# Implementation of a compiler for an imperative language IMP

Remy Detobel & Denis Hoornaert

October 24, 2017

## Contents



## 1 Introduction

The project aim is to implement a compiler for a 'simple' imperative language named *IMP*. Like any imperative programming language, *IMP* is structured 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 optimization. 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. In this assignment, the focus is set on the *analysis phase*.

## 2 Implementation of the lexical analyser

In the so called "Dragon book"[1] the *lexical anlyser* is defined as follow :

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

[1] V. Aho, A., 2007. *Compilers : Principles, techniques, & Tools*. 2nd ed. New York : Pearson.

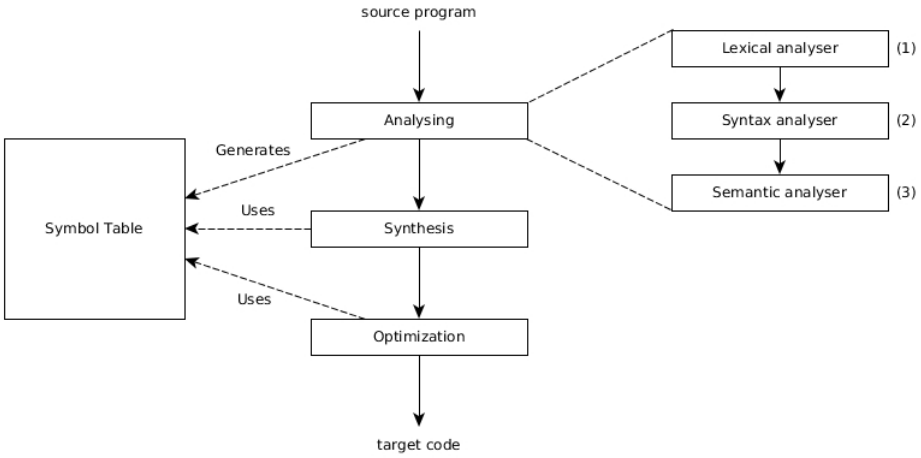


Figure 1: Compilation phases

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, that will be used by all the other steps (as shown in fig 1.), 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` many *lexical analysers* bave been develloped. 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.

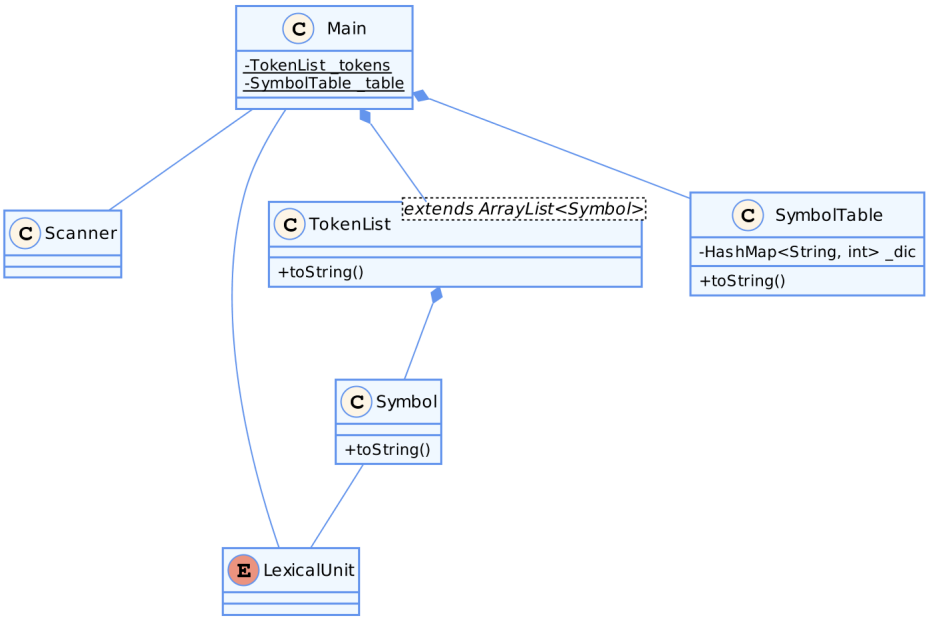


Figure 2: Model class

## 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 *keywords* and the variable/constant group.
The implementation of the *keywords* 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 directives 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]*
```

The structure of *numbers* given in the assignment directives 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 set 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]
2
```

Three 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 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 directive 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 differently :

```
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 language, the other characters are generally not included in. Given this fact and the fact that the assignment directive does not explicitly mention such a possibility, this idea has been overlooked.

- Not considering them. In this possibility, we just overlook them like if they were equivalent to a space character. This solution is the one who fits the best the assignment directives. 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*)
2
```

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 statement it came across and how many closing statement 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.

## 3 Tests and results

In this section, the results of the implementation are analysed and tested throught three *IMP* source codes : one given in the assignment directives and the two others inspired by algorithms from the Syllabus of Thierry Massart[2]. The aim of testing the *lexical analyser* on these three tests is to ensure that every *keyword* is recognized as the set of keywords in the `Euclid.imp` file is limited. This is why, even though the *IMP* language does not handle lists, these two codes have been chosen. As explained above in the Hypothesis section, every unknown/undefined character is simply overlooked. Therefore, the `[` and `]` characters (which are generally used as operator on lists) does not trouble the lexical analysing.

[2]Thierry Massart, 2014. *Programmation.* Release 3.3.3 .

```
begin
  read(a) ;
  read(b) ;
  while b <> 0 do
    c := b ;
    while a >= b do
      a := a-b
    done ;
    b := a ;
    a := c
  done ;
  print(a)
end
```

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

```
begin
    s := [45, 68, 23] ;
    bi := 0 ;
    len := 3 ;
    bs := len ;
    m := (bi+bs)/2 ;
    while bi < bs and x != s[m] do
        m := (bi+bs)/2 ;
        if s[m] < x then
            bi := m+1 ;
        else
            bs := m
        endif
    done
    if len <= m or s[m] != x then
        m := -1 ;
    endif
    print(m)
end
```

Figure 4: Implementation of a binary search using *IMP*

# 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, 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 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)
```

```
begin
    s := [45, 68, 23];
    n := 3 ;
    for i from 0 to n do
        save := s[i] ;
        j := i-1 ;
        while j >= 0 and s[j] > save do
            s[j+1] := s[j] ;
            j := j-1 ;
        done
        s[j+1] := save ;
    done
end
```

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

This command generates the corresponding ".class" files and put them in the "target" folder. If this folder does not exist at this moment it will then be created. 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`.