begin’ and end with ‘end’. The lines composing the file must follow a specific grammar structure:

time unit: object\_ID action (x, y…);

The purpose of the second DSL is to give the possibility to the user to clarify how his objects of the given log file will be animated. Concretely, he will firstly define for each kind of action how objects will graphically be presented. Then, he will describe the animation to be done for each line of such action. A typical ‘object’ file would contain lines like:

Description object\_ID {{….}}

Animation object\_ID action\_iD (p1, p2 …) {{….}}

This design of DSL is scalable since it may be subject to changes. In addition, new ideas will emerge. The specification of the language is hence crucial.

### Compiler

A compiler must be written to transform the log file plus the object file into another file to obtain an animation. A compiler is a software program that transforms a code written in one language (language defined for Animo) into another programming language (programming language of animation in this topic). This tool identifies any errors as well. It is composed of 2 parts.

The front-end part of a compiler verifies the accuracy of the syntax and the contextual meaning according to the given grammar. It generally consists of 3 phases:

* Lexical analysis: associate characters to form tokens (categories of the grammar, for instance: verb, noun….) with their value
* Syntactic analysis: associate tokens to form “sentences” based on the grammar
* Contextual analysis: checks the semantic meaning and adds information for the back-end

The mission of the back-end is essentially to generate the target code, usually accompanied by an optimizer. A traditional compiler generates intermediate code after the contextual analysis. This code is between high-level language and the machine language. The implementation of intermediate code is helpful for memory management, register allocation … The optimizer enhances the intermediate code. In the current study, the idea of developing an optimizer is irrelevant since there is not intermediate code.

# Domain Specific Languages

Language structure specification is entirely made of EBNF rules. Those rules are used to check the structure of the “sentences”. Semantic rules are specified to complete the definition of the language. Two languages have been implemented. Consequently, two grammars have been developed using EBNF.

## Grammar – EBNF

When describing languages, Backus-Naur form (BNF) is a formal notation for encoding context-free grammars intended for human consumption. Extended Backus-Naur form (EBNF) is a collection of extensions to Backus-Naur Form. An EBNF consists of terminal symbols and non-terminal production rules which are the restrictions governing how terminal symbols can be combined into a legal sequence. Examples of terminal symbols include alphanumeric characters and punctuation marks. The rules in EBNF has the following structure:

<NonTerminalSymbol> ::= expansion

The symbol ‘::=’ means “expand into”. An expansion is an expression containing terminal symbols and non-terminal symbols, joined together by sequencing and choice.

A vertical bar “|” indicates choice. The expression (1) means A either expands to B or C.

<A> ::= <B> | <C> (1)

Square brackets around an expansion, indicates that this expansion is optional. A star ‘\*’ points out that the expansion may be repeated 0 or more times while ‘+’ indicates that the expansion must be repeated at least once. Curly brackets are used to precise the number of possible occurrences. The expression (2) shows that an integer is composed of 1 or more digits. The expression (3) describes

Integer ::= [0-9]+ (2)

To indicate precedence, EBNF grammars may use parentheses, (), to explicitly define the order of expansion. The expression (4) describes how mathematical operations work.

Operation ::= Integer ( “+” | “-”) Integer (3)

Consider the following grammar:

<Mathematic> ::= <Operation>

<Operation> ::= <Positive-Integer> ( “+” | “-”) ( <Positive-Integer> | <Operation> )

<Positive-Integer> ::= [0-9]+

The addition ’45 + 12 -5’ is correct for this grammar and it is built up according to the expansions in Figure 3.

Une image contenant texte, carte

Description générée avec un niveau de confiance élevé

Figure 3

## Log DSL



### Grammar of the log language

The first task was to propose a grammar to define the language of log file. To do so, an EBNF made of 8 expansions describes the entire log file language. The start symbol is <Log-Program>. 8 rules are set. Non-terminals and terminals symbols are present within those 8 rules.

<Log-Program> ::= “*begin”* <Body> “*end”* (1)

<Body> ::= [<Line> “*;”*]+ (2)

<Line> ::= [<Timestamp> “:” <CallCommand>] (3)

<Timestamp> ::= <Integer> <Unit> (4)

<Unit> ::= “*sec”* | “*msec”* (5)

<CallCommand> ::= <Identifier> <Identifier> “(“ <Actual-parameter> “)” (6)

<Actual-Parameter> ::= <Parameter> [ “,” <Parameter>]\* | Ø (7)

<Parameter> ::= “[[“ UnicodeCharacter\* “]]” | [0-9]\* (8)

The grammar details how each line must be formed. A line corresponds to a particular event. The purpose of timestamp is to mark when the event happened. A line must contain an Identifier referring to an object and another Identifier referring to an action. Each action is specific to an object. In addition, parameters are given to the action. Parameters is either a string formed by ‘[[‘ and ‘]]’ or an identifier (in order to be able to pass Boolean, reals, integers as parameters). An example of log file is written down and its corresponding graph is drawn (Fig.4). ‘robot1’ and ‘robot2’ stands for two objects while ‘move’ and ‘rotate’ stands for two actions.

begin

2 sec : robot1 move (10, 25);

10 sec : robot2 rotate (45);

Une image contenant texte, carte

Description générée avec un niveau de confiance élevéend

Figure 4

### Lexicon of the log language

The lexicon the log language is how the different tokens must be written. It consists of an EBNF too. In this case, two rules are needed:

Token ::= “[[“UnicodeCharacter “]]” | [“-“]{0,1} [0-9]+ | Letter [Letter|[0-9]|\_]\* | ‘msec’ | ‘sec’ | “begin” | “end” | “:” | “;” | “,” | “(“ | “)” (1)

Separator ::= Space | End\_of\_line (2)

## Object DSL

### Grammar of the object language

The second Domain Specific Language to be written was the object file language. An EBNF made of 7 rules defines completely the language. The start symbol is <Object-Program>. The 8 rules contain the non-terminals and terminals symbols constituting the grammar structure.

<Object-Program> ::= [<Command> “*;”*]+ (1)

<Command> ::= <Description-Command> | <Animation-Command> (2)

<Description-Command> ::= “*Description”* <Identifier> <Instructions> (3)

<Animation-Command> ::= “*Animation**”* <Identifier> *(* <Formal-Parameter> *)* <Instructions> (4)

<Instructions> ::= “*{{”* *UnicodeCharacter*\* “*}}”* (5)

<Formal-Parameter> ::= <Identifier> [“,*”* <Identifier>]\* | Ø (6)

<Identifier> ::= [a-zA-Z] [a-zA-Z0-9\_]\* (7)

An object file is composed of object descriptions and definitions of the actions. A definition of an action may either have parameters or not. An example of object file is written down and its corresponding graph is drawn (Fig.5).

Description robot1 {{ robot1 = rectangle(red, large) }}

Animation robot1 move (x,y) {{ robot1.set\_position(#x, #y) }}

Une image contenant texte, carte

Description générée avec un niveau de confiance très élevé

Figure 5

### Lexicon of the object language

The lexicon the log language is how the different tokens must be written. It consists of an EBNF too. In this case, two rules are needed:

Token ::= “[[“UnicodeCharacter “]]” | [“-“]{0,1} [0-9]+ | Letter [Letter|[0-9]|\_]\* | ‘msec’ | ‘sec’ | “begin” | “end” | “:” | “;” | “,” | “(“ | “)” (1)

Separator ::= Space | End\_of\_line (2)

## Semantic rules

The semantic rules of a programming language describes in a more or less rigorous way the actions that can be done by this language. Most programming languages provide a standard that clearly indicates how the language works. Explanations are hence done to clearly understand the way Animo processes.

A log file may contain tons of lines for several objects related to different actions. This log file will be generated automatically by a system. The user will then have to program the object file according to his wishes.

- The user will write down the way objects will be described and/or animated.

- While writing description and animation, the identifier of an object has to be written exactly in the same way

- The identifier of an object must be written exactly in the same way in both log file and object file.

- Only object identifiers present in the object file will be considered during the code generation phase. The objects which are written in the log file but not in the object file will be ignored.

- Given an object, the user specifies how he wants to animate events. Each animation is peculiar to an object.

- The user may reference the time when an event happens in the instructions by writing ‘#t’. Consequently, ‘t’ cannot be used as a parameter. The given time is always in millisecond.

- When writing instructions for an animation, to insert the parameters, the users must add a ‘#’ in front of it. For instance, if an animation is set as in the expression (1 and there is the event of the expression (2) in the log file, Animo will replace the #x by ‘2500’; similarly for the expression (3), #x will be replaced by “string”.

‘Animation object\_ID action\_ID (x) {{ …. #x… }}’ (1)

‘3 sec : object\_ID action\_ID(2500); (2)

‘5 sec : object\_ID action\_ID([[string]]); (3)

* Inside instructions, the user may have access to the current time by writing #t such as for parmeters.
* The time must be either constant or increasing

Une image contenant capture d’écran

Description générée avec un niveau de confiance très élevé

Figure 6

Fig.6 containing a log file and an object file is an illustration of those rules. In the log file, many objects appear (robot1, light and robot2). However, since only robot1 is mentioned in the object file, the events for ‘light’ and ‘robot2’ are ignored. Code is generated for the description of ‘robot1’ and for each event when ‘robot1’ has an action of ‘move’ or ‘rotate’.

The output of Animo is code that the user will specify while writing the instructions. Actually, those instructions are templates. Template is code or text in which just few parameters are changed at each time.

# Compiler

## Compiler theory

### Compiler history

Computer architecture and computer applications have always been evolving. The compilation, being on the borderline of these two evolving worlds, has naturally evolved as well.

Schematically we could cut the story of the compilation into four main periods:

1950-1970: During this period, the main concepts of hardware architecture were in place, but it was difficult to program computers which were bulky machines. Programs were written in assembly language which is absolutely not readable nor easy to use. Regular expressions which use the theory of finite automata and the syntactic analyzers allowed the creation of tools to create programming languages ​​and their compilation. FORTRAN[[5]](#endnote-5) (1957) was the first high-level computer language ​​compiled. A high-level language is close to the English language and getting familiar with it is easier than with assembly language. This period corresponds to the creation of the theoretical and practical foundations of the compilation.

1970-1990: Throughout those 2 decades, consumer electronics was popularized. The Intel 4004 processor (1971) is the first mainstream processor. The MIPS processor was created (1984), it introduced the concept of RISC architecture. It was also during this period that Gordon Moore reevaluated his famous law stating, “the number of microprocessor transistors on a silicon chip doubles every two years”[[6]](#endnote-6). The well-known C (1972) and C ++ (1983) languages ​​were created and the first open source compiler GCC 1.0 was released. This compiler had a huge success and received the support of many companies (IBM…). The objectives of the compilation of a program was: “**to provide a binary program semantically equivalent to the source program**”.

1990-2000: This period was extremely rich with the creation of new hardware concepts: pipelines, memory caches, parallel machine. The Java language[[7]](#endnote-7) (1995) was created at that time. The objective of the compilation of a program has become: “**to provide a semantically equivalent binary program that makes the best use of hardware concepts**”. The optimization phases have become more and more complex. The first tools for parallel programming such as MPI for the passage of messages between processors (1991) or OpenMP for automatic parallelization (1997) were created.

2000-2019: Lately, the complexity of architectures has skyrocketed: multicore processors, graphic processors (GPU). The languages ​​CUDA (2007) and OpenCL (2009) have emerged to take advantage of graphic processors. The open source compiler LLVM appeared in 2003 with the support of Google and Apple. The current objective of the compilation is: “**to provide several binary programs optimized to exploit the different cores of the computer**”.

### Operating principles of a compiler

A program for a processor must be built by combining commands into a program in the machine language. This language is tedious and error-prone process. In fact, it is not readable by human being. The machine code is composed of 0 and 1 only. Consequently, most programming is done using a high-level programming language. This language is usually very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler takes action. Generally, compiler translates (or compiles) a program written in a high-level programming language that is suitable for human into the low-level machine language. Some compilers translate into another programming language, such as Animo. During this process, the compiler also detects and reports programmer mistakes.

Building a compiler is a tough task, the work is instead split into sub-tasks. A common way of doing this is to divide the compilation into many phases. These phases operate in sequence, each phase taking the output from the previous phase as its entry. Each phase is handled by a separate module. The front-end phase is composed of a lexical analysis, syntactic analysis, contextual analysis and intermediate code-generation. The back-end phase contains 2 processes: optimization and code generation. There may be additional phases, depending on compilers. A typical flow of compiling process is presented in the figure 3[[8]](#endnote-8).

A grammar is needed to build up the entire front-end phase. An Extended Backus-Naur Form (EBNF) is traditionally used to express the specificity of the language (such as we did for our 2 DSL).

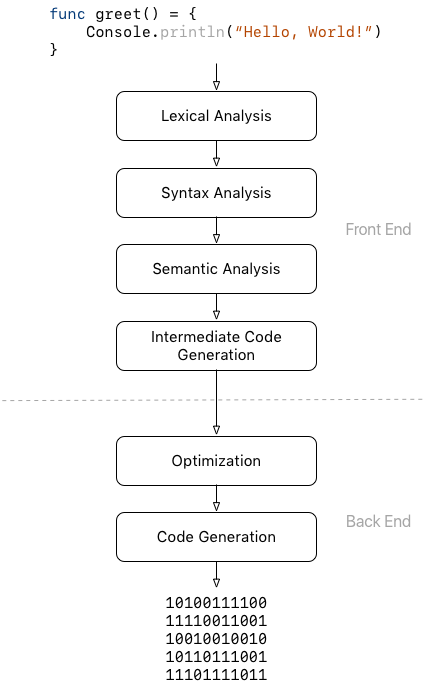


Figure 7 Compiling flow

## Compiling process

### Lexical analysis

The word “lexical” means, in terms of programming languages, objects like variable names, numbers, keywords etc. Such words are called tokens. A lexical analyzer, lexer for short or scanner, reads the input source program from left to right, one character at a time, and generates a stream of tokens associated with their value. Each token is a unit such as identifier, operator, integer… Additionally, it filters out whatever separates the tokens: white-space, new-line, comments … Actuallay, the lexer scans a token whenever the parser requires it to provide the next token.

The table indicates the result of the scan of the “sentence”:

Animation robot1 move (x,y) {{ robot1.set\_position(#x, #y) }}.

|  |  |
| --- | --- |
| Token kind | Token value |
| Animation | Animation |
| Identifier | robot1 |
| Identifier | move |
| Lparenthesis | ( |
| Identifier | x |
| Comma | , |
| Identifier | y |
| Rparenthesis | ) |
| Instructions | {{ robot1.set\_position(#x, #y) }} |

### Syntactic analysis

Syntax analyzer, so-called parser, takes as input the stream of tokens generated by the scanner to join them in order to form valid “sentence” regarding our context-free grammar. By combining sentences, the parser generates a data structure which represents the entire code structure: the abstract syntax tree. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree and how the interior nodes of the tree are labelled. In addition to finding the structure of the source program, the syntax analysis rejects invalid programs by reporting syntax errors, sentences which do not fit the context-free grammar. Our context-free grammar is specified by the EBNF described in 2.1. There are many kinds of parser, depending on how to parse the program. Parsers are categorized into two groups: parser which parses from top to bottom (called top-down parser) such as LL(1) and parser which parses from the bottom to the top (called bottom-up) such as LALR and LR(1). Our compiler implements an LL(1) syntactic analyzer which stand for Left-to-right, Leftmost derivation.

The trees of the 2.2 and 2.3 are the results of the parsing phase according to the given sentences.

### Contextual analysis

Given an abstract syntax tree, the purpose of contextual analysis is to check that the program conforms to the source language’s semantic and contextual constraints. This phase consists mainly of verifying 2 rules: scope rules and type rules. Scope rules determine the validity of variables, functions, objects and so on. Some may be local with a specific life period whereas some may be valid throughout the whole program. Type rules specify the expected type of expressions and to decide whether each expression has a valid type. Consequently, two subphases emerge: identification to apply scope rules to relate each applied occurrence of an identifier (variables, function, objects and so on) to its declaration (if any) and type checking to check the expected type of an expression with its actual type.

A symbol table (or identification table) is used to associate each identifier with their attributes. For example, the variable ‘var’ and its value ‘4’ are inserted in this table. The symbol table handles the identification part since whenever an identifier is seen, the compiler checks the table if the identifier is declared and either yields its value or raise an error if the identifier is not declared. This table is also used to do the type-checking. The Abstract Syntax tree is decorated of information such as type of variables, type of expressions….

There are 3 blocks structures: monolithic, flat and nested. Monolithic structure corresponds to one block which is the entire program. All declarations are global. Flat structure may contain two levels of blocks, either global or local. Nested structure when blocks may be nested within another. Fig.7 exposes the different structures. Animo implements monolithic block structure since everything (object identifiers, functions…) is global.

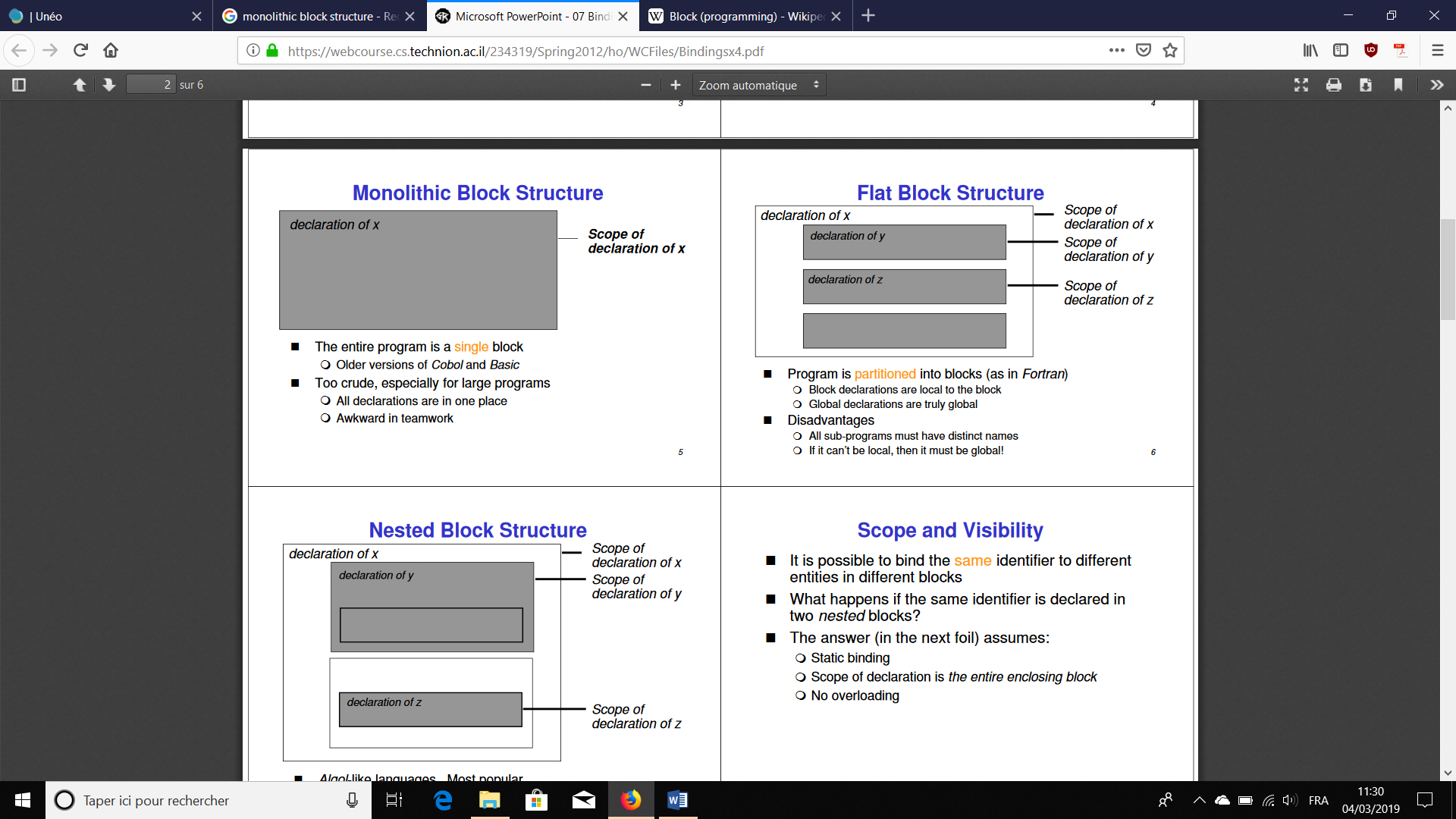
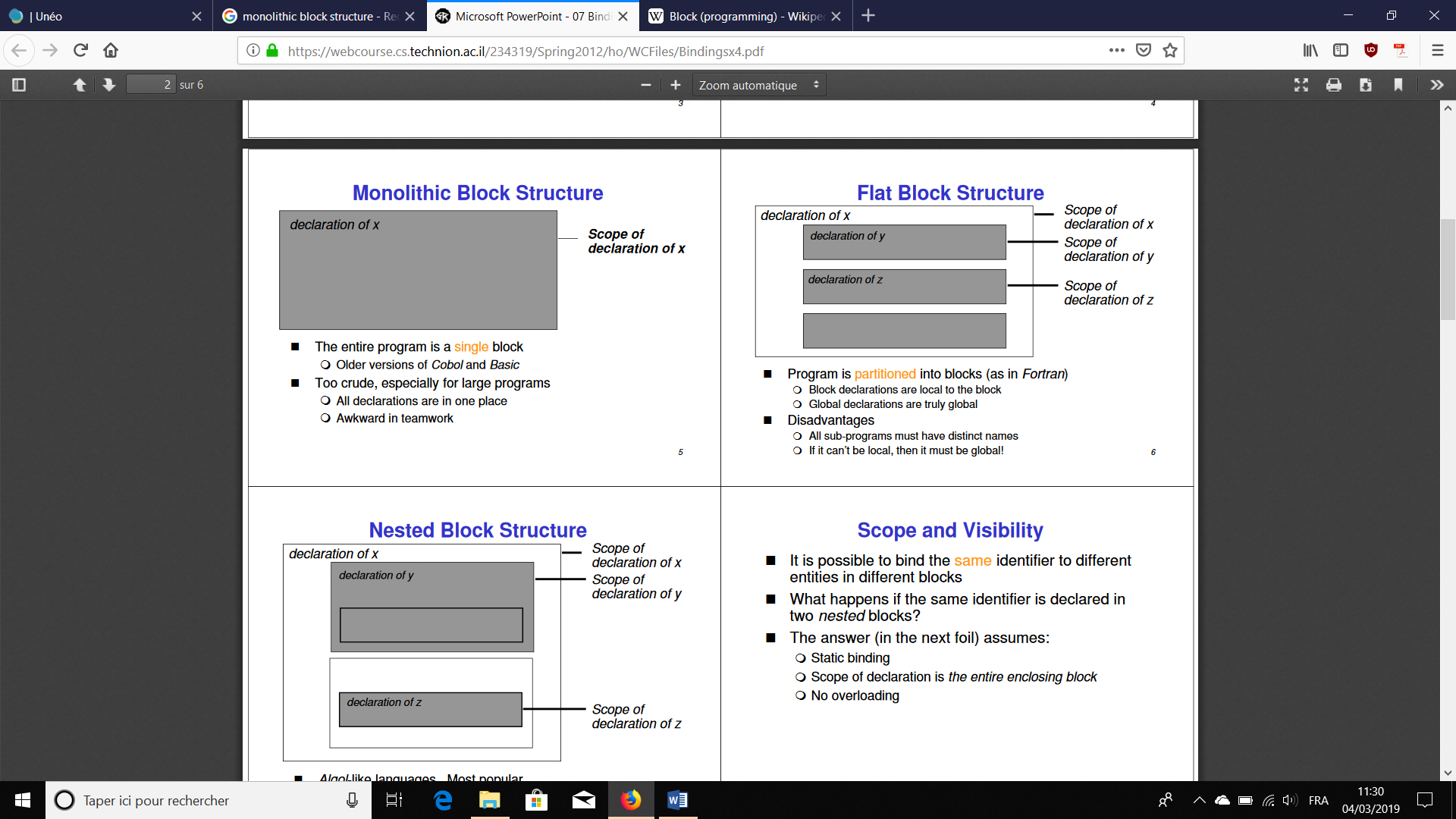
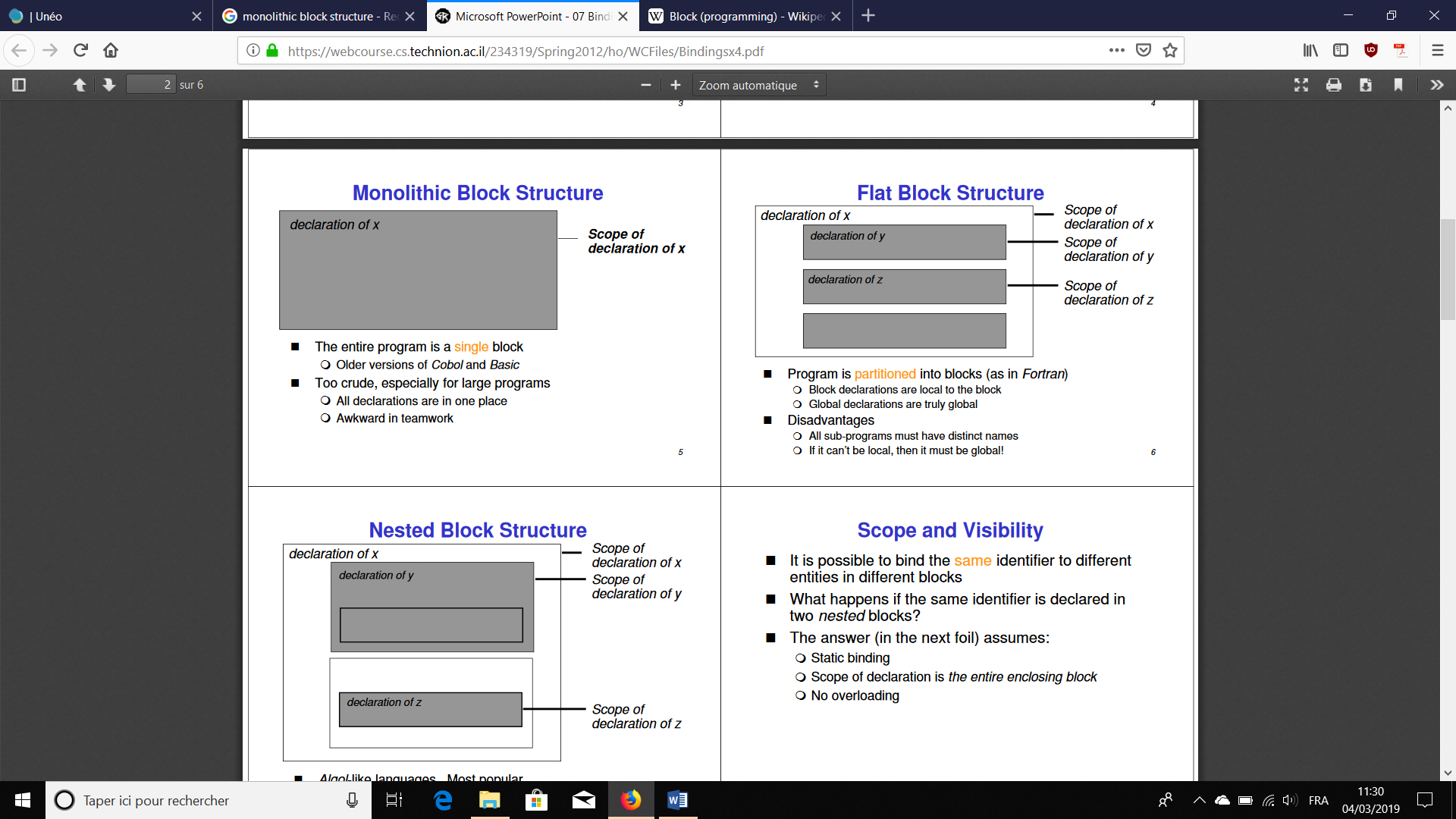


Figure 8 Block structures

### Code generation

Given a decorated abstract syntax tree, the purpose of this phase is to generate the target code. The final goal of a compiler is to transforms code written in a language (Log file language and Object file language) into another code. Mostly, the target code is machine code. This means that, eventually, the program will have to be expressed as machine code which can run on the processor. Many compilers use a medium-level language as a stepping-stone between the high-level language and the very low-level machine code. Such stepping-stone languages are called intermediate code. However, as mentioned in 1.3.2, the target code of Animo is not machine code but is plain text through templates. Consequently, intermediate representation is not implemented for Animo.

## Implementation

The work has been divided into two parts: modules to parse log files and modules to parse object files have been written. A common class called ‘AST.rb’ holds the root node ‘Program’. The *main* program is ‘compiler.rb’, it handles the entire compilation by firstly parsing the object files then parsing the log files. It proceeds then to the contextual analysis trough the class ‘checker.rb’. At last ‘CodeGenerator.rb’ is called to generate output.

### Implementation of a scanner

The purpose of the scanner is to identify tokens in the source file (log and object file). The scanner will iterate over the source file, one character at a time, to group them to form a token. The source file contains might contain separators such as blank space or new line. Their purpose is to separate tokens and to make the reading easier. Only tokens contribute to the program.

A scanner can systematically be developed by applying the following method.

1. Specify the lexicon of the language in EBNF (see 1.2.2 and 1.3.2). The terminal symbols are the characters while the non-terminal symbols are the tokens.
2. Transcribe each EBNF expansion A ::= B to a scanning method ‘scan\_A’, whose body is determined by B.
3. Provide the scanner a variable ‘current\_char’, two methods ‘take’ and ‘takeIt’ (explained later), the scanning methods of step (2) and a public ‘scan’ method accessible from another class which returns the next token in the source file.

On entry to each ‘scan\_A’ method, ‘current\_char’ is supposed to contain the first character of a character sequence of kind A. When exiting ‘scan\_A’, ‘current\_char’ is supposed to hold the character immediately after that character sequence. The method ‘takeIt’ will fetch the next character from the source file and store it in ‘current\_char’. The method ‘take’, which takes a character as argument, will do so only if its argument matches ‘current\_char’. This method is useful to ensure that the next character is the expected one. The ‘scan’ method fetches the next token from the source file by ignoring separators. Whenever the parser requires a token, it will call the method ‘scan’.

### Implementation of a parser

The purpose of the parser is to identify sentences in the source file. The parser will iterate over the source file, one token at a time, to group them to form sentences. The parser is analogous to scanner but at a higher level. In scanning the terminal symbols are characters while in parsing terminals symbols are tokens. The construction of a parser is very alike the construction of a scanner. It might be developed by following the following rules:

1. Express the grammar of the language in EBNF (see 1.2.1 and 1.3.1). The terminal symbols are the tokens while the non-terminal symbols are the categories of the grammar.
2. Transcribe each EBNF expansion A ::= B to a scanning method ‘parse\_A’, whose body is determined by B.
3. Provide the scanner a variable ‘current\_token’, two methods ‘accept’ and ‘acceptIt’, the parsing methods of step (2) and a public ‘parse\_S’ (with S the start symbol) method accessible from another class which parses the input file and returns the abstract syntax tree.

On entry to each ‘parse\_A’ method, ‘current\_token’ is supposed to contain the first token of a token sequence of kind A. When exiting ‘parse\_A’, ‘current\_token’ is supposed to hold the token immediately after that token sequence. The method ‘acceptIt’ will fetch the next token from the source file (by calling the ‘scan’ method) and store it in ‘current\_token’. The method ‘acccept’, which takes a token as argument, will do so only if its argument matches ‘current\_token’.

For instance, the method ‘parse\_log\_program’ is defined as follow:

def parse\_log\_program():

accept(‘begin’)

parse\_body()

accept(‘end’)

Animo implements a LL(1) recursive-descent parser. The parser processes from the top to the bottom and parses from left to right. The 1 means that the parser needs only one token to unambiguous.

### Implementation of an abstract syntax tree

A recursive-descent parser determines the structure of the source program implicitly, in the sense that it finds the beginning and the end of each ‘sentence’. In a multi-pass compiler (a compiler which browses the program many times), the parser must construct an explicit representation of the structure of the source program. The best-known and most used one is the abstract syntax tree.

An abstract syntax tree is formed of node, children and leaves to represent the whole structure of the program. A node is a non-terminal symbol of the grammar, a child is either a non-terminal symbol or a terminal symbol and a leaf is a terminal symbol. Concatenating each leaf makes emerge the source file (without the separators).

It is straightforward to make a recursive-descent parser construct an AST. The following rules have to be implemented:

1. Write classes for each non-terminal symbols of the grammar. Each class will have instance variables for each non-terminal in its expansion. For a rule like ‘A ::= BC’, the class A will have B and C as instance variables. Those variables will in turn be AST.
2. Each ‘parse\_A’ method will return the AST of A representing its structure.

Figure.4 and figure.5 in section 1.2 and 1.3 are two abstract syntax trees representing the structure of the related input program.

### Visitor pattern

The visitor design pattern is a way of separating an algorithm from an object structure. The idea is that you have a set of element classes that form an object structure. Each of these classes has a “accept” method that takes a visitor object as an argument. The visitor is an interface that has a different “visit” method (“visit\_program” for instance) for each class. The “accept” method of a class calls back the "visit" method for its class. Separate concrete visitor classes can then be written that perform some particular operations.

Concretely, visitor pattern is implemented in the AST class to firstly process the contextual analysis and secondly, to generate output through the code generation phase. Hence each class has a method ‘accept’ which calls the method ‘visit\_class’ to do actions. For instance, the AST node timestamp has the following method:

def accept(visitor):

return visitor.visit\_timestamp(self)

‘self’ refers to the AST node so that the method ‘visit\_timestamp’ could do actions with this node.

### Implementation of the contextual analyzer

The contextual analyzer is implemented as a visitor object that performs identification and type checking. Each visitor method ‘visit\_node’ in the contextual analyzer will check the node, raise an error if it determines that the ‘sentence’ represented by the node is ill-formed.

The purpose of the contextual analyzer of Animo consists mainly in identification since there is no type in the log and object languages. Symbol table is built up in this phase. Symbol table is a table which holds information for each identifier. An identifier has only one entry. The symbol table of Animo has object\_identifier as entry. The value of each object\_identifer is a table containing information about each action of this object.

1. A number representing the amount of formal parameters
2. An array containing the identifier of each formal parameter
3. The instructions of the action

|  |  |
| --- | --- |
| **Object\_identifier** | **Action\_table** |
| robot1 | move = 2, [x,y], ‘robot1.set\_position(#x,#y).at(#t);’  rotate = 1, [angle], ‘robot1.turn(#angle).at(#t);’ |
| light | on = 0, [], ‘light.turn\_on();’ |

In the table, the robot1 has two actions (‘move’ and ‘rotate’). ‘move’ expects two parameters while ‘rotate’ expects one.

When *visiting* the object tree, Animo fills in this table. When *visiting* the log tree, for each line it verifies if the object\_identifier is in the symbol table. If it is not, Animo will skip this line. If the object is in this table Animo will then checks if the action is in the table then checks the number of parameters. If the number of given parameters is less or more than the expected one, Animo will raise an error. In addition, the contextual analyzer of Animo checks that the time is increasing as well.

### Implementation of code generator

The implementation of the code generator is similar to the implementation of the contextual analyzer. It is a visitor object too. The code generator will generate code for some specific nodes. Typically, when visiting a ‘description’ in the object file, it will immediately generate the instructions of this description in the output file. When visiting the log file, for each event when the action is specified in the object file by the user, Animo will write in the output file the corresponding instructions. The parameters of the instructions will be replaced by the actual parameters of the log file. For instance, assume the following object file:

Animation robot1 move (x,y) {{ robot1.set\_position(x,y).at(#t); }}

Now assumer that in the log file, the following line is written:

3 msec: robot1 move (4, 5);

The code generator will write in the output file:

robot1.set\_position(4,5).at(3);

# Results

The objectives are accomplished. As usual, some bugs will appear of course. And they would be fixed then. The two domain specific languages have been specified and the compiler Animo has been developed. The first language defines how log files must be written to be parsed and analyzed by Animo. The second language defines how object files must be written to let the user represents the objects and the animations of events as he wishes. Concretely, given a huge log file, the user may choose which events to animate and how to animate them according to the situation (parameters and time).

The output of a compilation is code (text) depending on how descriptions and animations are written in the object file. The output has to be then inserted in another source code to obtain a possible animation. The output is not an animation. A continuation of Animo could be to integrate a simulation engine to execute an animation after compiling code.

# Alternative solution

Previously to the idea presented, another solution was being developed. The idea remains essentially the same however the way to address the animation is different. The current solution proposes the user to generate code according to the instructions made in the ‘object file’. While, the first idea consists of multiple already made animations. The user has to choose among them. In concrete terms, the compiler contains a large amount of available animations for different type of actions. The user assigns for each object of his log file a type of animation and provide an image (SVG file) representing graphically the objects. Then Animo compiles and the user obtains JavaScript and HTML files which contain the animation as result. For example, assume that the log file containing such lines is provided as input:

1s: robot1 move(x1, y1);

2s: robot1 move(x2, y2);

…

The user writes then in an object file: “robot1 = move, ‘robot\_image.svg’;”. The compiler will tie each occurrence of ‘robot1’ to its representation ‘robot\_image.svg’, it will then generate code in JavaScript with the SVG.js library corresponding to the ‘move’ animation. That is: ‘robot1.moveTo(X, angle)’ with X the distance to cross and ‘angle’ the angle to rotate.

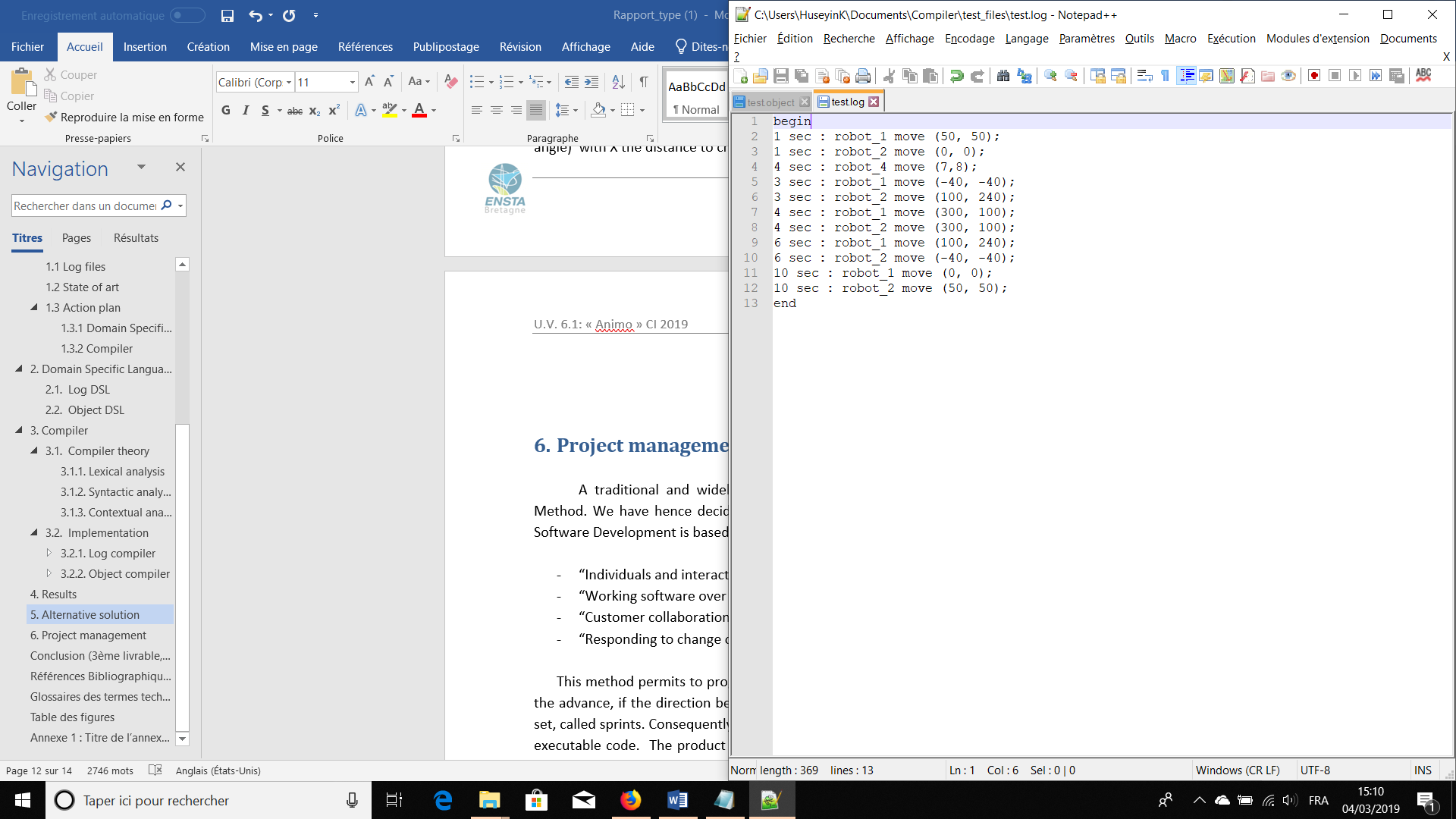
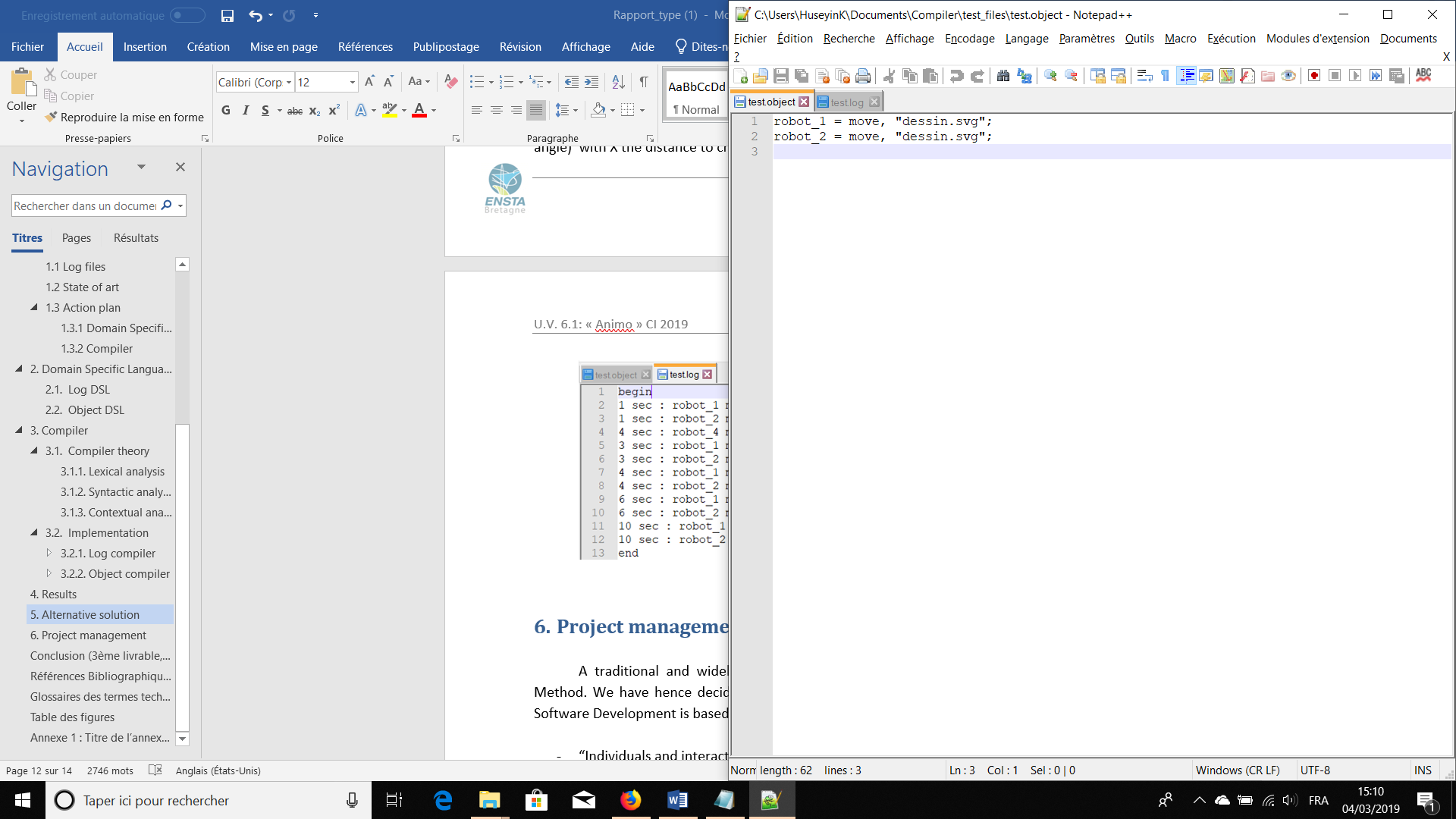
A complete example of a log file is provided in figure 5 and 6. The alternative compiler analyzes the two files, scans, parses, generates the ASTs, checks the types then generate the corresponding files for the animation (see annex 1). The animation is available at the address: <https://www.youtube.com/watch?v=6IfCPAS60ys>

Figure 9 Log file from the alternative solution

Figure 10 Object file from the alternative solution

When this solution was developed, something scalable was intended to build. By this way, new kind of animation like ‘move’ or ‘chart’ would be added gradually. For example, ‘chart2D’ was being added which would draw the evolution of the given input in the time. A typical log line would be: “20s: ID\_chart plot(x)” and a typical object description would be: “ID\_chart = chart;”.

The strength of this method is its simplicity and its ease of access. In addition, it is highly scalable to add new animations. Its weakness is the user cannot animate how he wishes and has to fit to a certain kind of animation.

# Project management

A traditional and widely popular method of software project management is the Agile Method. The Agile Software Development is based on 4 principles[[9]](#endnote-9):

- “Individuals and interactions over processes and tools”

- “Working software over comprehensive documentation”

- “Customer collaboration over contract negotiation”

- “Responding to change over following a plan”

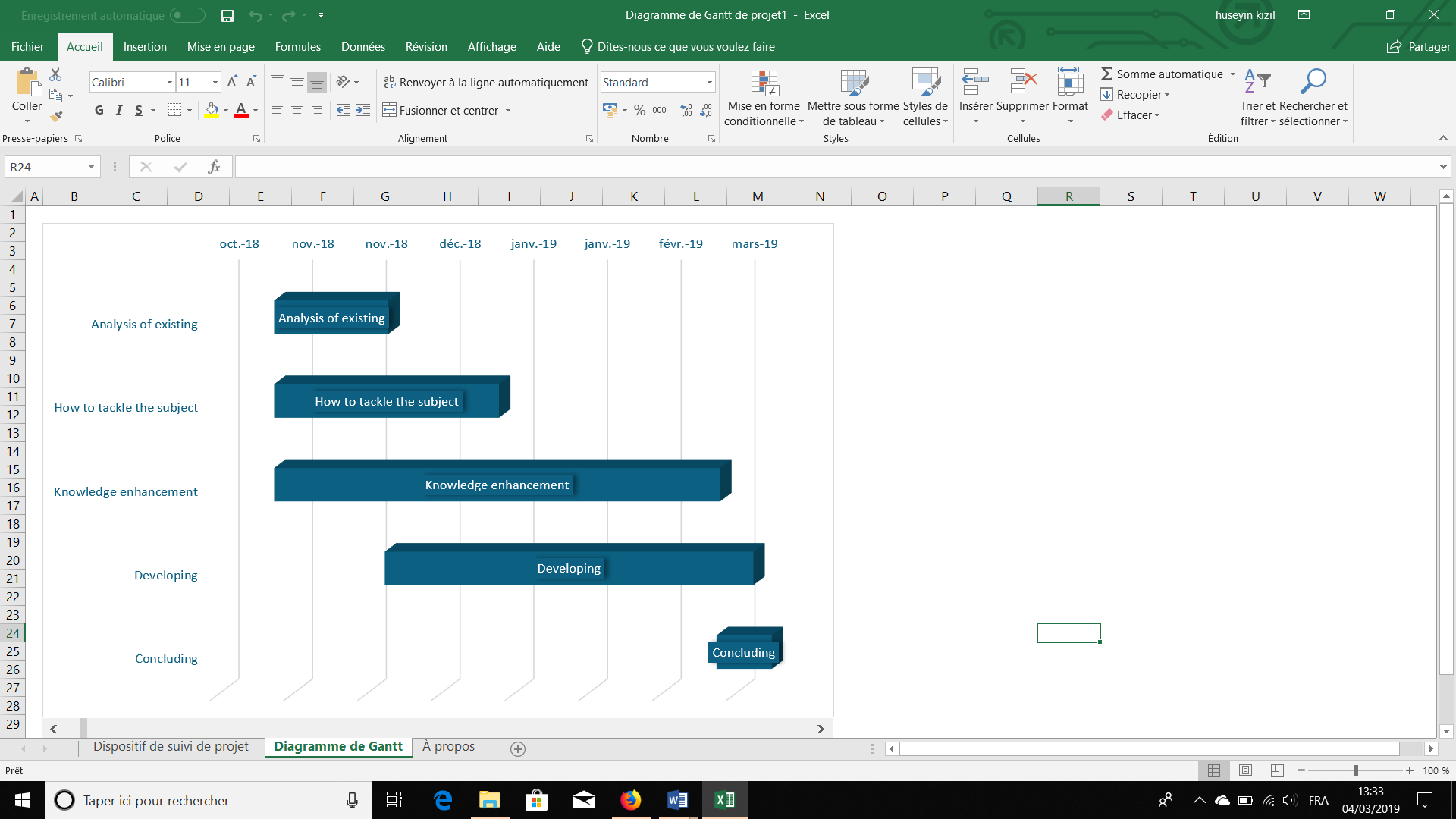
The progress of the project is constantly tracked using this tool and corrective actions can be made immediately if necessary. Short-term objectives are set, called sprints. The progression of the project occurred mainly from the February. Progress meeting are often held to verify the development. In this way, the product is progressively enhanced. A Gantt diagram which shows the overall conduct of the project is drawn (Fig.11).

Figure 11 Gantt diagram

Since the beginning of this project, some directions have been taken. Those directions illustrate the methods which were thought out to be implemented to provide a possible solution. Concretely, 2 main ideas have been developed. The one which is presented in this report and the alternative one. A succinct presentation of the second idea is described in section 5.

The strength of the Agile method is illustrated in this project. Indeed, adaptations have been made by changing quickly the work to fit neatly with the expectations of the client.

# Conclusion

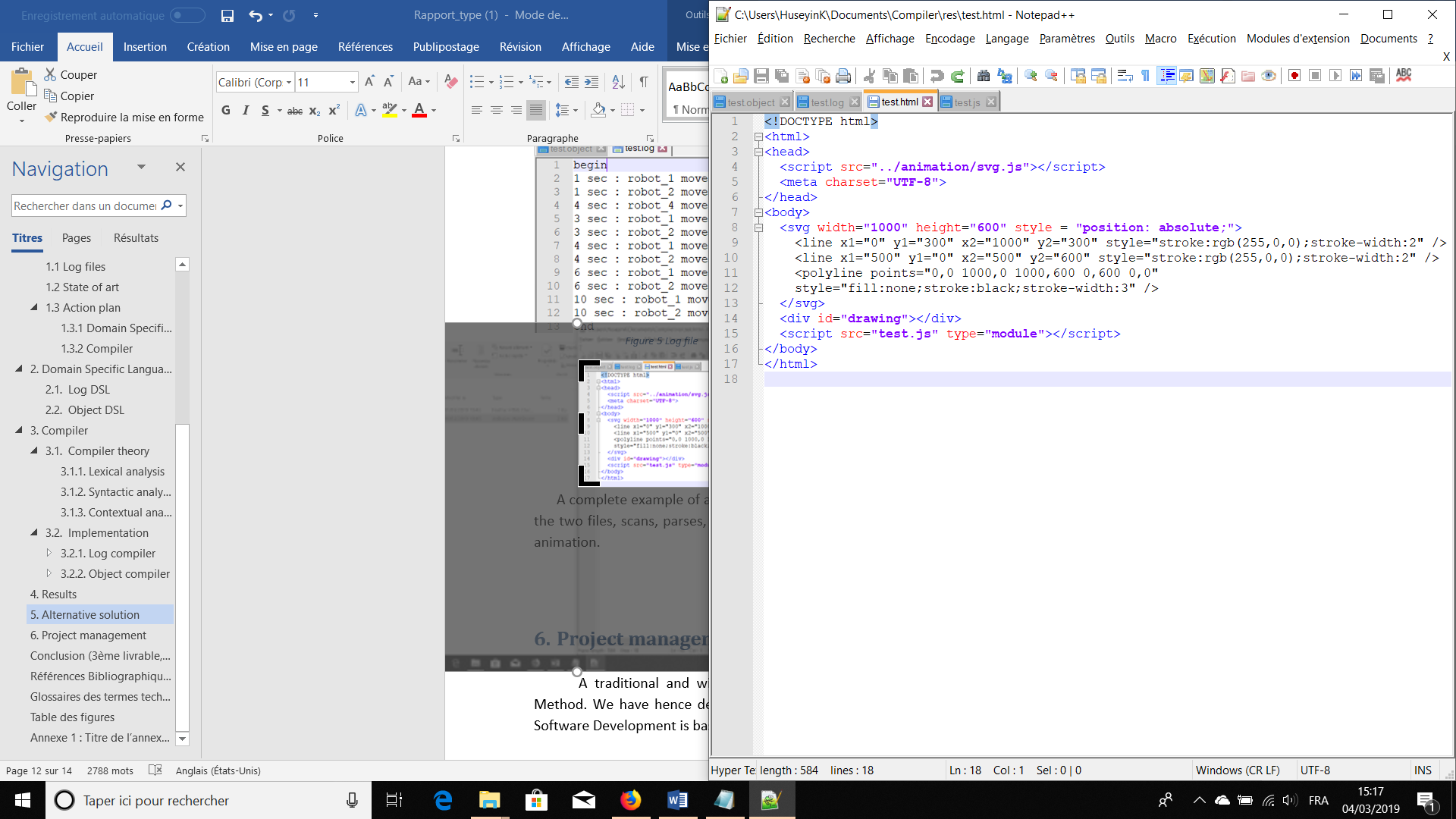
To conclude, the animation of log files is a domain which is under construction. There is no currently existing published log file animator. The solution proposed is to let the user choose the way he wants to animate events. The user firstly indicates in the object file how objects must be presented. Then, the user defines the animation for events of the log file. At last, the compiler Animo analyze the object file and the log file and generates code according to the information provided by the user.

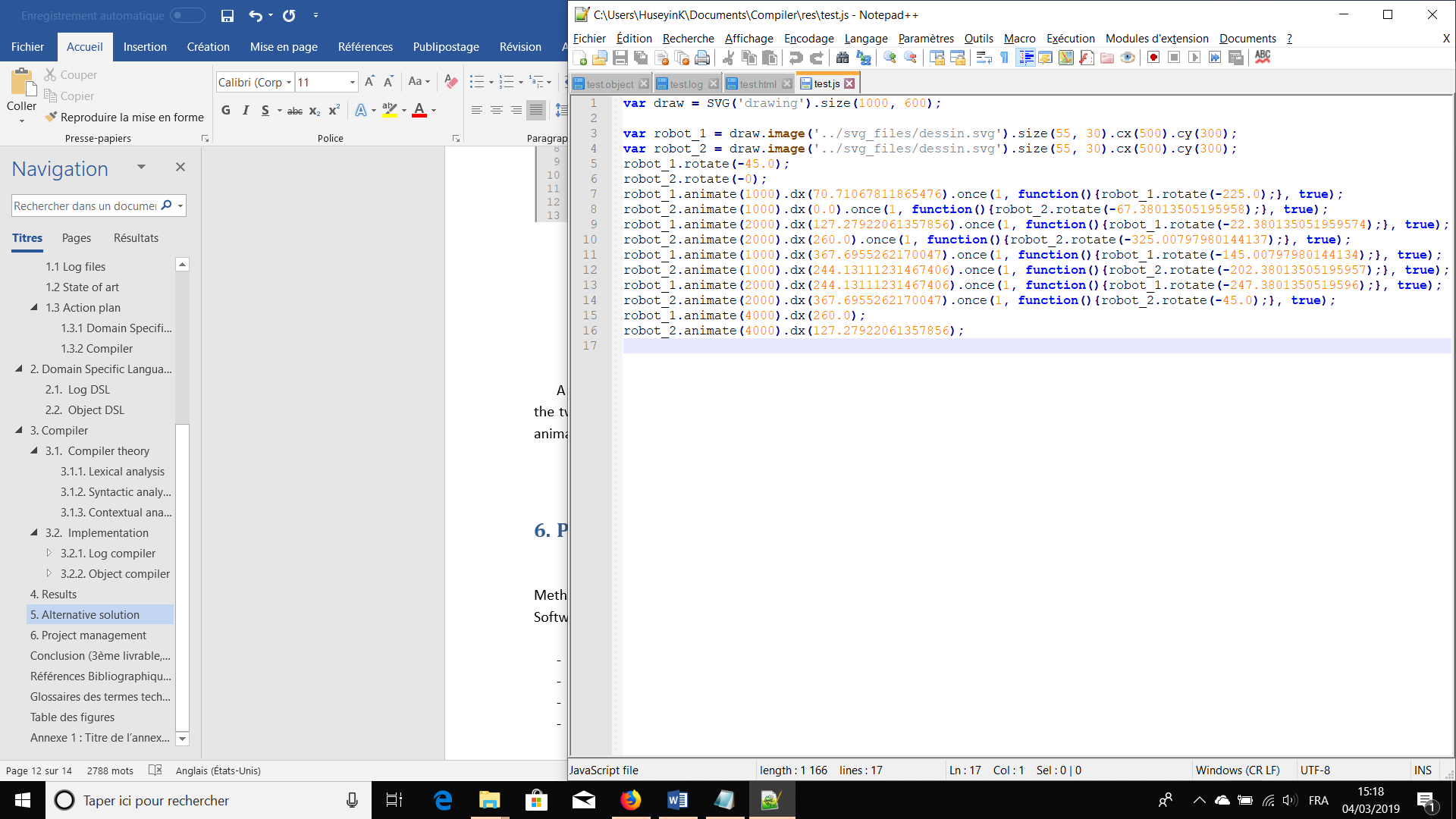
For the moment, the output is text which is formatted as the user wants. This output has to be inserted in another code to be executed. This aspect is interesting because it provides an abstract animation scalable to several languages. For instance, the user might animate with Python, Java, C and so on. A possibility of extension would be to create plugins to existing animation tools so that the user would not need to code.

This project has been an opportunity to learn a new programming language (‘Ruby’). Compiler knowledge have been strengthened. The vision of language and domain specific languages has also changed. This has also been a chance to discover the world of research since this topic is an uncovered one.

# Annex

Annex 1





# Bibliography

1. “What Is a Log File (and How Do I Open One)?” https://www.howtogeek.com/359463/what-is-a-log-file/ (January 30, 2019). [↑](#endnote-ref-1)
2. Portent. (2010, March 25). Retrieved January 30, 2019, from https://www.portent.com/blog/analytics/how-to-read-a-web-site-log-file.htm [↑](#endnote-ref-2)
3. “GoAccess - Visual Web Log Analyzer.” https://goaccess.io/ (January 30, 2019). [↑](#endnote-ref-3)
4. “CsvView/DatCon.” http://www.datfile.net/ (January 30, 2019). [↑](#endnote-ref-4)
5. “History of FORTRAN and FORTRAN II — Software Preservation Group.” http://www.softwarepreservation.org/projects/FORTRAN/ (March 7, 2019). [↑](#endnote-ref-5)
6. “Over 50 Years of Moore’s Law.” Intel. https://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html (March 7, 2019). [↑](#endnote-ref-6)
7. “History of Java Technology.” https://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html (March 7, 2019). [↑](#endnote-ref-7)
8. Tchirou, Faiçal. 2017. “Compilers and Interpreters.” Hacker Noon. https://hackernoon.com/compilers-and-interpreters-3e354a2e41cf (March 3, 2019). [↑](#endnote-ref-8)
9. “Manifesto for Agile Software Development.” https://agilemanifesto.org/ (January 30, 2019). [↑](#endnote-ref-9)