Implementation of a compiler for an imperative language IMP

Remy Detobel & Denis Hoornaert

November 24, 2017

Contents

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*.

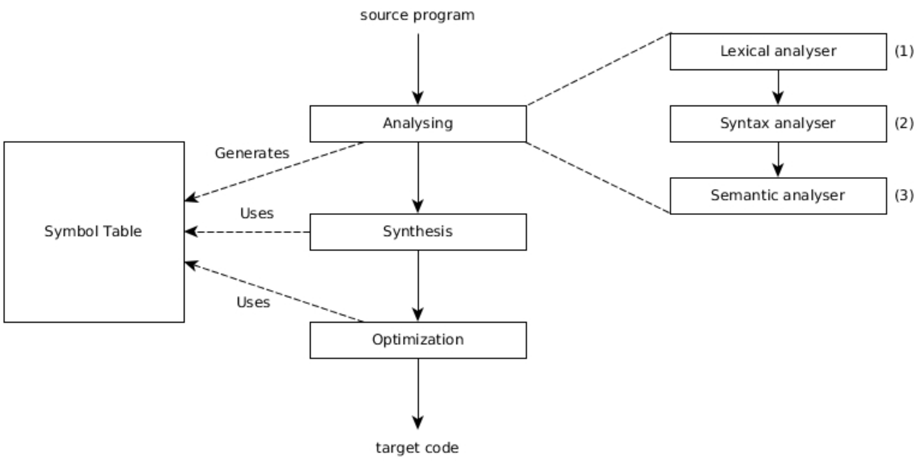


Figure 1: Compilation phases

2 Implementation of the lexical analyser

In the so called "Dragon book"[1] the *lexical anlyser* 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 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

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

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.

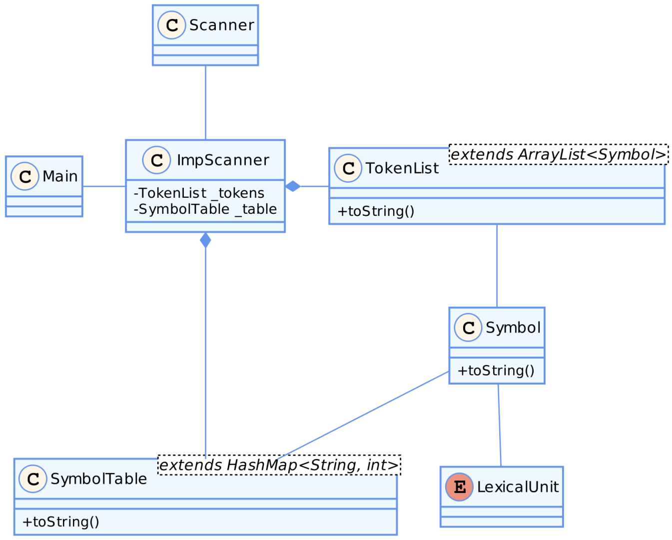


Figure 2: Model class (Pseudo-UML). The TokenList class is the sequence of lexemes and the Scanner class is the file generate by jflex

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]*
```

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 :

```
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 : '`*)`').

```
(*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 throught three *IMP* source codes : one given in the assignment statements 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 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.

```
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

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.

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

```
(*
    Algorithme to calcul Fibonacci
*)
begin
    read(n) ;
    a := 0 ;
    b := 0 ;
    for n from 0 to n do (* loop from 0 to n *)
        tmp := b ;
        b := a ;
        a := tmp
    done ;
    print(b)
end
```

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

```
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*

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 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* :

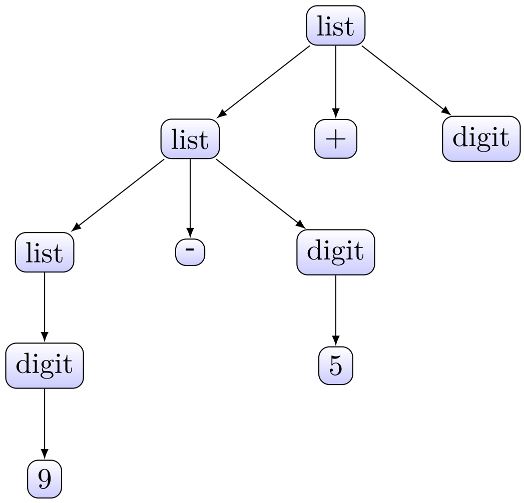


Figure 7: Syntax tree.

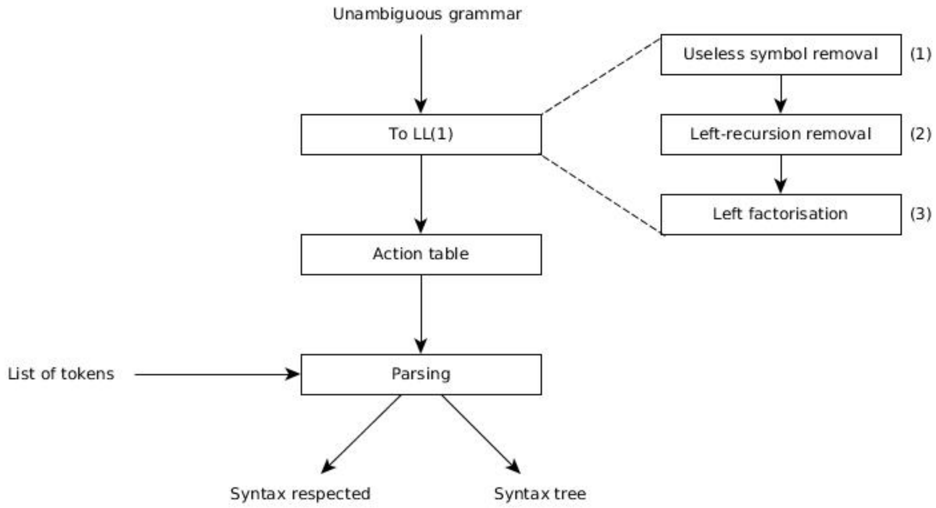


Figure 8: Phases of a parser generator

3.1 Use and implementation of a parser generator

Nowadays, when designing a programming language, one might find convenient to use a program dedicated to the determinisation of the grammar and the syntax tree generation. Such programs are well known and have been developed a long time ago (yacc in the 1970's).
However, the assignment statement does not allow the use of such a program. Henceforth, it has been decided to implement a dedicated *parser generator* so that the given grammar will be modified without suffering from human mistakes. All the steps that a *parser generator* must achieved are shown in the fig.8 and explained the section 3.2.

3.2 Transforming the grammar to LL(1) grammar

In order to transform a given grammar that can be either deterministic or non-deterministic into a LL(1) — which is deterministic —, one must apply four transformations on this grammar. These transformations are *factorisation*, *left-recursion removal*, *useless symbol removal* and *ambiguity removal*.

3.2.1 Grammar factorisation

This mechanism is applied every time a given variable has two (or more) rules that have a common prefix. The aim is to reduce the number of repetitions. To achieve this, each variable that has two (or more) rules with a common prefix sees these rules replaced by concatenation of the prefix and a new variable. This new variable has for rules the remaining of the factorized rules (i.e. the rules that have a common prefix without this prefix).

$$\begin{aligned} S &\rightarrow friendship && (1)\\ &\rightarrow friend && (2)\\ &\rightarrow relationship && (3)\\ &\rightarrow friendly && (4) \end{aligned}$$

(a) Unmodified grammar

$$\begin{aligned} S &\rightarrow relationship && (5)\\ &\rightarrow friendS' && (6)\\ S' &\rightarrow ship && (7)\\ &\rightarrow ly && (8)\\ &\rightarrow \epsilon && (9) \end{aligned}$$

(b) Factorisation outcome

Figure 9: Caption place holder

For instance, in the fig.9a, the rule (3) has no prefix whit the other rules whereas (1), (2) and (3) have a common prefix : '*friend*'. Thus, following the mechanism explained above, we replace these three rules by a new one (rule (6)) composed of the prefix ('*friend*') and the new variable (*S'*). The variable *S'* is then associated whit the remaining of each former rule with a common prefix of *S*. Notice that the rule (2) is a particular case as it matches exactly the prefix. To overcome this issue, the created rule is formed of ϵ (rule (6)).

Such a technique is used to ensure that the parser will be deterministic. In our case, we want to implement a parser with a look ahead of one. Therefore, if a variable has two (or more) rules like $S \rightarrow fA$ and $S \rightarrow fZ$, the parser won't be able to decide which one to apply.

3.2.2 Removing left-recursion

Even though recursion is a main feature of grammars as it allows them to recognise non-finite language, it also introduce non-determinism when the recursion occurs at the very first element of the right-hand side. To make a grammar (and thus the parser) deterministic but keep the recursivity, one must execute to manipulations.

$$\begin{aligned} S &\rightarrow S'b && (10)\\ S' &\rightarrow Sa && (11)\\ &\rightarrow \epsilon && (12) \end{aligned}$$

(a) Unmodified grammar

$$\begin{aligned} S &\rightarrow Sab && (13)\\ &\rightarrow a && (14) \end{aligned}$$

(b) Indirect recursion removal

$$\begin{aligned} V &\rightarrow aV' && (15)\\ V' &\rightarrow abV' && (16)\\ &\rightarrow \epsilon && (17) \end{aligned}$$

(c) Transformation to right-recursive

Figure 10: Caption place holder

First, one wants to transform every indirect left-recursion into direct left-recursion. Achieving that is quite simple as one only has to take a rule and replace every variable located at the very beginning of the left-hand side and replace it by all of its own rules. For instance, in fig.10a, the grammar is indirectly recursive because *S* call *S'* which, when applying rule (11), call *S*. The out come of this transformation (see fig.10b) recognises the same language but is now directly left-recursive.
Secondly, one wants to transform every left-recursion by a right-recursion for determinism purpose (similar to factorisation). One can achieve it by introducing two new variables. The first variable will be associated to a set of rules each composed of the concatenation of a non-recursive rule and the newly

created second variable. This second variable will be associated with a set of rules composed of every recursive rules where the first element (the recursive variable)has been removed concatenated with this exact second variable. Doing so transforms every left-recursion in a right-recursion. However, this right-recursion will never stops. This is why a rule composed of ϵ is associated to the second variable.

3.2.3 Removing useless variables

When speaking of *useless* variables, we distinguish two types of variables :

The *unproductive* ones : An unproductive variable is a variable that never leads to any formation of a word. Typically, such a variable does not have any non-recursive rule. Thus, forming a word using this variable leads to an infinite recursion.

The *unreachable* ones : An unreachable variable is a variable that is not called by any other rule of the grammar in which it belongs.

So far, the best way to find both unproductive and unreachable variables is to look respectively for productive and reachable variables and remove them from the grammar afterwards. However, eventually, we are only interested in productive and reachable variables. Thus, once the former and the latter are found, we consider them as the final grammar.

Determining the set of productive symbols consists of first considering every terminal as productive. Then, for each variable, we look at each rule and add the variable to the set if and only if every symbols appearing in the rule are already in the set. The resulting set is the set of every reachable symbols of the given grammar.

Retrieving the set of reachable symbols from a given grammar can be achieved by using a similar method to the one explained above. In fact, one must consider first a set containing only the initial variable of the grammar which is — without lost of generality — always considered as reachable. Then, for each variable of the grammar, one must check whether the variable is in the set of reachable symbols. If yes, one can then add all the symbols appearing in the rules of this variable.

3.2.4 Ambiguous grammar

Ambiguity occurs when, for a given word/input, multiple interpretations (or trees) can be derived due to an *ambiguity* in the rules the parser has to choose making it non-deterministic and thus in proper for any implementation. Unfortunately, there does not exist any algorithm resolving this issue as the given grammar gives little information. Henceforth, extra information only known by the language designer must be integrated. The most common example of ambiguity is the arithmetic priority (Reminder : the multiplication has an higher priority than the addition).

$$
\begin{aligned}
Exp &\rightarrow Exp + Exp && (18)\\
&\rightarrow Exp * Exp && (19)\\
&\rightarrow Cst && (20)\\
&\rightarrow Id && (21)
\end{aligned}
$$

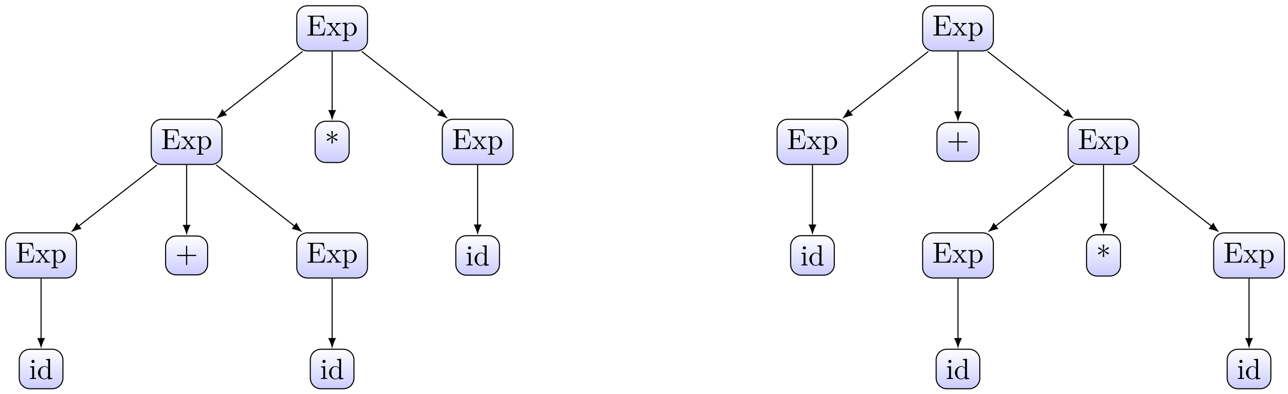
(a) Ambiguous grammar

$$
\begin{aligned}
Exp &\rightarrow Exp + Prod && (22)\\
&\rightarrow Prod && (23)\\
Prod &\rightarrow Prod * Atom && (24)\\
&\rightarrow Atom && (25)\\
Atom &\rightarrow Cst && (26)\\
&\rightarrow Id && (27)
\end{aligned}
$$

(b) Unambiguous grammar

Figure 11: Caption place holder

For example, applying the grammar of fig.11a on the word $id + id * id$ will result in two different interpretation as shown in Fig.12a and Fig.12b.



(a) First possible derivation of the input. (b) Second possible derivation of the input.

Figure 12: Derivations of the input $id + id * id$ using the grammar in fig.11a

To address this issue, the language designer must 'force' the derivation (and hence the priority) by introducing new variables that could be seen as extra layers. For instance, on fig.11a, the grammar is composed of two *atomic* terminals : *Cst* and *Id*. These terminals will be encapsulated in a new variable called *Atom*. In addition, we decide — based on the arithmetic priority — that multiplication has an higher priority than addition. Therefore, as for atomic elements, we introduce a new variable called *Prod* that has for rules a single atomic value (25) and the product of a multiplication and a atomic element (24). Finally, the same mechanism is once again applied to addition. Resulting in the rules (22) and (23).

As previously mentioned, there does not exist an algorithm that resolves grammar ambiguity. However, there exists many ambiguity detection algorithms with have their own proprieties as mentioned in this article[a]. The reader is invited to read this document for further information.

[a] H.J.S. Basten, August 17, 2007. *Ambiguity Detection Methods for Context-Free Grammars.* Master's Thesis, Universiteit Van Amsterdam.

3.3 Results of simplifications on the IMP grammar

As previously mentioned, being able to transform a non deterministic — but yet rather simple to write — into a deterministic grammar that is implementation friendly, is something one wants to do when asked to implement a parser for this grammar. In our case, all the methods previously presented in the section 3.2 have been applied to the IMP grammar (which is available in the annex fig.14) through the use of a package containing an implemented version of these methods.
The present section is composed of four subsections that each describes and analyses the different results obtained that constitute the version of the IMP grammar that is eventually used.

For a matter of readability, it has been decided to only display the modified part of the grammar. To see the full version of the modified IMP grammar one must used the process described in the subsection X.X.

3.3.1 Ambiguity

Following the assignment statement, identifying the rules where ambiguity occurs is rather simple. the two ambiguous part of the grammar involved both arithmetic expressions and conditions. For both the former and the latter, atomic rules have been identified and categorised as such/so. Then, two layers have been introduced in order to force the derivation.

In the case of arithmetic expression, the operator $-$ has been given the highest priority thus it has been introduced directly in the set of atomic rules (9 to 12). After that, the $*$ and / operators were given the highest priority, this is why they are part of the first layer (5 to 8). Finally, the second and last layer (1 to 4) is composed of the rules involving the remaining of the operators.
Notice that, even though it has not been explicitly specified in the assignment statement, any arithmetic expression surrounded by parenthesis has been considered as having a priority as high has the $-$ operator. Arithmetic expressions :

```
1  <ExprArith>       -> <ExprArith> <OpAdd> <ExprArithMul>
2  <ExprArith>       -> <ExprArithMul>
3  <OpAdd>           -> +
4  <OpAdd>           -> -
5  <ExprArithMul>    -> <ExprArithMul> <OpMul> <ExprArithAtom>
6  <ExprArithMul>    -> <ExprArithAtom>
7  <OpMul>           -> *
8  <OpMul>           -> /
9  <ExprArithAtom>   -> VarName
10 <ExprArithAtom>   -> [Number]
11 <ExprArithAtom>   -> ( <ExprArith> )
12 <ExprArithAtom>   -> - <ExprArithAtom>
```

Conditions :

```
<Cond>            -> <Cond> or <CondAnd>
<Cond>            -> <CondAnd>
<CondAnd>         -> <CondAnd> and <CondAtom>
<CondAnd>         -> <CondAtom>
<CondAtom>        -> not <SimpleCond>
<CondAtom>        -> <SimpleCond>
<SimpleCond>      -> <ExprArith> <Comp> <ExprArith>
<Comp>            -> =
<Comp>            -> >=
<Comp>            -> >
<Comp>            -> <=
<Comp>            -> <
<Comp>            -> <>
```

3.3.2 Useless symbols removal

The unambiguous IMP grammar does not contain any useless symbols. One can be easily convinced of the reachability by drawing a graph were every vertex represent the fact that the variable (a node) appears at least on one the rules of the other. One can also be convinced of the productiveness by observing that for each recursive rule, there is another that stops the recursivity. These intuitions were proved right by the implementation.

3.3.3 Left-recursion removal

When one wants to remove the left-recursion, one knows following the definition given above ntat one has to look for rules where the left-hand side is also the first element of the right-hand side.
Doing so on the unambiguous and useless symbols free IMP grammar returns once again the arithmetic expressions and the conditions. But this is not surprising given the trick used to make the grammar unambiguous. in addition, why should one pays attention to not introduce left-recursion while the left-recursion removal step will be achieved afterwards ? This explains partly the order of which these steps are executed.
In both cases, there are the introduction of a suffixed U variable and a suffixed V variable. These variables respectively remove indirect left-recursion and transform the, then, direct left-recursion into a right-recursion. Notice that the introduced right-recursion are productive as the algorithm stipulates that the suffixed V variable needs a rule composed of an ϵ.

```
<ExprArith>       -> <ExprArithU> <ExprArithV>
<ExprArithU>      -> <ExprArithMul>
<ExprArithV>      -> <OpAdd> <ExprArithMul> <ExprArithV>
```

```
<ExprArithV>       -> eps
<ExprArithMul>     -> <ExprArithMulU> <ExprArithMulV>
<ExprArithMulU>    -> <ExprArithAtom>
<ExprArithMulV>    -> <OpMul> <ExprArithAtom> <ExprArithMulV>
<ExprArithMulV>    -> eps
```

```
<Cond>             -> <CondU> <CondV>
<CondU>            -> <CondAnd>
<CondV>            -> or <CondAnd> <CondV>
<CondV>            -> eps
<CondAnd>          -> <CondAndU> <CondAndV>
<CondAndU>         -> <CondAtom>
<CondAndV>         -> and <CondAtom> <CondAndV>
<CondAndV>         -> eps
```

3.3.4 Factorisation

Looking in a grammar for rules to factorise consists in identifying two or more rules of a given variable that share a common prefix (i.e. a same sequence of tokens/symbols). Theses rules are then modified following the mechanism explained in the subsection 3.2.1.
In the case of the unambiguous, useless symbols free and left-recursion free IMP grammar, only three variables need to see their rules factorised : `InstList`, `If` and `For` (respectively line 4, 22 and 37 in fig.14).

```
1 <InstList>       -> <Instruction> <InstList'>
2 <InstList'>      -> ; <InstList>
3 <InstList'>      -> eps
```

The factorisation of `InstList` is an instance of the particular case mentioned in subsection 3.2.1. In fact, one of the rule to factorised does not really diverge as it is equal to the suffix. Thus a rule only composed of ϵ is introduced (rule 3).

```
<If>               -> if <Cond> then <Code> <If'>
<If'>              -> else <Code> endif
<If'>              -> endif
```

```
<For>              -> for VarName from <ExprArith> <For'>
<For'>             -> by <ExprArith> to <ExprArith> do <Code> done
<For'>             -> to <ExprArith> do <Code> done
```

The factorisation of `If` and `For` are quite mainstream as they present a real divergence. Thus after the prefix, a new variable is introduced and associated to the remaining of the other common prefixed rules.

3.4 Design of the LL(1) parser

A LL(1) grammar is a class of grammars that can be defined by a predictive top-down parser. Such grammars define context-free languages which can be recognised by push-down automaton (i.e. automaton that use a stack as memory). We define a predictive parser as a type of parser that has an access to the input. This access is characterised by the possibility to know what is the following character to be read (this is very different from reading on the input) in order to enable the parser to take deterministic decisions throughout its execution. Consequently the definition of the push-down automata used must be altered so that it can benefit from the look ahead advantages.
Another consequence is that there is a clear relation between the variable analysed (i.e. the one on the top of the stack) and the next character to be read. The structure established to summarised these relations is called an action table.
In the present section, we will see how an action table is constituted and how, by using all the tools previously introduced, a push-down automata can be designed to verify that a user wrote correctly a program (i.e. respected the grammar established).

3.4.1 Action table

As mentioned, an action table is a structure that emphasises the relation between a given variable and a terminal. Actually, it helps answering the following question : Which rule of a variable has to be applied if the next symbol to be read is x ?
To construct this structure, one has to introduce the function $first(\alpha)$ that returns all the symbols that can be reach in one step (i.e. using only the first element of a rule). If the first element of the rule is a terminal, $first(\alpha)$ add this terminal to the return set (the set of reachable variables). Otherwise, if the element is a variable, it calls $first(\alpha)$ on it and then merges the returned set with the current one. Unfortunately, there is a special case : the epsilon-rule[3]. In fact, because ϵ is neither a variable or a terminal and is — by definition — not expected on the input, one must, when encountering an epsilon, apply the function $first(\alpha)$ on any element following an appearance of the variable to which the epsilon-rule belongs. The research of those elements is done by a function called $follow(\alpha)$.

3.4.2 Syntax checking

Once one has an action table, checking the syntax of the input is possible through the use of a simple stack that will only contains variables and terminals. The mechanism used to check the syntax of the input works as follows. Firstly, one starts by pushing on the stack the initial variable of the grammar. Then, if the element on the top of the stack is a variable, one pops it and identifies using the action table which rule to apply given the variable and the look ahead. Once identified, we push the rule on the stack and we make sure that the left-most element is on the top of the stack. Otherwise, if the element on the top of the stack is a terminal, one pops it and compares it to the input. If it matches, one resumes, otherwise, it means that the syntax has not been respected.
In addition to that, for IMP, for add others rejecting or accepting conditions to our configuration. The IMP parser will rejects if there is no left character to read on the input and that the stack is not empty or is there are characters left on the input and that the stack is empty. The only accepting configuration is when the stack is empty and there is no character left on the input.
For example, let's consider the final IMP grammar and the following input `begin a := b end`, the sequence of used rules and the stack utilisation will be as in the fig.13.

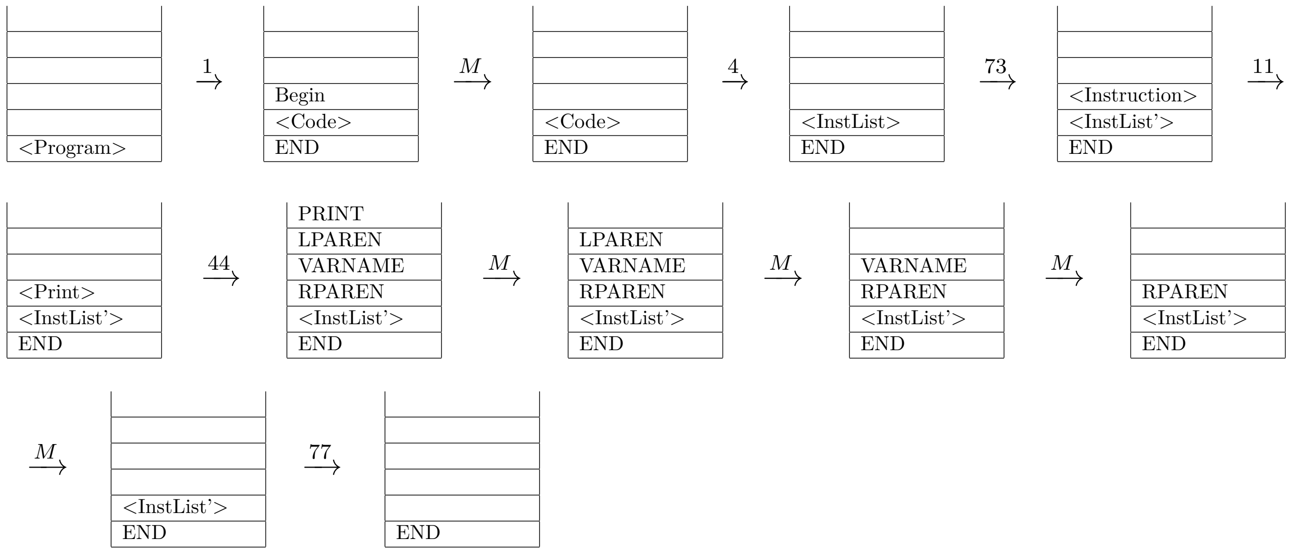


Figure 13: Stack utilisation for a simple program.

[3] a variable rule which is only composed of an epsilon (ϵ)

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 user 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 Basic IMP grammar

```
<Program>        -> begin <Code> end
<Code>           -> eps
                 -> <InstList>
<InstList>       -> <Instruction>
                 -> <Instruction> ; <InstList>
<Instruction>    -> <Assign>
                 -> <If>
                 -> <While>
                 -> <For>
                 -> <Print>
                 -> <Read>
<Assign>         -> [VarName] := <ExprArith>
<ExprArith>      -> [VarName]
                 -> [Number]
                 -> ( <ExprArith> )
                 -> - <ExprArith>
                 -> <ExprArith> <Op> <ExprArith>
<Op>             -> +
                 -> -
                 -> *
                 -> /
<If>             -> if <Cond> then <Code> endif
                 -> if <Cond> then <Code> else <Code> endif
<Cond>           -> <Cond> <BinOp> <Cond>
                 -> not <SimpleCond>
                 -> <SimpleCond>
<SimpleCond>     -> <ExprArith> <Comp> <ExprArith>
<BinOp>          -> and
                 -> or
<Comp>           -> =
                 -> >=
                 -> >
                 -> <=
                 -> <
                 -> <>
<While>          -> while <Cond> do <Code> done
<For>            -> for [VarName] from <ExprArith> by <ExprArith> to <ExprArith> do <Code> done
                 -> for [VarName] from <ExprArith> to <ExprArith> do <Code> done
<Print>          -> print ( [VarName] )
<Read>           -> read ( [VarName] )
```

Figure 14: The basic IMP grammar as given in the assignment statement.

	(	<=	begin	if	do	-	read	while	and	then	:=	VarName	)	else	end	done	not	by	from	>=	<>	to	print	for	or	+	[Number]	;	=	/	*	<	endif	>
If				67																														
CondAndU	61					61						61					61										61							
InstList				73			73	73				73											73	73										
Assign												13																						
ExprArith	50					50						50															50							
Program			2																															
ExprArithMulV		53			53	53			53	53				53	53	53		53		53	53	53			53	53		53	53	52	52	53	53	53
CondV					58					58															57									
OpMul																														21	20			
ExprArithMulU	51					51						51															51							
ExprArithMul	55					55						55															55							
For'																		81				83												
If'														69																			71	
CondAtom	33					33						33					32										33							
ExprArithU	46					46						46															46							
InstList'														77	77	77												75					77	
Instruction				8			12	9				7											11	10										
CondU	56					56						56					56										56							
OpAdd						17																				16								
ExprArithV		48			48	47			48	48				48	48	48		48		48	48	48			48	47		48	48			48	48	48
CondAndV					63				62	63															63									
Read							45																											
While								41																										
SimpleCond	34					34						34															34							
ExprArithAtom	24					25						22															23							
Comp		38																		36	40								35			39		37
Cond	60					60						60					60										60							
For																								79										
Code				4			4	4				4		3	3	3							4	4									3	
CondAnd	65					65						65					65										65							
Print																							44											

Table 1: IMP action table.

5.2 IMP action table

5.3 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 memorised a given set of strings in a compact way using trees (the names comes from the mix of **st**ring and tr**ee**). 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 has 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 string but in our case, strees are used differently. Indeed, the way strees behave is perfect for left prefixes detection as each time to strings differ, the previous character (so the last common character) is the parent node of them. For instance, in fig.16, 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 string but sequence 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 variable which has for rule(s) the following sequence of tokens.

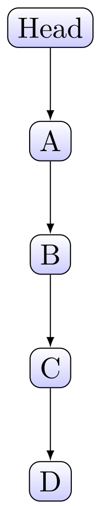


Figure 15: Insertion of 'ABCD'

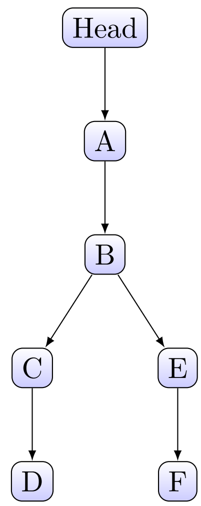


Figure 16: Insertion of 'ABEF'