Implementation of a compiler for an imperative language IMP

Remy Detobel & Denis Hoornaert

December 23, 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*.

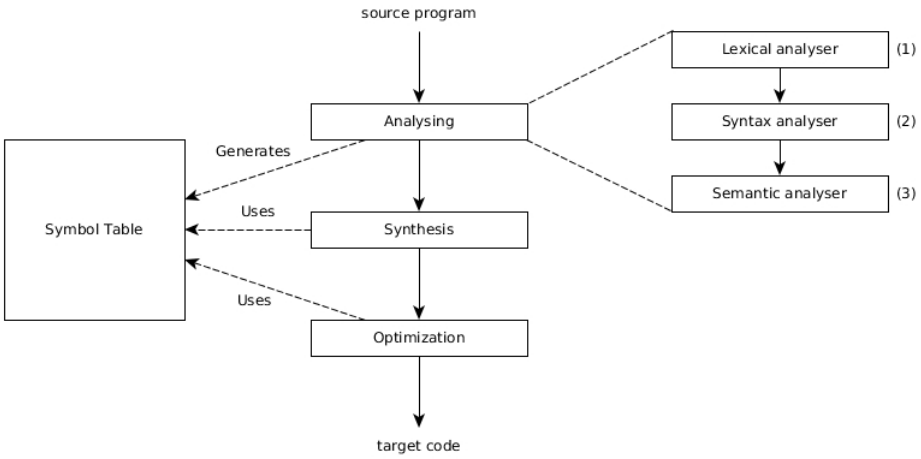


Figure 1: Compilation phases

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.

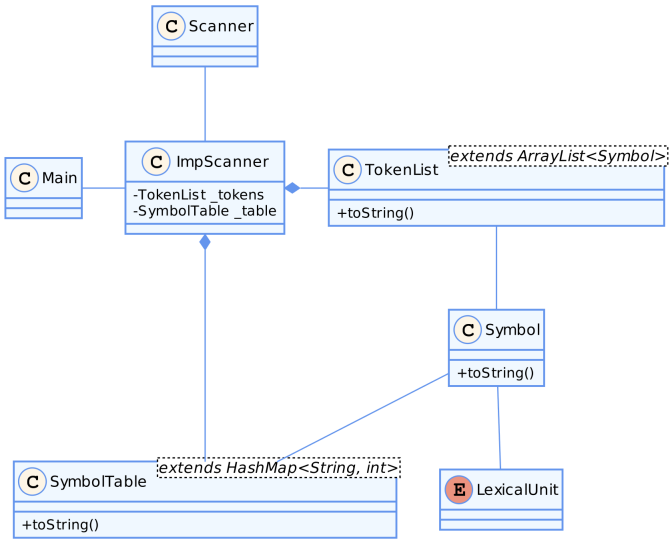


Figure 2: Model class (Pseudo-UML). The TokenList class is the sequence of lexemes and the Scanner class is the file generate by jflex

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 : '`*)`').

```
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 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 1 ('`[`'). 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

3 Implementation of the syntax analyser

The *syntax analysis* (or *parsing*) is the step of the *analysis* phases (see fig.1) that aims to verify the *syntax* of the source code (under the form of a list of tokens) and to report the potential errors using a grammar (see definition later) given by the language designer.
The outcome of the parser is a *syntax tree* that will be used by the following phase (as shown in the fig.1). We distinguish two types of *parsers* : the *top-down* and the *bottom-up*.

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

```
(*
    Algorithm to compute Fibonacci
*)
begin
    read(n);
    a := 0;
    b := 1;
    for i from 0 to n do (* loop from 0 to n *)
        tmp := a + b;
        a := b;
        b := 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*

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 variables to which are associated at least one rule. Typically, a *grammar* is written following a fixed convention (see fig.12a or any other grammar). 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 an iterative development. However, *context-free* grammars are not well adapted for implementation. Therefore, *parser generators* usually transform them into *LL(K)* grammars. In the assignment statement, it is asked to transform the IMP grammar into a *LL(1)*.
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 (other than read). 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.

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 *parse tree* generation. Such programs are well known and have been developed a long time ago (e.g. 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 possible human mistakes. All the steps that a *parser generator* must achieved are shown in the fig.6 and explained the section 3.2.1.

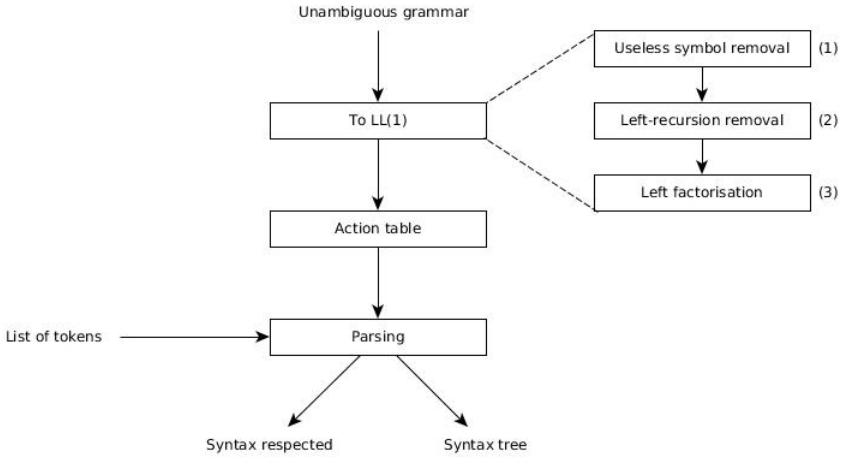


Figure 6: Phases of a parser generator

3.2 Parser phases

The present section is divided in three subsections each explaining the utility of the different mechanisms that composes a *parser generator*. The first subsection concerns the transformation of a *context-free* grammar into an *LL(1)* grammar, the second subsection focuses on the creation of an *action table* and the last subsection explains how to check the syntax of an input.
A reader aware of the mechanisms used can easily skip the present section and resumes at the results section (3.4).

3.2.1 Transforming the grammar into an 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 aim to make the grammars deterministic and thus implementable.

Ambiguity removal : Consists of ensuring, by the introduction of rule layers, that for a same input only one interpretation/derivation is possible. It also allows to force and to fix the priority and the associativity of some terminals in the grammar. (For further explanations see annex A.4)

Useless variable removal : Consists of removing every variable that does not appear in any other rule of the grammar (as they will never be called/used) and removing variable that does not contain a rule capable of stopping the recursion of the other rules of the same variable. (For further explanations see annex A.3)

Left-recursion removal : Consists of modifying every rules where the recursion occurs at the first element so that a grammar can still accept non-finite languages but becomes deterministic because the parser will eventually always encounter a terminal. (For further explanations see annex A.2)

Left-factorisation : Consists of finding every rules of a variable that have a common prefix (i.e. same sequence of tokens) and of modifying it in consequence so that the parser can take deterministic decision. For example : which rule to choose between $S \rightarrow \alpha A$ and $S \rightarrow \alpha Z$ when the look ahead is α. (For further explanations see annex A.1)

3.2.2 Action table

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 reached in one step (i.e. using only the first element of a rule). If the first element of the rule is a terminal, $first(\alpha)$ adds this terminal to the returned 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 that owns the epsilon-rule. The research of those elements is done by a function called $follow(\alpha)$.

$S \rightarrow aAcCa$ (1)
$A \rightarrow Sc$ (2)
$A \rightarrow \epsilon$ (3)
$C \rightarrow c$ (4)

(a) Grammar

	a	c
S	1	
A	2	3
C		4

(b) Action table

Figure 7: Simple grammar and its corresponding action table

3.2.3 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 contain 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 *tos* (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 the rule identified, we push it on the stack (with the left-most element on the tos). Otherwise, if the element on the *tos* is a terminal, one pops it and compares it to the input. If it matches, one resumes by repeating the process. Otherwise, it means that the syntax has not been respected.
In addition to that, one can add new accepting or rejecting conditions based on the "state" of the stack and/or the input. For instance in the case of IMP, the parser will rejects if there is no character to read on the input and if 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 print(a) end`, the sequence of the rules used and the stack utilisation is as displayed in the fig.8.

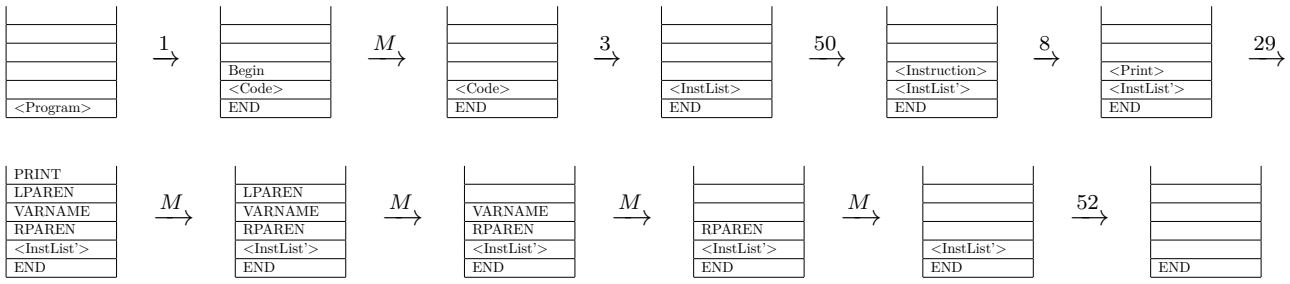


Figure 8: Stack utilisation for a simple program.

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

3.3 Resulting parser generator architecture

After having listed the main steps, it is easier to design a full program architecture.
All the mechanisms needed to transform a grammar into a LL1 are well encapsulated in an object called `Grammar` (see annex 18 for the pseudo-uml). This object is also used to play the role of the action table. Thus, no `ActionTable` class has been implemented. Consequently, the `Grammar` object and the `TokenList` are used by an object called `Ll1`. Similarly to the theory, this object aims to verify the syntax, with the help of a stack, following the mechanism described in the subsection 3.2.3. One might notice the presence of an object `Stack`. In fact, the syntax checking method explicitly uses a dedicated object `Stack`, which implies that the method is not recursive. In addition, `Ll1` also creates a parse tree while checking the syntax. Typically, the tree is composed of a bunch of `RuleTree` objects following the composite design pattern. Afterwards, this tree is manipulated by other objects (not displayed on purpose) to create a parser tree visualisation.

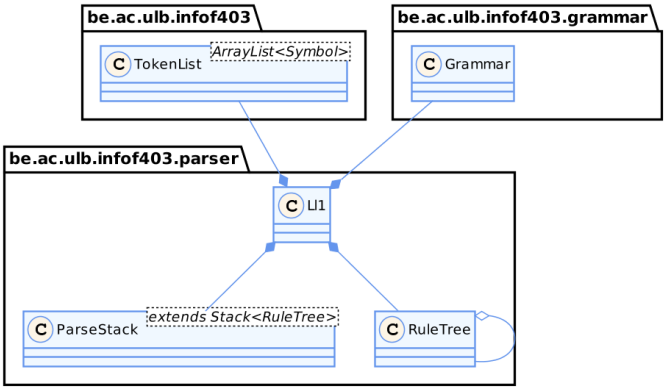


Figure 9: Pseudo uml of the LL1 architecture

3.4 Tests and results

As previously mentioned, being able to transform a non deterministic — but yet rather simple to write — grammar into a implementable deterministic grammar, is something one wants to do when asked to implement a parser based on this grammar. In our case, all the methods previously presented in the section 3.2.1 have been applied to the IMP grammar (which is available in the annex fig.16) through the use of a package containing an implemented version of these methods.
The present section is composed of six subsections. The four first subsections describe and analyse the steps of *LL(1)* transformation whereas the two last subsections focus respectively on the action table generated by the implementation and the results of the syntax checking.

> For a matter of readability, it has been decided to only display the modified parts of the grammar. To see the full version of the modified IMP grammar one must used the process described in the subsection 5.5 or must see annex fig.17.

3.4.1 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. Then, two layers have been introduced in order to force the derivation.
In the case of arithmetic expressions, 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 operators (+ and −).
Notice that, even though it has not been explicitly specified in the assignment statement, any arithmetic expressions surrounded by parenthesis has been considered as having a priority as high has the − operator.

```
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>
```

The same modifications have been applied on conditions than on arithmetic expressions.

```
1  <Cond>            -> <Cond> or <CondAnd>
2  <Cond>            -> <CondAnd>
3  <CondAnd>         -> <CondAnd> and <CondAtom>
4  <CondAnd>         -> <CondAtom>
5  <CondAtom>        -> not <SimpleCond>
6  <CondAtom>        -> <SimpleCond>
7  <SimpleCond>      -> <ExprArith> <Comp> <ExprArith>
8  <Comp>            -> =
9  <Comp>            -> >=
10 <Comp>            -> >
11 <Comp>            -> <=
12 <Comp>            -> <
13 <Comp>            -> <>
```

3.4.2 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 variable is represented by a node and every edge represents the fact that a variable appears at least once in one the rules of the other. One can also be convinced of the grammar productiveness by observing that for each recursive rule, there is another that stops the recursivity. These intuitions were proved right by the implementation.

3.4.3 Left-recursion removal

When one wants to remove the left-recursion, one knows, following the definition given above, that 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. This partly explains the reasons behind the order of the steps.
In both cases, there are the introduction of a suffixed U variable and a suffixed V variable. These variables are respectively used to remove indirect left-recursion and to transform each direct left-recursion into a right-recursion. Notice that the introduced right-recursion are productive as the algorithm stipulates that the suffixed V variable must own an epsilon-rule[4]. (see lines 4 and 8 of the following grammars)

```
1 <ExprArith>       -> <ExprArithU> <ExprArithV>
2 <ExprArithU>      -> <ExprArithMul>
3 <ExprArithV>      -> <OpAdd> <ExprArithMul> <ExprArithV>
4 <ExprArithV>      -> eps
5 <ExprArithMul>    -> <ExprArithMulU> <ExprArithMulV>
6 <ExprArithMulU>   -> <ExprArithAtom>
```

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

```
<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
```

3.4.4 Factorisation

Looking in a grammar for rules to factorise consists in identifying variables that have two or more rules that share a common prefix (i.e. a same sequence of tokens). Theses rules are then modified following the mechanism explained in the subsection A.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.16).

```
<InstList>       -> <Instruction> <InstList'>
<InstList'>      -> ; <InstList>
<InstList'>      -> eps
```

The factorisation of `InstList` is an instance of the particular case mentioned in subsection A.1. In fact, one of the rules to be factorised does not really diverge as it equals the suffix. Thus an epsilon-rule[4] 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` is quite mainstream as they present a real divergence. Thus after the prefix, a new variable is introduced and associated to the remaining of the common prefixed rules.

3.4.5 Resulting action table

The last tool required before starting to check the syntax of the input is the action table. Based on the final grammar obtained using the implementation and the function $first(\alpha)$ and $follow(\alpha)$, the action displayed in the annexe 1 has been obtained. At the first look, the content of the action table might seems weird but everything is logic.
Firstly, the variables *<Print>*, *<Read>*, *<Program>*, *<Assign>*, *<While>*, *<If>* and *<For>* have only one filled cell. This can easily be explained as they all own only one rule that starts with a terminal. Thus nothing else than this terminal can be observed during the parsing. In the case of *<If>* and *<For>*, it was expected as it was meant after the factorisation. Notice that, as expected regarding the results of the factorisation, the two derived variables *<If'>* and *<For'>* both have two cells filled. On the other hand, *<InstList'>* has more than to two cells filled (actually five) due to the introduced ϵ.
Secondly, the terminals *:=* and *from* have no filled cells due to the place in which they appear in the grammar : never as the first element of a rule and always preceded by an other terminal (*VarName* for both) which explains why even $follow(\alpha)$ cannot reach them.
Finally, *begin* is the only terminal that has one filled cell and it is not surprising as it only appears at the first element of the only rule of the initial variable of the grammar (cannot be reach by $follow(\alpha)$).

3.4.6 Syntax checking results

After having used three different source codes to test the lexical analyser (subsection 2.5), we use them to assess the parser.
Firstly, `Euclid.imp` (fig.3) totally respects the syntax defined by the final IMP grammar and requires 120 transitions (matches excluded).
Secondly, `Fibonacci.imp` (fig.4) also respects the syntax and requires 85 transitions (matches excluded). During the very first test of the parser on `Fibonacci.imp`, we encountered an error and then realised that the first version (written during the realisation of first part of the project) contained an error. In fact, the instruction line 11 ended with a semicolon and thus did not respect the grammar (more precisely the rule 52 of the final grammar (annex fig.17)). Despite being counter intuitive for a programmer, this rule helped us ensuring the well functioning of the parser.

```
(line : 12 col :4) Unexepected character 'done' of type DONE
    The expected character is one of the following : while if print VarName for read
```

Finally, as expected, `Sort.imp` (fig.5) is not even parsed as an error occurs during the scanning. Hence, the compiler will display the following message :

```
Error with a token: Unknown symbol '[' (9 character at line 1)
```

4 Implementation of the Code generator

The last part of the project aims to translate the IMP code into an intermediate representation using both the *parse tree* generated during the *syntax analysis* and the *symbol table* obtained during the *lexical analysis* (see fig.1).
Typically, no *semantic analysis* has been required as the IMP language only allows to manipulate integers. Hence, this phase of the compilation scheme has been skipped.
In the present project, the targeted code is *LLVM*. This low-level programming language is widely used and pretty popular among language designer because it is not linked to any particular processor architecture but can be itself compiled in order to target a specific processor architecture.
For instance, this important feature would alow the project to run on both `x86_64` and `ARM` processors without having to deal with the fundamental hardware differences.

4.1 Structure of the code generator

The choice made is to simply create a large set of classes where each class correspond to a grammar variable of the final IMP grammar (see annex 17). All th classes inherit from `RuleTree` so that they become fully part of the *parse tree*. To enable the parser to know exactly which type of node to instanciate, a factory called `RuleTreeFactory` has been introduced. The *parse tree* construction mechanism is as follow :

1. given a grammar variable, the parser watches the look-ahead and determines the rule.
2. the parser sends to the `RuleTreeFactory` the grammar variable just used.
3. the parser receives back a `RuleTree` object (which is actually one of its children)
4. the parser adds the received object in its *parse tree*

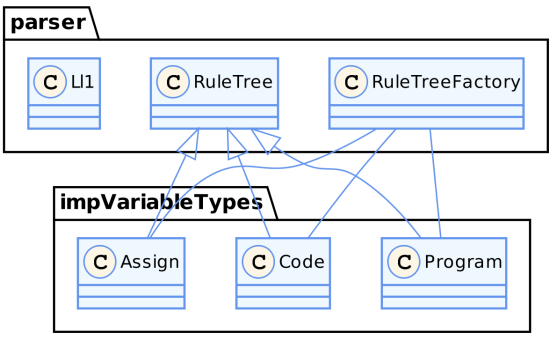


Figure 10: Pseudo-uml of the RuleTree composite and of the Factory that builds it.

4.2 How the project generates LLVM code

Like mention in the section 3.3, the parse tree is composed by `RuleTree` object. We have create a children of this class for each state of the grammer. To generate the LLVM code we just make recursive call. Indeed the first state `Program` call the result of `Code`. Which call the result of the instruction, ect.
When a state compute his result he could also "output" LLVM code (this output is first store in varable and then output on the right support: file or terminal). In other words, children of `RuleTree` have a method which return the value that represent the current state **and** output an LLVM code to compute this value.
This method work for a lot of state. There are just some problem with the addition structure. The figure 11 show us that the state `ExprArithMulV` have not all element to compute the division. To fix that there are two choice: get the missing value from the parent or compute the division on another state. We chose to implement the second option. Thus the state `ExprArithMul` will compute the result of the division and store this value on a new variable.
To make that the state `ExprArithMul` will get the operator with two get: one on his second children and then directly ask to get the value of the first children. To have the value 4 (in this case), `ExprArithMul` will ask at state `ExprArithMulV` to compute his final value. Then, `ExprArithMul` have all element to compute the division.

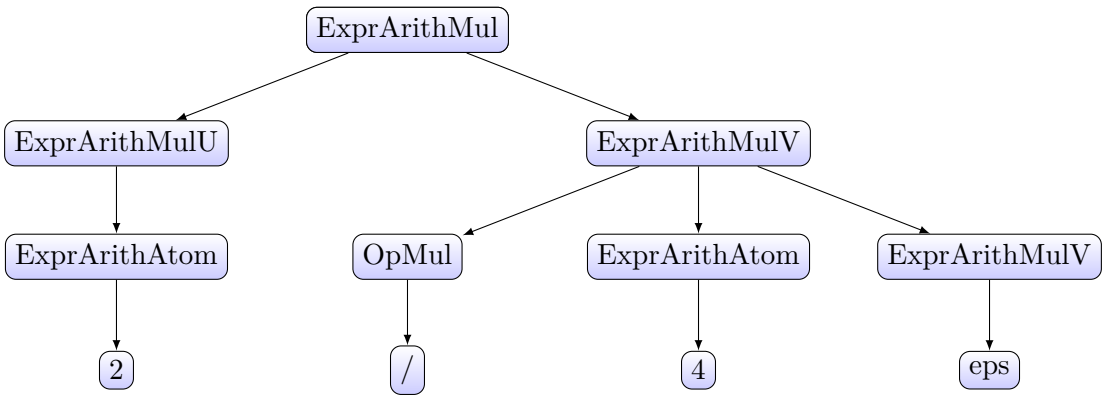


Figure 11: Part of parse tree which compute Expression.

4.3 Limitations

Even though the mechanism used to generate the *LLVM* code was simple to simple to implement, not transforming the *parse tree* returned by the *syntax analyser* into an *Abstract Syntax tree* or another structure disabled us from using many optimisation techniques mentioned in the assignment statement. The most obvious consequence is that it is impossible to implement a mechanism that evaluates the arithmetic expressions for which the involved variables value is already known at the compilation

time. This has for consequence that useless code is generated by the compiler and that the resulting executable will be heavier than what might be achieved. Hopefully, the *LLVM* compiler/interpreter implements this features.

4.4 Hypothesis

Throughout the project, only one hypothesis has been settled. This hypothesis is the following : when using the rule

```
for varName from <ExpreArith> by <ExpreArith> to <ExpreArith> do <Code> done
```

the first `<ExpreArith>` has to be strictly lower than the third `<ExpreArith>` and the second `<ExpreArith>` has to be greater or equal than 0 (even if incrementing by 0 does not make any sense).
This issue arises because, unlike c/c++, the programmer does not have to explicitly specify the condition. Thus, the compiler has to be able to decide by itself whether a $>$ or $<$ operator should be used. This would imply that the compiler should be able to evaluate its arithmetic expressions during the *compilation time* but this is not possible as mentioned above (see section 4.3). Further more, if the arithmetic expressions of the `for` involve a varibale that is modified in the instruction block, no evaluation is possible. In sum, no decision over the operator to used can be taken.

4.5 Results

You can find the LLVM code of Euler (figure 19) and of Fibonacci alorithm in annex (figure 20).

5 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.

5.1 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/scanner/ src/be/ac/ulb/
    infof403/lex/Scanner.flex
java -jar jflex-1.6.1.jar -d src/be/ac/ulb/infof403/grammar/ src/be/ac/ulb/
    infof403/lex/GrammarScanner.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. The user 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 the `.class` files are located.

5.2 Execution

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

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

Where `<IMPFile>` is the path to the IMP file (by default: `./test/imp/Euclid.imp`) and `<grammarFile>` is the path to the Grammar used to parse IMP file (by default: `./test/grammar/UnambiguousIMP.gram`).

5.3 Test

Maven contains instruction to automatically make some test. If the user wants to execute these tests, he must execute this command: `mvn verify`. The user can execute the tester using the compilation command : `mvn verify clean compile`.
There are not lot of tests but it's easy to create more. Tests are located in the folder `srcTest/`. It's also possible to test the IMP scanner. Indeed, the user can see the outcome of the scanning of the IMP file by adding the argument `--printscan` (or the alias `-ps`) and can compare this output to an existing "`.out`" file. To do so, the user needs to add the argument `--testscan` (or the alias `-ts`) followed by the expeted output file.
To summarize, you could execute:

```
java -jar dist/INFO-F403-IMP.jar -ps -ts test/imp/Euclid.out
```

Note that if the user does not specify the expected output file, the program will automatically load a test file based on the *source file* name.

5.4 Javadoc

The javadoc is located in the `doc/` folder. To generate the javadoc with Maven the user must execute `mvn javadoc:javadoc`. If he does not want to user Maven, he can execute the following command :

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

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

5.5 Parameters

The user can see all the parameters by using the argument `--help` (or the alias `-h`). If he wants to analyse a specific IMP file he **must** precise the grammar file first. There is no order for the other arguments. The user will find bellow the list of possible arguments:

- `-h/--help` Displays all arguments
- `-o/--output` Write the generated llvm code into the specied file
- `-e/--exec` Perform an output (see point just above) and execute directly the generated LLVM.
- `-gram/--grammar` Print all informations about grammar enhance
- `-ta/--table` Prints the action table (see point 3.2.2)
- `-ts/--testscan` Checks that the result of the scan is the same that the expected output (see point 5.3).

- `-ps/--printscan` Prints the scan result (used to compare with the ".out" file (if specified with previous argument)).
- `-gojs/--gojstree` Displays the parse tree using GoJS library.
- `-tex/--latex` Displays the parse tree in a LaTeX format. (Note: this method uses the Tikz library and big trees cannot be compiled).
- `-d/--debug` Displays a lot of debug messages (used to develop)

A Annex

A.1 Grammar factorisation

This mechanism is applied every time a given variable has two (or more) rules with 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).

$$
\begin{aligned}
S &\rightarrow friendship && (5)\\
&\rightarrow friend && (6)\\
&\rightarrow relationship && (7)\\
&\rightarrow friendly && (8)
\end{aligned}
$$

(a) Unmodified grammar

$$
\begin{aligned}
S &\rightarrow relationship && (9)\\
&\rightarrow friendS' && (10)\\
S' &\rightarrow ship && (11)\\
&\rightarrow ly && (12)\\
&\rightarrow \epsilon && (13)
\end{aligned}
$$

(b) Factorisation outcome

Figure 12: Left-factorisation example

For instance, in the fig.12a, the rule (7) has no prefix whit the other rules whereas (5), (6) and (8) have a common prefix : '*friend*'. Thus, following the mechanism explained above, we replace these three rules by a new one (rule (10)) 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 (6) is a particular case as it matches exactly the prefix. To overcome this issue, the created rule is formed of ϵ (rule (13)).

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.

A.2 Removing left-recursion

Even though recursion is a main feature of grammars as it allows them to recognise non-finite languages, it also introduces 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 while keeping the recursivity, one must execute to manipulations.

$$
\begin{aligned}
S &\rightarrow S'b && (14)\\
S' &\rightarrow Sa && (15)\\
&\rightarrow \epsilon && (16)
\end{aligned}
$$

(a) Unmodified grammar

$$
\begin{aligned}
S &\rightarrow Sab && (17)\\
&\rightarrow b && (18)
\end{aligned}
$$

(b) Indirect recursion removal

$$
\begin{aligned}
V &\rightarrow bV' && (19)\\
V' &\rightarrow abV' && (20)\\
&\rightarrow \epsilon && (21)
\end{aligned}
$$

(c) Transformation to right-recursive

Figure 13: Left-recursion removal example

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 the variable located at the very beginning of the left-hand side and replace it by all of its own rules. For instance, in fig.13a, the grammar is indirectly recursive because *S* calls *S'* which, when applying rule (15), calls *S*. The out come of this transformation (see fig.13b) 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.

A.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.

A.4 Ambiguous grammar

Ambiguity occurs when, for a given word/input, multiple interpretations (or trees) can be derived due to a multiple choice of rules, leading the parser to take a non-deterministic choice and thus make it not adapted for 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).
For example, applying the grammar of fig.14a on the word $id + id * id$ will result in two different interpretation as shown in Fig.15a and Fig.15b.

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.14a, 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 (29) and the product of a multiplication and a atomic element (28). Finally, the same mechanism is once again applied to addition. Resulting in the rules (26) and (27).

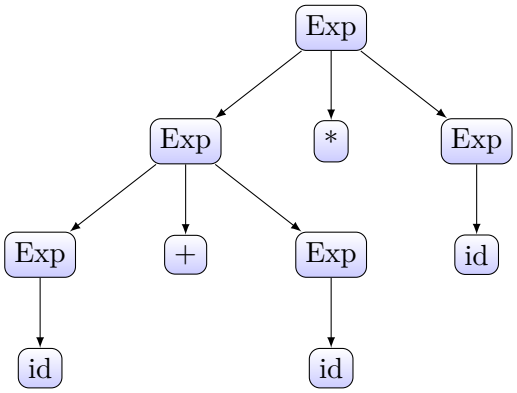
$$
\begin{aligned}
Exp &\rightarrow Exp + Exp && (22)\\
&\rightarrow Exp * Exp && (23)\\
&\rightarrow Cst && (24)\\
&\rightarrow Id && (25)
\end{aligned}
$$

(a) Ambiguous grammar

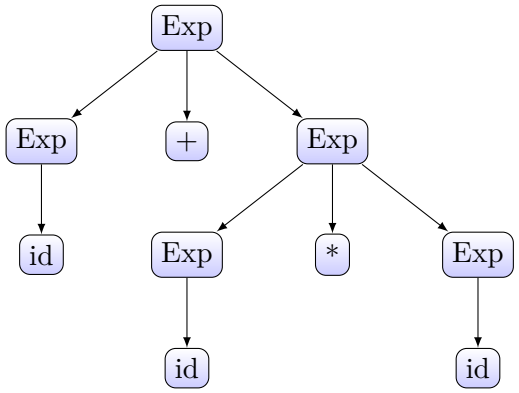
$$
\begin{aligned}
Exp &\rightarrow Exp + Prod && (26)\\
&\rightarrow Prod && (27)\\
Prod &\rightarrow Prod * Atom && (28)\\
&\rightarrow Atom && (29)\\
Atom &\rightarrow Cst && (30)\\
&\rightarrow Id && (31)
\end{aligned}
$$

(b) Unambiguous grammar

Figure 14: Unambiguity mechanism example



(a) First possible derivation of the input.



(b) Second possible derivation of the input.

Figure 15: Derivations of the input $id + id * id$ using the grammar in fig.14a

As previously mentioned, there does not exist an algorithm that resolves grammar ambiguity. However, there exists many ambiguity detection algorithms that have their own proprieties as listed 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.

	(	<=	begin	if	do	-	read	while	and	then	:=	VarName	)	else	end	done	not	by	from	>=	<>	to	print	for	or	+	[Number]	;	=	/	*	<	endif	>
If				47																														
CondAndU	43					43						43					43										43							
InstList				50			50	50				50											50	50									
Assign												10																					
ExprArith	34					34						34															34							
Program			1																														
ExprArithMulV		37			37	37			37	37			37	37	37	37		37		37	37	37			37	37		37	37	36	36	37	37	37
CondV					41					41															40								
OpMul																														14	13			
ExprArithMulU	35					35						35															35							
ExprArithMul	38					38						38															38							
For’																		54				55											
If’														48																		49	
CondAtom	20					20						20					19										20							
ExprArithU	31					31						31															31							
InstList’														52	52	52												51					52	
Instruction				5			9	6				4											8	7									
CondU	39					39						39					39										39							
OpAdd						12																				11							
ExprArithV		33			33	32			33	33			33	33	33	33		33		33	33	33			33	32		33	33			33	33	33
CondAndV					45				44	45															45								
Read							30																										
While								28																									
SimpleCond	21					21						21															21							
ExprArithAtom	17					18						15															16							
Comp		25																		23	27								22			26		24
Cond	42					42						42					42										42							
For																								53									
Code				3			3	3				3		2	2	2							3	3									2	
CondAnd	46					46						46					46										46							
Print																							29											

Table 1: IMP action table.

```
<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 16: The basic IMP grammar as given in the assignment statement.

```
<Program>          -> begin <Code> end
<Code>             -> eps
<Code>             -> <InstList>
<Instruction>      -> <Assign>
<Instruction>      -> <If>
<Instruction>      -> <While>
<Instruction>      -> <For>
<Instruction>      -> <Print>
<Instruction>      -> <Read>
<Assign>           -> VarName := <ExprArith>
<OpAdd>            -> +
<OpAdd>            -> -
<OpMul>            -> *
<OpMul>            -> /
<ExprArithAtom>    -> VarName
<ExprArithAtom>    -> [Number]
<ExprArithAtom>    -> ( <ExprArith> )
<ExprArithAtom>    -> - <ExprArithAtom>
<CondAtom>         -> not <SimpleCond>
<CondAtom>         -> <SimpleCond>
<SimpleCond>       -> <ExprArith> <Comp> <ExprArith>
<Comp>             -> =
<Comp>             -> >=
<Comp>             -> >
<Comp>             -> <=
<Comp>             -> <
<Comp>             -> <>
<While>            -> while <Cond> do <Code> done
<Print>            -> print ( VarName )
<Read>             -> read ( VarName )
<ExprArithMulU>    -> <ExprArithAtom>
<ExprArithMulV>    -> <OpMul> <ExprArithAtom> <ExprArithMulV>
<ExprArithMulV>    -> eps
<ExprArithMul>     -> <ExprArithMulU> <ExprArithMulV>
<CondAndU>         -> <CondAtom>
<CondAndV>         -> and <CondAtom> <CondAndV>
<CondAndV>         -> eps
<CondAnd>          -> <CondAndU> <CondAndV>
<CondU>            -> <CondAnd>
<CondV>            -> or <CondAnd> <CondV>
<CondV>            -> eps
<Cond>             -> <CondU> <CondV>
<ExprArithU>       -> <ExprArithMul>
<ExprArithV>       -> <OpAdd> <ExprArithMul> <ExprArithV>
<ExprArithV>       -> eps
<ExprArith>        -> <ExprArithU> <ExprArithV>
<InstList>         -> <Instruction> <InstList'>
<InstList'>        -> ; <InstList>
<InstList'>        -> eps
<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
```

Figure 17: The final IMP grammar after modifications.

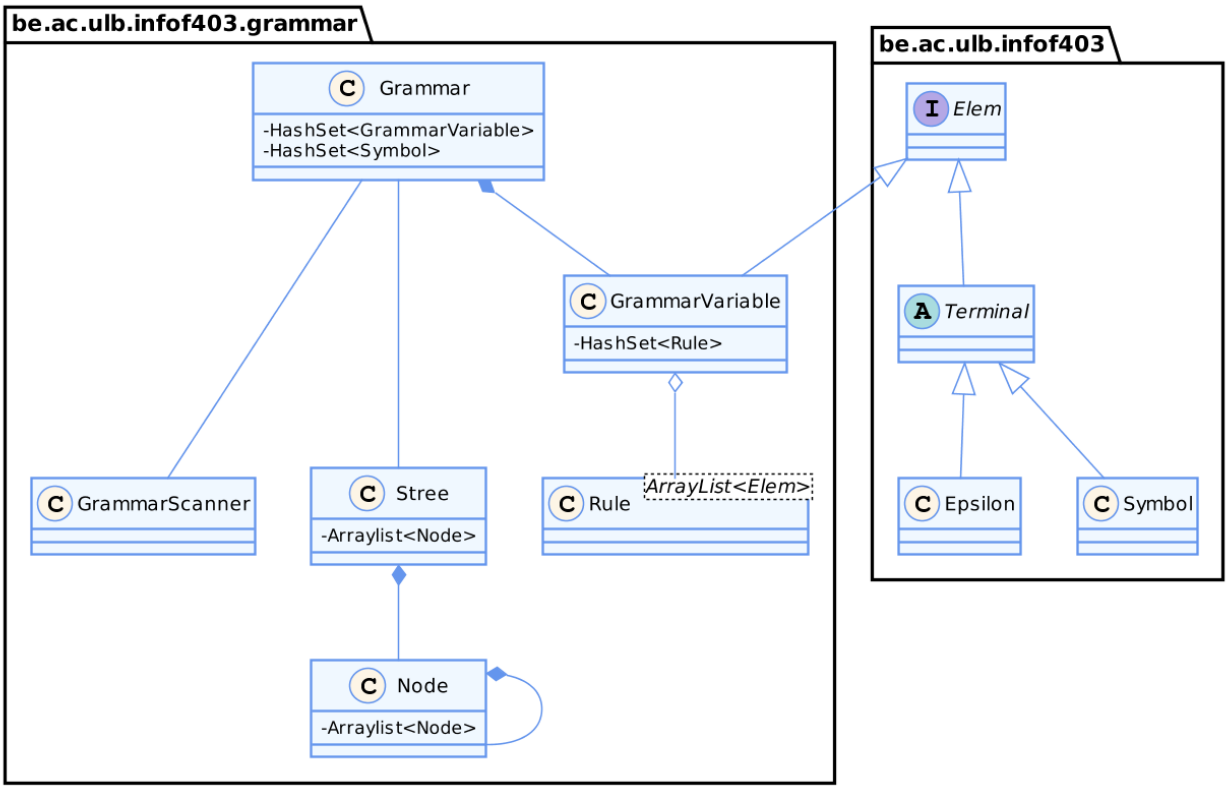


Figure 18: Architecture of the grammar

```
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1

define i32 @readInt() {
  %x = alloca i32, align 4
  %1 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x
    i8]* @.str, i32 0, i32 0), i32* %x)
  %2 = load i32, i32* %x, align 4
  ret i32 %2
}

declare i32 @__isoc99_scanf(i8*, ...)
@.str2 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

define void @println(i32 %x) {
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.
    str2, i32 0, i32 0), i32 %2)
  ret void
}

declare i32 @printf(i8*, ...)
define i32 @main() {
%a = alloca i32
%b = alloca i32
%c = alloca i32
%1 = call i32 @readInt()
store i32 %1, i32* %a
%2 = call i32 @readInt()
store i32 %2, i32* %b
br label %startloop26
startloop26:
%3 = load i32, i32* %b
%4 = icmp ne i32 %3, 0
br i1 %4, label %loop26, label %endloop26
loop26:
%5 = load i32, i32* %b
store i32 %5, i32* %c
br label %startloop83
startloop83:
%6 = load i32, i32* %a
%7 = load i32, i32* %b
%8 = icmp sge i32 %6, %7
br i1 %8, label %loop83, label %endloop83
loop83:
%9 = load i32, i32* %b
%10 = load i32, i32* %a
%11 = sub i32 %10, %9
store i32 %11, i32* %a
br label %startloop83
endloop83:
%12 = load i32, i32* %a
store i32 %12, i32* %b
%13 = load i32, i32* %c
store i32 %13, i32* %a
br label %startloop26
endloop26:
%14 = load i32, i32* %a
call void @println(i32 %14)
ret i32 0
}
```

Figure 19: The final LLVM code generation for Euclid.

```
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1

define i32 @readInt() {
  %x = alloca i32, align 4
  %1 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x
    i8]* @.str, i32 0, i32 0), i32* %x)
  %2 = load i32, i32* %x, align 4
  ret i32 %2
}

declare i32 @__isoc99_scanf(i8*, ...)
@.str2 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

define void @println(i32 %x) {
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.
    str2, i32 0, i32 0), i32 %2)
  ret void
}

declare i32 @printf(i8*, ...)
define i32 @main() {
%a = alloca i32
%b = alloca i32
%tmp = alloca i32
%i = alloca i32
%n = alloca i32
%1 = call i32 @readInt()
store i32 %1, i32* %n
store i32 0, i32* %a
store i32 1, i32* %b
store i32 0, i32* %i
br label %startloop56
startloop56:
%2 = load i32, i32* %n
%3 = load i32, i32* %i
%coundRes56 = icmp slt i32 %3, %2
br i1 %coundRes56, label %loop56, label %endloop56
loop56:
%4 = load i32, i32* %b
%5 = load i32, i32* %a
%6 = add i32 %5, %4
store i32 %6, i32* %tmp
%7 = load i32, i32* %b
store i32 %7, i32* %a
%8 = load i32, i32* %tmp
store i32 %8, i32* %b
%9 = load i32, i32* %i
%inc56 = add i32 %9, 1
store i32 %inc56, i32* %i
br label %startloop56
endloop56:
%10 = load i32, i32* %b
call void @println(i32 %10)
ret i32 0
}
```

Figure 20: The final LLVM code generation for Fibonacci.