Introduction to language theory and compiling
Project – Part 2

Antoine Passemiers
Alexis Reynouard

November 26, 2017

Contents

1 Transforming Imp grammar **1**

1.1 Remark about symbols 2

1.2 Removing useless rules and symbols 2

1.2.1 Unreachable symbols 2

1.2.2 Unproductive variables 3

1.3 Removing ambiguities 5

1.3.1 Operator priority 5

1.3.1.1 The problem of parentheses 6

1.3.1.2 Arithmetic operators 7

1.3.1.3 Logical operators 7

1.3.2 Operator associativity 8

1.4 Removing left-recursion and applying factorization 8

1.4.1 Left-recursion removal 8

1.4.2 Left-factoring 8

1.5 Resulting grammar 8

2 First sets, follow sets, action table **11**

2.1 Proving that Imp grammar is now LL(1) 11

2.2 First sets 11

2.3 Follow sets 11

2.4 Action table 11

3 Implementation of the LL(1) parser **12**

3.1 Theoretical considerations 12

3.2 Automating the process, but not too much 13

3.3 Grammar symbols and rules 13

3.4 Action table 14

3.5 LL(1) parser 14

3.6 How to use the parser 14

1. Transforming Imp grammar

This part of the project consists in implementing a $LL(k)$ parser for the Imp programming language. A $LL(k)$ parser is a recursive descent parser composed of:

- An **input buffer**, containing k input tokens. Since we are designing a $LL(1)$ parser only, the latter only considers one token at a time to decide how to grow the syntactic tree.
- A **stack** containing the set of remaining terminals and non-terminals to process.
- An **action table**, mapping the front of the stack and the current token to the corresponding rule. Every time a rule is retrieved from the action table, the top of the stack is replaced by the sequence of symbols to the right of the current rule.

One is not able to construct an action table if the given grammar is not $LL(1)$. For that purpose, we were asked to modify the grammar in such a way that it becomes possible to determine the $first^1$ and $follow^1$ sets of each symbol of the grammar. More specifically, we must remove useless variables, unproductive rules, take into account operators priority and associativity, remove left-recursion and finally left-factor the grammar.

The order in which one processes these steps in crucial: if one tries to add rules with respect to the operators priority, one may encounter problems due to unproductive rules. In addition to that, the method we were suggested to use to deal with associativity and priority actually introduces new left-recursive rules. That is the reason why the left-recursion removal must be applied after using this method.

1.1 Remark about symbols

In this report, we will refer to symbols as **grammar symbols** or simply **symbols**. To speak about symbols for the lexer, we will refer to it as **lexical symbols**. The distinction is crucial because the lexer and the parser are based on different types of automata. The deterministic automaton (used by the lexer) can only process **characters** while the pushdown automaton (used by the parser) takes **tokens as input**. A token is an instance of a lexical unit and consists in a sequence of characters. Plus, tokens are generated by the lexer and used as input to the parser. By nature, they use **different alphabets**.

Also, we will refer to sequences of grammar symbols as **words**. They latter are different from lexical words, which represent tokens.

1.2 Removing useless rules and symbols

A **useless symbol** is a symbol that is either unproductive (terminals are productive by nature, so they are not considered when we talk about productivity) or unreachable. A **useless rule** can only be an unproductive rule, which means that it contains unproductive variables. A variable is called unproductive if none of its derivations yields anything. An unreachable symbol is a symbol that cannot be derived from the start grammar symbol. Let's summarize: a rule is useless if it is unproductive, a terminal is useless if it is unreachable, and a variable is useless if it is either unreachable or unproductive. This step of the project consists in removing those symbols and rules, when they are useless.

1.2.1 Unreachable symbols

Unreachable symbols are symbols that cannot be derived from the start variable. The start variable <Program> is reachable by nature because every sequence of input tokens is a program and this implies that it is the only variable to begin with. The property of a reachable variable A is as follows: if there exists a rule of the form $A \rightarrow \alpha$, then all symbols in sequence α are reachable as well. By induction, we can infer that any symbol is reachable if:

$$<\text{Program}> \Rightarrow_G^* A\beta B, \text{ where } A, B \in (T \cup V)^* \text{ and } \beta \in (T \cup V) \tag{1.1}$$

where G is the Imp grammar.

To remove the unreachable symbols, we first enumerate the reachable symbols using the following procedure:

- Initialize the set of reachable symbols as the set containing only the start symbol <Program>.
- Every time a rule of the form $A \rightarrow \alpha$ is found, where A is a reachable variable, we add every symbol from sequence α to the set of reachable symbols.
- We stop after we have explored all rules without discovering new reachable symbols.

This yields the following table:

i	V_i
0	$\{Program\}$
1	$V_0 \cup \{Code\}$
2	$V_1 \cup \{IntList\}$
3	$V_2 \cup \{Instruction\}$
4	$V_3 \cup \{Assign, If, While, For, Print, Read\}$
5	$V_4 \cup \{ExprArithm, Cond\}$
6	$V_5 \cup \{Op, BinOp, SimpleCond\}$
7	$V_6 \cup \{Comp\}$
8	V_7

We observe that in the end the set of reachable symbols contains all the symbols of the grammar. As a result, **all symbols are reachable**. To validate this result, we implemented the unreachable-symbol removal algorithm and got the same result.

1.2.2 Unproductive variables

A rule is productive if all symbols of its right-hand part are productive. Because terminals are always productive, we must only look for productive variables. The intuition is that a variable is unproductive if none of its derivations yields anything. In other words, it is

impossible to find a **finite** word such that it can be derived from this variable. Typically, this happens when the variable is recursive. For example:

$$<\text{ExprArith-p1}> \rightarrow <\text{ExprArth-p1}> + <\text{ExprArith-p2}> \tag{1.2}$$

This rule, **all by itself** (with no other rule including $<$ExprArithmP1$>$ as left-hand variable), is unproductive. Indeed, the derivation requires an infinite number of steps to reach $+ <$ExprArithP2$>$, which results in an infinite word of variables. To be productive, this word must be composed of **terminals**. More formally, a productive rule $A \rightarrow \alpha$ must be such that its left-hand productive variable yields a finite sequence of terminals:

$$A \Rightarrow_G^* w \text{ where } w \in T^* \tag{1.3}$$

where G is the Imp grammar.

To remove unproductive rules, we had to remove unproductive variables first. The easiest way to proceed is to enumerate the productive variables. We did it as follows:

- Initialize the set of productive symbols as the set of terminals, because we know that terminals are productive by nature. Indeed, a string composed of a **finite number of terminals** is finite itself.
- Every time a rule of the form $A \rightarrow \alpha$ is found, where α is a string composed of productive symbols, we add its left-hand variable to the set of productive symbols. The reason is that a string composed of strings of terminals are also finite. Using an inductive reasoning, one can notice that this intuition holds for as many recursion levels as needed.
- We stop after we have explored all rules without discovering new productive variables.

This yields the following table:

i	V_i
0	T
1	$V_0 \cup \{Code, ExprArithm, Op, BinOp, Comp, Print, Read\}$
2	$V_1 \cup \{Program, Instruction, Assign, For, SimpleCond\}$
3	$V_2 \cup \{IntList, Cond\}$
4	$V_3 \cup \{While, If\}$
5	V_4

At the end, we get that the set contains all grammar symbols. As consequence, **all symbols are productive**. To validate this result, we implemented the unproductive-rule removal algorithm and got the same result.

1.3 Removing ambiguities

A grammar is ambiguous if one can draw at least two different derivation trees from a same input word, using the grammar rules. The Imp grammar is ambiguous because of its logical and arithmetic operators. Let's replace ambiguous rules by new rules in such way that is takes into account both operator associativity and operator priority.

1.3.1 Operator priority

Let's take the sequence of tokens $A + B * C$ as example. In any case, a top-down parser would apply rule <ExprArith> $\rightarrow$ <ExprArith> <Op> <ExprArith>. Because of the ambiguity of the grammar, the parser would not know which operator to apply next, it could use any of the two following rules:

<Op> $\rightarrow$ +
$\rightarrow$ *

Because operators * and / have a higher priority than operators + and -, the parser must consider expression $A + B * C$ as **the sum of two expressions** A and $B * C$. Let's rename A by <ExprArith-p0> (where p0 stands for lowest priority 0), and $B * C$ by <ExprArith-p1>. We now get a new rule:

<ExprArith-p0> $\rightarrow$ <ExprArith-p0> <Op> <ExprArith-p1>

where <Op> can be any of the operators that have the same priority level as + (actually we only consider + and -). The problem of this technique is that **it makes the assumption that the total expression contains an addition or a subtraction**. Let's add a rule to give the parser the opportunity to skip these operations if it needs to:

<ExprArith-p0> $\rightarrow$ <ExprArith-p1>

The grammar now takes into account the differences in priority between {+, -} and {*, /}, but not between {*, /} and operators () and unary minus. Because unary minus has a higher priority than operators * and /, let's use the same method. This results in the following new rules:

<ExprArith-p1>	→	<ExprArith-p1> <Op-p1> <Atom>
	→	<Atom>
<Atom>	→	- <Atom>

where <Op-p1> can be any of the operators that have the same priority level as * (can be either * or /).

1.3.1.1 The problem of parentheses

The problem of parentheses is less trivial than the unary minus problem. Let's take $(A + B) * C$ as example. We must add a new rule such as the () operator has the highest priority. To respect the priority order, this rule must be of the form
<Atom> $\rightarrow \alpha$. In our example, **the parser will apply the multiplication first** (because applying addition first would lead the parser in a state where (A and B) are atoms). Let's consider the case where we use this rule:

<Atom>	→	(<Atom>)

The parser will apply it and end up with $A + B$ as an atom. Since $A + B$ is **not a terminal**, this produces an error. Because $A + B$ is an arithmetic expression itself, we must not regard it as an atom. Let's correct the rule:

<Atom>	→	(<ExprArith-p0>)

Let's check the derivation of $(A + B) * C$ (and skip some steps for making it more concise):

<ExprArith-p0>
→ <ExprArith-p1>
→ <ExprArith-p1> <Op-p1> <Atom>
→ <Atom> * <Atom>
→ (<ExprArith-p0>) * [VarName]
→ (<ExprArith-p0> <Op-p0> <ExprArith-p1>) * [VarName]
→ (<ExprArith-p1> + <ExprArith-p1>) * [VarName]
→ (<Atom> + <Atom>) * [VarName]
→ ([VarName] + [VarName]) * [VarName]

We obtain the desired outcome. Let's add rule <Atom> → (<ExprArith-p0>) to the grammar.

1.3.1.2 Arithmetic operators

Let's summarize all the changes we did to the grammar:

<SimpleCond>	→	<ExprArith-p0> <Comp> <ExprArith-p0>
<ExprArith-p0>	→	<ExprArith-p0> <Op-p0> <ExprArith-p1>
	→	<ExprArith-p1>
<ExprArith-p1>	→	<ExprArith-p1> <Op-p1> <Atom>
	→	<Atom>
<Atom>	→	[VarName]
	→	[Number]
	→	(<ExprArith-p0>)
	→	- <Atom>
<Op-p0>	→	+
	→	-
<Op-p1>	→	*
	→	/

1.3.1.3 Logical operators

We apply the same reasoning we did for arithmetic operators. The changes to apply to deal with priorities of operators {or, and, not} **are similar to the changes we made** to take the priorities of operators {+, *, -} into account. After repeating our analysis, we obtain the following new rules for logical operators:

<Cond-p0>	→	<Cond-p0> or <Cond-p1>
	→	<Cond-p1>
<Cond-p1>	→	<Cond-p1> and <Cond-p2>
	→	<Cond-p2>
<Cond-p2>	→	not <SimpleCond>
	→	<SimpleCond>

1.3.2 Operator associativity

TODO: Dire que les regles qu'on a ajoutees sont recursives gauche, donc associatives gauche, pour ce qui est des parentheses, l'associativite n'a pas de sens. Enfin pour le - unaire, montrer que c'est bon aussi

1.4 Removing left-recursion and applying factorization

1.4.1 Left-recursion removal

TODO: Dis-donc Jamy, c'est quoi la recursivite a gauche ? Faudrait pas lire les notes de Geeraerts sur ca ? Expliquer pourquoi cet algo marche

¡A¿ ¡A¿a On ajoute: ¡A¿ ¡A-i¿ ¡A-j¿ ¡A-j¿ epsilon et, pour chacune rgle ¡A¿ ¡A¿b on change par ¡A-j¿ b et, pour chaque rgle ¡A¿ c on change par ¡A-i¿ c

1.4.2 Left-factoring

TODO: Pareil, expliquer

<If>	→	if <Cond> then <Code> <IfTail>
<IfTail>	→	endif
	→	else <Code> endif
<For>	→	for [VarName] from <ExprArith> <ForTail>
<ForTail>	→	<ExprArith> to <ExprArith> do <Code> done
	→	<ExprArith> do <Code> done

1.5 Resulting grammar

[1]	<Program>	→ `begin` <Code> `end`
[2]	<Code>	→ `epsilon`
[3]		→ <InstList>

[4] <Instruction> → <Assign>
[5] → <If>
[6] → <While>
[7] → <For>
[8] → <Print>
[9] → <Read>
[10] <Assign> → `[VarName] :=` <ExprArith-p0>
[11] <ExprArith-p0-j> → <Op-p0> <ExprArith-p1>
[12] <ExprArith-p0-i> → <ExprArith-p1>
[13] <ExprArith-p1-j> → <Op-p1> <Atom>
[14] <ExprArith-p1-i> → <Atom>
[15] <Atom> → `[VarName]`
[16] → `[Number]`
[17] → `(` <ExprArith-p0> `)`
[18] → `-` <Atom>
[19] <Op-p0> → `+`
[20] → `-`
[21] <Op-p1> → `*`
[22] → `/`
[23] <Cond-p0-j> → `or` <Cond-p1>
[24] <Cond-p0-i> → <Cond-p1>
[25] <Cond-p1-j> → `and` <Cond-p2>
[26] <Cond-p1-i> → <Cond-p2>
[27] <Cond-p2> → `not` <SimpleCond>
[28] → <SimpleCond>
[29] <SimpleCond> → <ExprArith-p0> <Comp> <ExprArith-p0>
[30] <Comp> → `=`
[31] → `>=`
[32] → `>`
[33] → `<=`
[34] → `<`
[35] → `<>`
[36] <While> → `while` <Cond-p0> `do` <Code> `done`
[37] <Print> → `print ( [VarName] )`
[38] <Read> → `read ( [VarName] )`
[39] <If> → `if` <Cond-p0> `then` <Code> <If-Tail>
[40] <If-Tail> → `else` <Code> `endif`
[41] → `endif`

[42] <For> → `for [VarName] from` <ExprArith-p0> <For-Tail>
[43] <For-Tail> → `by` <ExprArith-p0> `to` <ExprArith-p0> `do` <Code> `done`
[44] → `to` <ExprArith-p0> `do` <Code> `done`
[45] <InstList> → <Instruction> <InstList-Tail>
[46] <InstList-Tail> → `;` <InstList>
[47] → `epsilon`
[48] <ExprArith-p1> → <ExprArith-p1-i> <ExprArith-p1-j>
[49] <ExprArith-p0-j> → `epsilon`
[50] <Cond-p1> → <Cond-p1-i> <Cond-p1-j>
[51] <ExprArith-p0> → <ExprArith-p0-i> <ExprArith-p0-j>
[52] <Cond-p0-j> → `epsilon`
[53] <Cond-p0> → <Cond-p0-i> <Cond-p0-j>
[54] <ExprArith-p1-j> → `epsilon`
[55] <Cond-p1-j> → `epsilon`

2. First sets, follow sets, action table

TODO: Expliquer l'intuition derriere les ensembles first et follow, a quoi sert l'action table, etc.

2.1 Proving that Imp grammar is now LL(1)

2.2 First sets

2.3 Follow sets

2.4 Action table

3. Implementation of the LL(1) parser

3.1 Theoretical considerations

The objective is to implement a **top-down parser** that tells whether a word (sequence of tokens) is part of the language (Imp grammar) or not. This parser begins with the start variable and builds the tree in a depth-first fashion using **production rules**. More specifically, it consists of a pushdown automaton (PDA) that accepts the language by empty stack. Let's define this PDA formally:

$$P_{Imp} = (\{q\}, T, V \cup T, \delta, q, \$, \phi) \tag{3.1}$$

- Because P_{Imp} accepts the grammar **by empty stack** and not by accepting state, it only requires the state to work. Let's call this state q. The set of states is thus $\{\mathrm{q}\}$.
- The alphabet of the input buffer is T. The set of terminals T is the set of **lexical units** of Imp grammar. Each terminal represents a type of token.
- The alphabet of the stack corresponds to $T \cup V$. Indeed, the stack is supposed to contain both lexical units and variables at some point, since variables are mandatory to use production rules.
- The transition function δ is such that the PDA can either **produce** or **match**. This will be explained in more details later on.
- The start symbol is q.

- We use $ as the first symbol to push on the stack. When the stack becomes empty, the input buffer must have only one token remaining, which is $. If it is not the case, the word is rejected.
- The set of accepting states is empty since we are not accepting words by accepting states, but by empty stack.

TODO: Comment ca marche ?

3.2 Automating the process, but not too much

Hard coding is bad. That is the reason why we did not described the grammar in the source code itself. We encoded the Imp grammar (as it was presented in the project statement) in a dedicated *more/grammars* folder and called *imp.grammar*. After that, we checked for useless rules and symbols (and found nothing) by hand but also programmatically, and finally dealed with the operators priority and associativity by hand. The resulting grammar file, called *imp_prim.grammar* is used as input by our program. We will refer to the later as the **unambiguous Imp grammar**.

We removed left-recursion, left-factored the grammar and finally computed first sets and follow sets of each grammar symbol by hand. Everything has been verified programmatically. When the parser is executed on an input Imp file, a lexer is instantiated to be able to get the resulting list of tokens. Then this sequence of tokens is passed to the parser that first loads the unambiguous Imp grammar, processes all the steps that we just enumerated **to make it LL(1)**, builds its action table from first sets and follow sets, and finally does the parsing itself.

3.3 Grammar symbols and rules

Because everything had to be implemented from scratch, we had to define what grammar symbols actually are. Because class *Symbol* was already present in our source code, and because it is (as explained earlier) fondamentally different from grammar symbols, we created a new class *GrammarSymbol*. A grammar symbol **can be either a variable or a**

terminal. Everytime one does need to know whether a grammar symbol is a terminal or not, he calls the method *isTerminal*.

A *Rule* is an object that consists of a left-hand variable represented by a *GrammarSymbol* and a right-hand sequence of symbols represented by a list of *GrammarSymbol* objects. All rules are instantiated when loading the unambiguous grammar file. For better productivity, we added special methods to display the rules, pretty print them or export them to LaTex.

3.4 Action table

Class *ActionTable* has been implemented as a HashMap, where keys are pairs of *GrammarSymbol* objects, and values are integers that represent actions. Each key pair is composed of two parts: the symbol that is on top of the stack, and the next input terminal to read. Integer values that are positive represent rule numbers, in particular the rule that must be used to produce. Strictly negative integers represent other types of actions: -1 stands for **match**, -2 stands for **accept** and -3 stands for **error**.

3.5 LL(1) parser

3.6 How to use the parser

Arguments :
(1) [OPTIONS] –ru ¡grammar file¿ -o ¡grammar output file¿
(2) [OPTIONS] –ll ¡grammar file¿ -o ¡grammar output file¿
(3) [OPTIONS] –at ¡ll1 unambiguous grammar file¿
(4) [OPTIONS] ¡ll1 unambiguous grammar file¿ ¡code¿ -o ¡latex output file¿