Implementation of a compiler for an imperative language IMP

Remy Detobel & Denis Hoornaert

October 16, 2017

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), the use of *variables*, the use *numbers* and the use of *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 composes 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*.

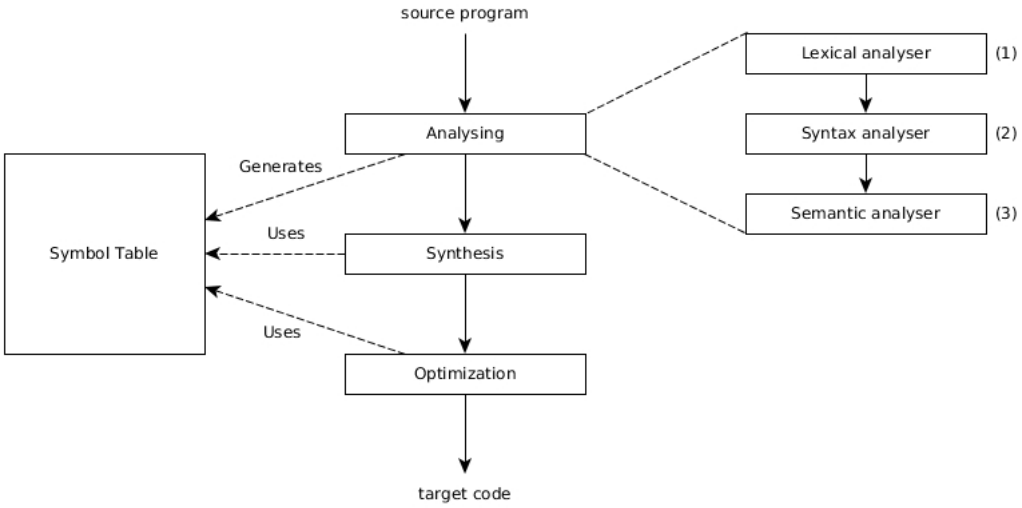

Figure 1 - Compilation phases.

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.

A *lexeme* can be defined as a tuple which contains both a *token name* and the associated value. 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 family and all its derived versions. In the present project, `jflex` is used as it has been decided to implement the project with the `java` programming language. Using a *lexical analyser generator* eases the analysis of any source 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 match them all. This generated `.java` file can then be used as any other `java` class.

2.2 Lexical analyser structure

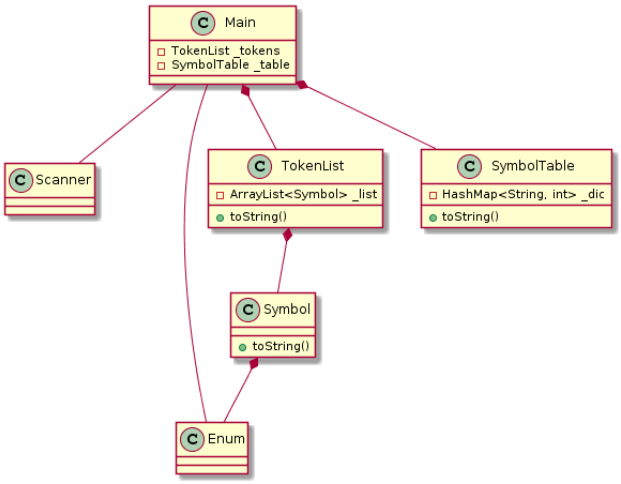

Figure 2 - Class model.

2.3 Regular expressions

pass

2.4 Tests

```
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
```

```
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
```

3 How to set up the project