

# Implementation of a compiler for an imperative language IMP

Remy Detobel & Denis Hoornaert

November 11, 2017

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Implementation of the lexical analyser</b>	<b>2</b>
2.1	The use of a lexical analyser generator . . . . .	2
2.2	Regular expressions . . . . .	2
2.3	Hypothesis on regular expressions . . . . .	3
2.4	Dealing with nested comments . . . . .	4
2.5	Tests and results . . . . .	5
<b>3</b>	<b>Implementation of the syntax analyser</b>	<b>5</b>
3.1	Transforming the grammar to LL(1) grammar . . . . .	7
3.2	Results of simplifications on the IMP grammar . . . . .	7
<b>4</b>	<b>How to set up the project</b>	<b>7</b>
4.1	Compilation . . . . .	7
4.2	Execution . . . . .	7
4.3	Test . . . . .	7
4.4	Javadoc . . . . .	8
<b>5</b>	<b>Annexe</b>	<b>8</b>
5.1	Left factorisation using strees . . . . .	8

## 1 Introduction

The aim of project is to implement a compiler for a 'simple' imperative language named *IMP*. Like any imperative programming language, *IMP* is composed of mainstream features such as *keywords* (*if*, *while*, ... statements), *variables*, *numbers* and *comments*. The form of these features follows some defined rules :

- a *variable* is a sequence of alphanumeric characters that must start by a letter.
- a *number* is a sequence of one or more digits.
- a *comment* must start by the combination '*\**' and ends by the reversed combination '*\**'.

The compilation scheme is generally divided in three main phases : analysis, synthesis and optimisation. The phases are themselves composed of different steps. For instance, the analysis phase is composed of *lexical analysing* step (or *scanning*), a *syntax analysing* step (or *parsing*) and a *semantic analysing* step as shown in fig.1. In this assignment, the focus is set on the *analysis phase*.

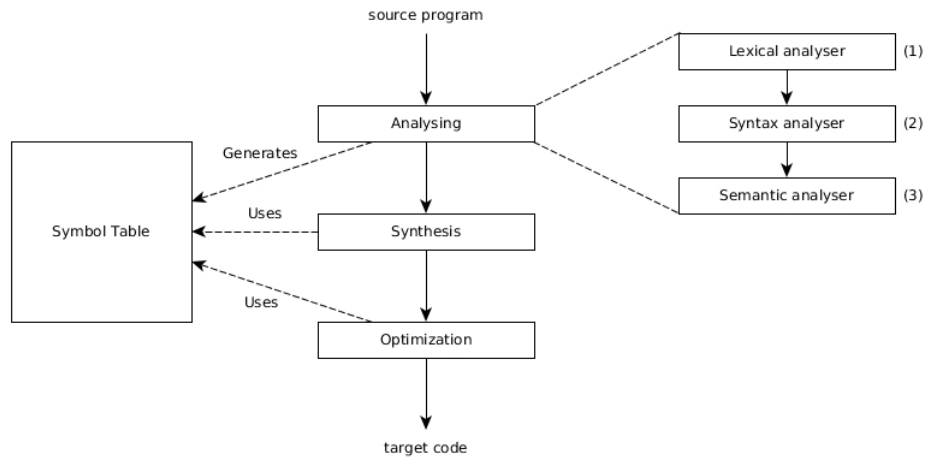


Figure 1: Compilation phases

## 2 Implementation of the lexical analyser

In the so called "Dragon book"<sup>1</sup> the *lexical analyser* is defined as follow :

«The *lexical analyser* reads the stream of characters making up the source program and groups the characters into a meaningful sequence called *lexemes*.»

A *lexeme* can be defined as a tuple which contains both a *token name* and the associated value (not always mandatory). The sequence of *lexemes* generated by the *lexical analyser* will be used by the following step. In addition, the *lexical analyser* will generate a very useful tool used during all the other steps (as shown in fig.1 .) and called a *symbol table*. The role of the *symbol table* is to store every variable encountered while scanning the source code and the line where it appears for the first time.

### 2.1 The use of a lexical analyser generator

In order to ease the process of recognizing the lexemes defined in the given `LexicalUnits.java` file many *lexical analysers* have been developed. Among them, the most well known generator is the flex program and all its derived versions. In the present project, `jflex` is used as it has been decided to implement the project using the java programming language. Using a *lexical analyser generator* eases the analysis of any input because it enables the programmers to describe every *regular expression* by using the *Regex* writing convention and then to generate a `.java` file that will recognise all of them. This generated `.java` file can then be used as any other java class.

### 2.2 Regular expressions

Based on the content of `LexicalUnits.java`, we can easily divide the set of lexical units into two distinct groups : the *keyword* group and the variable/constant group.

The implementation of the *keyword* group using regular expressions is pretty straightforward as simply writing the *keyword* is sufficient. For instance, the regular expression of the *keyword* `if` is simply `if`. On the other hand, the implementation of the variable/constant group requires slightly more work. This small group is composed of two elements the variables and the numbers.

The structure of *variables* given in the assignment statements is "a sequence of alphanumeric characters that must start by a letter". Thus, the equivalent regular expression is :

`[a-zA-Z][a-zA-Z0-9]*`

<sup>1</sup>V. Aho, A., 2007. *Compilers : Principles, techniques, & Tools*. 2nd ed. New York : Pearson.

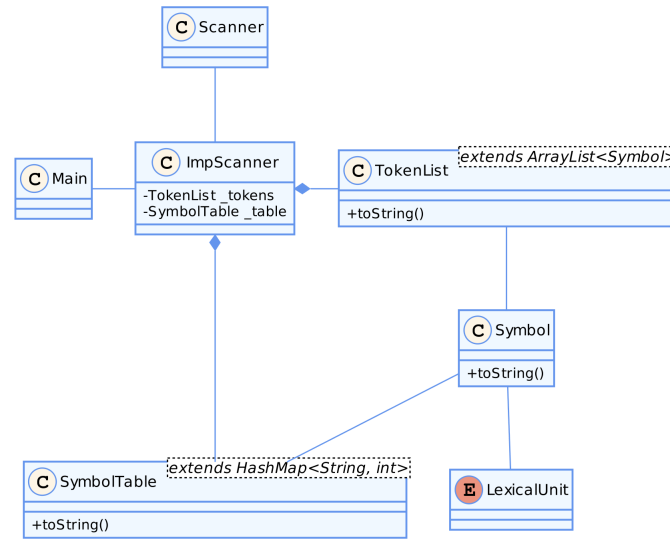


Figure 2: Model class (Pseudo-UML). The TokenList class is the sequence of lexemes and the Scanner class is the file generate by jflex

The structure of *numbers* given in the assignment statements is "a sequence of one or more digits". thus, the equivalent regular expression is :

`[0-9]+`

## 2.3 Hypothesis on regular expressions

The only hypothesis that has been made throughout the realisation of the project concerns the behaviour of the *lexical analyser* when a character not specified in either the structure of a *number*, a *variable* or a *keyword* is encountered. Typically, amongst this set there are the following characters :

`}`, `{`, `_`, `|`, `&`, `[`, `]`, `(`, `)`

In order words, the question is : What does the *lexical analyser* do for the following line :

```
1 index_of_loop := list[x]
```

Four ideas have been considered :

- Considering these characters as a new lexical unit identified by the following regular expression (not exhaustive) :

`SpecialChar = ["}", "{", "_", "|", "&", "[", "]", "(", ")"]`

The example above would be transposed by the *lexical analyser* into the following token list :

token: index	lexical unit: VARNAME
token: _	lexical unit: SPECIALCHAR
token: of	lexical unit: VARNAME
token: _	lexical unit: SPECIALCHAR
token: loop	lexical unit: VARNAME
token: :=	lexical unit: EQUAL
token: [	lexical unit: SPECIALCHAR
token: x	lexical unit: VARNAME
token: ]	lexical unit: SPECIALCHAR

Unfortunately, the assignment statement disallows us to modify the `LexicalUnits.java` file. Consequently, this possibility is not relevant.

- Considering these characters as normal characters. This would mean that they could be part of a *variable* name. Thus, the regular expression for identifying *variables* has to be modify to look like :

```
SpecialChar = ["}", "{", "_", "|", "&", "[", "]", "(", ")"]  
[a-zA-Z|SpecialChar][a-zA-z0-9|SpecialChar]*
```

As a consequence, the token list generated by the *lexical analyser* will behave as follow :

```
token: index_of_loop      lexical unit: VARNAME  
token: :=                 lexical unit: EQUAL  
token: [x]                lexical unit: VARNAME
```

Even though, using characters like `_` in variable name is common in many programming languages, the other characters are generally not used for this purpose. Given this fact and the fact that the assignment statement does not explicitly mention such a possibility, this idea has been overlooked. Moreover, this implies that variable such as `{!^$'#` would be considered as valid even though having such *variables* is not handy.

- Not considering them. In this possibility, we just overlook them like if they were equivalent to a space character. Implementing this idea is quick but does not really make sense because theses characters would then cause many problems in the following steps.
- Throwing an error. This idea consists simply on printing a warning when an unexpected character is encountered and on stopping the program as resuming does not make any sense. Moreover, this behaviour is pretty common in many programming languages. This solution is the one who fits the best the assignment statements. Therefore, this solution has been preferred over the two others.

## 2.4 Dealing with nested comments

The management of comments using regular language is quite simple. Once an opening statement (here : `'(*)`) has been encountered, it overlooks the following characters until it encounters a closing statement (here : `*)'`).

```
1 (*I am a (*nested*) comment*)
```

Unfortunately, applying the same mechanism on a nested comment will result in a ill-formed outcome. Indeed, in the case of the example above, the analyser will overlook the second opening statement (columns 9 & 10) and will stop when it comes across the first closing statement (columns 17 & 18) having for consequence that the third part of the *nested comment* will remain.

To overcome this problem, the analyser must know how many opening statements it came across and how many closing statements it should expect to encounter in order to know whether it is still in a comment.

The most obvious and smartest way to implement it is to use a counter (i.e. a memory) that will be incremented for every opening statement encountered and decreased for every closing statement encountered. However, from a theoretical point of view, by using a memory the language cannot be considered as regular any more. In the present project, it is not a problem and `jflex` allows us to implement such a language.

## 2.5 Tests and results

In this section, the results of the implementation are analysed and tested through three *IMP* source codes : one given in the assignment statements and the two others inspired by algorithms from the Syllabus of Thierry Massart<sup>2</sup>. The aim of testing the *lexical analyser* on these three tests is to ensure that a maximum of the *keywords* and the *variables* are recognized because the set of keywords in the `Euclid.imp` (fig.3) file does not cover every possibilities. This is why these two codes have been chosen. As explained above in the hypothesis subsection 2.3, the program in fig.5 simply stops its execution as it encounters an undefined character at line 2 ('['). Unfortunately, it is difficult to cover all the different statements as finding interesting samples of code that do not use list(s) is hard.

```
1 begin
2   read(a) ;
3   read(b) ;
4   while b <> 0 do
5     c := b ;
6     while a >= b do
7       a := a-b
8     done ;
9     b := a ;
10    a := c
11  done ;
12  print(a)
13 end
```

Figure 3: *IMP* code to compute the gcd of two numbers

```
1 (*
2   Algorithme to calcul Fibonacci
3 *)
4 begin
5   read(n) ;
6   a := 0 ;
7   b := 0 ;
8   for n from 0 to n do (* loop from 0 to n *)
9     tmp := b ;
10    b := a + b ;
11    a := tmp ;
12  done
13  print(b) ;
14 end
```

Figure 4: Implementation of the Fibonacci "*algorithm*" using *IMP*

## 3 Implementation of the syntax analyser

The *syntax analysis* (or *parsing*) is the step of the *analysis* phases that aims to verify the structure *syntax* of the source code and then reporting the potential errors using both the list of tokens generated previously by the *scanner* and a grammar (see definition later) given by the language designer. In addition, *parsers* might have many other features or aims such as type checking, information collecting and so on. Nevertheless, some designer may decide to complete these tasks during other steps.

The outcome of the parser is a *syntax tree* that will be used by the following phase (as shown in the fig.X). We distinguish two types of *parsers* : the *top-down* and the *bottom-up*. As their name indicates it, the former is constructed from the *root* to the bottom whereas the latter is constructed from the bottom to the *root*. In practice, *top-down* parsers are easier to implement than their counter part but

---

<sup>2</sup>Thierry Massart, 2014. *Programmation*. Release 3.3.3 .

```

1 begin
2   s := [45, 68, 23];
3   n := 3 ;
4   for i from 0 to n do
5     save := s[i] ;
6     j := i-1 ;
7     while j >= 0 and s[j] > save do
8       s[j+1] := s[j] ;
9       j := j-1 ;
10    done
11    s[j+1] := save ;
12  done
13 end

```

Figure 5: Implementation of a sorting algorithm using *IMP*

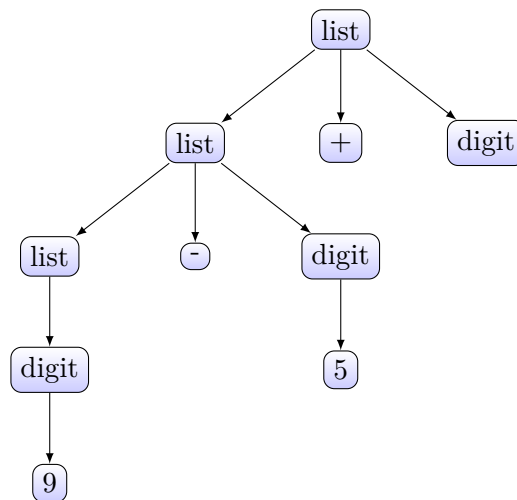
shows less performances according to [X].

As previously mentioned, *parsers* uses a specific structure called *grammar* which, similarly to spoken languages, aims to describe the allowed structures that a language can display. The *grammar* is composed of a set of variable to which are associated one or more rule(s). Typically, a programming language *grammar* is written following a fixed convention (see fig.6). Generally, in the case of a programming language, *context-free grammar* are used because of its user-friendly aspect and the fact that it allows the use of an iterative development. Using the *grammar* in fig.6 and the following

$$\begin{aligned}
 list &\rightarrow list + digit \\
 &\rightarrow list - digit \\
 &\rightarrow digit \\
 digit &\rightarrow 0|1|2|3|4|5|6|7|8|9
 \end{aligned}$$

Figure 6: Example of a simple grammar

instruction  $9-5+2$  (that is yet to be translate into a list of token) the outcome of the parser will be the following *syntax tree* :



### 3.1 Transforming the grammar to LL(1) grammar

### 3.2 Results of simplifications on the IMP grammar

## 4 How to set up the project

In order to simplify the compilation and the support of external libraries, it has been decided to use a well known *java* project manager named *Maven*. Its configuration file (`pom.xml`) defines the `main` file, defines the source folder, manages the *JFlex* library and the package that must be compiled with this library.

### 4.1 Compilation

Compiling the project with *Maven* is easy as the user only needs to execute : `mvn clean compile`. However, at the first execution, the user needs to execute `mvn install` so that *Maven* can install the required library.

If the user does not want to use *Maven*, he can execute different commands from the root project :

```
java -jar jflex-1.6.1.jar -d src/be/ac/ulb/infof403/ src/be/ac/ulb/infof403/lex/Scanner.flex
```

Where `jflex-1.6.1.jar` is the path to the `.jar` executable library, `-d` is the output folder path specifier and the last parameter is the path to the `.flex` file.

Then, the user can compile the java source codes and can create the corresponding `.class` files. The bash command to compile all the *java* files located in the `src/` folder is the following :

```
javac -d target $(find ./src/* | grep .java)
```

This command generates the corresponding `.class` files and put them in the `target/` folder. You must create the "target" folder if it does not currently exist. Finally, the *jar* file can be generated by using the command :

```
jar cvfe dist/INFO-F403-IMP.jar be/ac/ulb/infof403/Main -C target/ .
```

Where `INFO-F403-IMP.jar` is the name of the generated *jar* file and *target* is the folder where are located the `.class` files.

### 4.2 Execution

To execute the resulting jar file, the user only has to type :

```
java -jar dist/INFO-F403-IMP.jar <sourceFile>
```

Where `<sourceFile>` is the path to the IMP file. If the source file is not specified then the program will use the file `test/Euclid.imp`.

### 4.3 Test

The program has a system which automatically compares each output file (`.out`) to the result of the execution of the corresponding `.imp` file. The execution of the test can be specified by adding the parameter `-test` at the program execution instruction, like this :

```
java -jar dist/INFO-F403-IMP.jar <sourceFile> -test <testFile>
```

Where `<testFile>` is the name of the output file. If not specified, the program will automatically load a file test based on the *source file* name. It will only change the file extension from `.imp` to `.out`.

## 4.4 Javadoc

The javadoc is located in the `doc/` folder. To generate the javadoc with Maven you must execute `mvn javadoc:javadoc`. If you do not want to use Maven, you could execute the following command :

```
javadoc -d doc/javadoc/ -keywords -sourcepath src -subpackages be
```

Where `doc/javadoc/` is the output folder. The option `-keywords` enable HTML in the javadoc.

## 5 Annexe

### 5.1 Left factorisation using strees

Implying *left-factorisation* is a good practice in order to optimise a given grammar. During the practicals, the following algorithm has been given : Even though this algorithm is pretty handy for humans, automatising it is harder because of the third(?) line. In fact, finding common prefixed sequences for a human is "easy" whereas it is harder for a computer and requires further structures to be implemented. In order to implement a simple-to-use algorithm that finds common prefixes, it has been decided to use a Stree and then to analyse its structures to find the common prefixes.

A stree is a memory structure that first aims to memorise a given set of strings in a compact way using trees (the name comes from the mix of **string** and **tree**). Typically, a stree consists in a tree in which each node is associated with both a value (generally a character) and a set of other nodes called *children*. When inserting a string in the *stree*, each node will check whether one of its children is associated with the same value as the character of the string it is given. If yes, the algorithm repeats the same mechanism with the following character of the string, otherwise, it creates a new node (to which it associates the character) and repeats the mechanism.

Generally, strees are used to retrieve stocked strings but in our case, strees are used differently. Indeed, the way strees behave is perfect for left prefixes detection as each time two strings differ, the previous character (so the last common character) is the parent node of them. For instance, in fig.8, the node **B** is the last common node of the two inserted strings and because the two strings then differ, the node **B** has two children.

This behaviour is something interesting to exploit in the case of left factoring however, two adaptations are required. Firstly, we do not consider strings but sequences of tokens thus every node has an associated token and secondly, we do not want to retrieve sequences but to find nodes with more than one child so that we can replace them by new variables which have for rule(s) the following sequence of tokens.

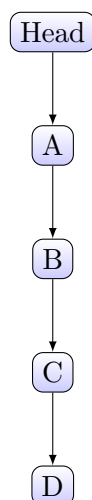


Figure 7: Insertion of 'ABCD'

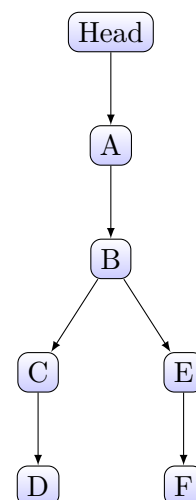


Figure 8: Insertion of 'ABEF'