Introduction to language theory and compiling
Project – Part 2

Antoine Passemiers
Alexis Reynouard

November 27, 2017

Contents

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 unambiguous 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 he tries to add rules with respect to the operators priority, he 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 and, optionally, some additional information. 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:

$$\text{<Program>} \Rightarrow_G^* A\beta B, \text{ where } A, B \in (T \cup V)^* \text{ and } \beta \in (T \cup V) \tag{1}$$

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:

$$\text{<ExprArith-p1>} \rightarrow \text{<ExprArth-p1>} + \text{<ExprArith-p2>} \tag{2}$$

This rule, **all by itself** (with no other rule including `<ExprArith-p1>` as left-hand variable), is unproductive. Indeed, the derivation requires an infinite number of steps to reach + `<ExprArith-p2>`, 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:

$$A \Rightarrow^*_G w \text{ where } w \in T^* \tag{3}$$

where $\Rightarrow^*_G$ is the reflexo-transitive closure of $\Rightarrow$ accordingly to 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>` → `<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>    → +
        → *
```

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>   →   <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>   →   <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 /).

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> → α. 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.

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>	→	*
	→	/

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

Because **we introduced left-recursive rules to remove ambiguities**, the latter are left-associative. Let's take the example of $4 + 5 + 8$. The following rule will be applied to that expression:

<ExprArith-p0> → <ExprArith-p0> <Op-p0> <ExprArith-p1>

where <Op-p0> will then produce a $+$. Because the only variable that can produce a sum is <ExprArith-p0>, this same rule will be applied one again, with $4 + 5$ being <ExprArith-p0>. It follows that the expression will be evaluated as $(4 + 5) + 8$.

The case of parentheses is trivial because there is no order in parentheses associativity. The fact that the rule is left-recursive or right-recursive has no effect on it. Finally, it makes no sense to speak about associativity for unary operators. It results that we did not have to deal with the right-associativity of the unary minus.

1.4 Removing left-recursion and applying factorization

1.4.1 Left-recursion removal

Because left recursive rules cause problems to build the parser (for example to compute the $First^1$ sets of the rules, as it will be explained below), we have to remove it. Note that a productive left recursion may be direct, *i.e.* of the form:

<A>	→	<A> a
<A>	→	ϵ

or of an indirect form:

<A>	→	<B> a
<B>	→	<S> b
<B>	→	ϵ

The later form does not occur in the Imp grammar. So we have implemented an algorithm to remove the left recursion that does not take into account indirect left recursion.

The idea behind this algorithm it to transform left-recursion into right recursion. To achieve this, we take all the variables <A> that appear in a rule of the form <A> → <A>a. For each of these variables, we remove the problematical rule and replace it by <A> → <A-i><A-j>. <A-i> will represent the first symbol f

1.4.2 Left-factoring

Left-factoring consists in removing groups of rules that have the following form:

<A>	→	$\alpha\beta$
<A>	→	$\alpha\gamma$

where α, β, γ are sequences of symbols $\in (T \cup V)^*$. The problem of these rules is that **they consume some look-ahead** and make the grammar non-LL(1). More specifically, if α produces at least one terminal (and not only ϵ), it follows that the rule to used becomes unpredictable based only on the LL(1) action table. If the first terminal produced by α is, let's say token *for*, it will be the case for all rules where the right-hand part starts with α. Using a look-ahead of one symbol, the parser can see that the next input token is *for*, but that is not sufficient to decide which rule to use. Such a grammar would be LL(p), where p is the maximal number of terminals that can be produced using any of the previous rules.

Let's call α the common prefix. We implemented the left-factoring algorithm as follows: For rules sharing a common prefix, we replaced the parts that followed the common prefix by a common variable U, and then for each rule sharing the prefix, replaced it by a new rule where U produces the part that follows the common prefix. This yields the following new rules:

<InstList>	→	<Instruction> <InstList-Tail>
<InstList-Tail>	→	; <InstList>
	→	epsilon
<If>	→	if <Cond-p0> then <Code> <If-Tail>
<If-Tail>	→	endif
	→	else <Code> endif
<For>	→	for [VarName] from <ExprArith-p0> <For-Tail>
<For-Tail>	→	to <ExprArith-p0> do <Code> done
	→	by <ExprArith-p0> to <ExprArith-p0> do <Code> done

1.5 Resulting grammar

The final unambiguous LL(1) grammar is as follows:

[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

Because the objective is to build a LL(1) parser, which is a predictive parser, one does need to design the predictive features of the parser. Given a variable on the stack, let's say <Code>, the parser is not supposed to know which rule to apply between the following two rules:

<Program>	→	begin <Code> end
<Code>	→	ϵ
	→	<InstList>

This is solved by looking at the next input token. If the latter is, let's say *a*, the code will contain instructions. As a result, rule <Code> → <InstList> should be applied. However, if the next input token is *end*, the code will be empty and rule <Code> → ϵ should be applied.

2.1 First sets

We will consider only look-aheads of one symbol at a time in the framework of this report. The *First*[1] set of a symbol is the **set of first terminals** that can be derived from this symbol using different rules where the left-hand part consists in this symbol. Let's adapt the definition of first sets for LL(1) grammars:

$$First^1(A) = \{b \in T | A \Rightarrow^*_{Imp} bx\} \tag{4}$$

The first set of a terminal is the terminal itself. Indeed, with a terminal on top of the stack, there must necessarily be a **match** with the next input token: the two terminals must be equals. As a result, the only first token that can be derived from the terminal is the terminal itself.

This is more complex for variables. Let's take the previous example again. The first set of <Code> is composed both of *end* token and the first set of <InstList>. Because the second rule of the example does not produce any terminal but ϵ, the next symbol to be produced is the one that **follows in the parent production rule**. This is the purpose of **follow sets**. Regarding the third rule, <InstList> is a variable (which does not produce a token by itself), so one has to explore its first set before to be able to find first terminals.

The algorithm that computes first sets has been implemented as a **fixed-point iteration**: it starts by computing, for each terminal, its first set as the set containing only the terminal itself. Then it proceeds by reduction: for each rule producing sequences of symbols from which first sets are known, it computes the first set of the left-hand variable. The first set of a variable is then the **concatenation of first sets obtained** in all rules where this variable is in left-hand part, except for first sets containing ϵ (for the reason given earlier).

2.2 Follow sets

Because the right-hand part if a rule may produce ϵ, the first set is **not sufficient** to predict which rule to use. Let's consider rule $A \rightarrow w$, where the first set of w is the empty word. On that case, one can not infer the first terminal to be produced after A has been pushed on the stack. Actually, this terminal comes after A. So we define the follow set of A as the set of terminals such that there exists a derivation from the root that gives AaB, where a is a terminal.

Let's take our first example again. We get that $Follow^1(< Code >)$ is { *end* } because the only rule where <Code> is on right-hand side contains a *end* right after <Code>. This implies that, when <Code> $\rightarrow \epsilon$ is applied, the terminal that follows is *end*. We must build the action table is such a way that the parser becomes able to predict that rule <Code> $\rightarrow \epsilon$ must be used, only by seeing *end* as the next input token (look-ahead).

2.3 $First^1$ and $Follow^1$ sets for Imp

Symbol A	$First^1(A)$	$Follow^1(A)$
`<If>`	`if`	`else end endif ; done`
`<For>`	`for`	`else end endif ; done`
`<Atom>`	`[VarName] ( - [Number]`	`<= <> or ) * + do then - done / else and by end endif ; to < = > >=`
`<Code>`	`epsilon [VarName] print read for while if`	`else end endif done`
`<Comp>`	`<= <> < = > >=`	`[VarName] ( - [Number]`
`<Read>`	`read`	`else end endif ; done`

<Op-p0>	+ -	[VarName] (- [Number]
<Op-p1>	* /	[VarName] (- [Number]
<Print>	print	else end endif ; done
<While>	while	else end endif ; done
<Assign>	[VarName]	else end endif ; done
<Cond-p0>	[VarName] not (- [Number]	do then
<Cond-p1>	[VarName] not (- [Number]	or do then
<Cond-p2>	[VarName] not (- [Number]	or and do then
<If-Tail>	else endif	else end endif ; done
<Program>	begin	epsilon
<For-Tail>	by to	else end endif ; done
<InstList>	[VarName] print read for while if	else end endif done
<Cond-p0-i>	[VarName] not (- [Number]	or do then
<Cond-p0-j>	epsilon or	do then
<Cond-p1-i>	[VarName] not (- [Number]	or and do then
<Cond-p1-j>	epsilon and	or do then
<SimpleCond>	[VarName] (- [Number]	or and do then
<Instruction>	[VarName] print read for while if	else end endif ; done
<ExprArith-p0>	[VarName] (- [Number]	<= <> or) do then done else and by end endif ; to < = > >=
<ExprArith-p1>	[VarName] (- [Number]	<= <> or) + do then - done else and by end endif ; to < = > >=
<InstList-Tail>	epsilon ;	else end endif done
<ExprArith-p0-i>	[VarName] (- [Number]	<= <> or) + do then - done else and by end endif ; to < = > >=
<ExprArith-p0-j>	epsilon + -	<= <> or) do then done else and by end endif ; to < = > >=
<ExprArith-p1-i>	[VarName] (- [Number]	<= <> or) * + do then - done / else and by end endif ; to < = > >=
<ExprArith-p1-j>	epsilon * /	<= <> or) + do then - done else and by end endif ; to < = > >=

3 Action table

To keep the table below concise, match and accept does not appear in the table below. Indeed, when the top of stack contains a terminal, either the terminal is the same that the one on input and a match is performed (or an accept if the terminal is "$"; either an error if the two terminals does not match. TODO: A-t-on explique la semantique de chaque action?

	(	)	*	+	-	/	;	<	=	>	:=	<=	<>	>=	by	do	if	or	to	and	end	for	not	done
<If>																	41							
<For>																						44		
<Atom>	16	15	15	15	17	15	15	15	15	15		15	15	15	15	15		15	15	15	15			15
<Code>																	2				2	2		2
<Comp>	34				34			33	29	31		32	34	30										
<Read>																								
<Op-p0>	19			18	19																			
<Op-p1>	21		20		21	21																		
<Print>																								
<While>																								
<Assign>																								
<Cond-p0>	51				51																		51	
<Cond-p1>	53				53																		53	
<Cond-p2>	27				27											27		27		27			26	
<If-Tail>							42														42			42
<Program>																								
<For-Tail>															46				45					
<InstList>																	38					38		
<Cond-p0-i>	23				23											23		23					23	
<Cond-p0-j>																52		22						
<Cond-p1-i>	25				25											25		25		25			25	
<Cond-p1-j>																54		54		24				
<SimpleCond>	28				28																			
<Instruction>							8										4				8	6		8
<ExprArith-p0>	47				47																			
<ExprArith-p1>	49				49																			
<InstList-Tail>							39														40			40
<ExprArith-p0-i>	11	11		11	11		11	11	11	11		11	11	11	11	11		11	11	11	11			11
<ExprArith-p0-j>		48		10	10		48	48	48	48		48	48	48	48	48		48	48	48	48			48
<ExprArith-p1-i>	13	13	13	13	13	13	13	13	13	13		13	13	13	13	13		13	13	13	13			13
<ExprArith-p1-j>		50	12	50	50	12	50	50	50	50		50	50	50	50	50		50	50	50	50			50

	else	from	read	then	begin	endif	print	while	[Number]	[VarName]	$	epsilon
`<If>`												
`<For>`												
`<Atom>`	15			15		15			15	14		15
`<Code>`	2		2			2	2	2		2		2
`<Comp>`									34	34		34
`<Read>`			37									
`<Op-p0>`									19	19		19
`<Op-p1>`									21	21		21
`<Print>`							36					
`<While>`								35				
`<Assign>`										9		
`<Cond-p0>`									51	51		
`<Cond-p1>`									53	53		
`<Cond-p2>`				27					27	27		27
`<If-Tail>`	43					42						42
`<Program>`					0							
`<For-Tail>`												
`<InstList>`			38				38	38		38		
`<Cond-p0-i>`				23					23	23		23
`<Cond-p0-j>`				52								52
`<Cond-p1-i>`				25					25	25		25
`<Cond-p1-j>`				54								54
`<SimpleCond>`									28	28		
`<Instruction>`	8		8			8	7	5		3		8
`<ExprArith-p0>`									47	47		
`<ExprArith-p1>`									49	49		
`<InstList-Tail>`	40					40						40
`<ExprArith-p0-i>`	11			11		11			11	11		11
`<ExprArith-p0-j>`	48			48		48						48
`<ExprArith-p1-i>`	13			13		13			13	13		13
`<ExprArith-p1-j>`	50			50		50						50

3.1 Imp grammar is now LL(1)

One is able to prove that Imp grammar is now LL(1), only by checking that there is no conflict in the action table. Two actions located in the same cell of the table would mean that, based on the same look-ahead token, there is two possible actions. When multiple actions can be performed without any additional information, the parser becomes **non-deterministic**, and eventually make **wrong guesses** about which rule to apply.

Since each cell of our action table contains at most one possible action, there can be only one rule to apply to produce any terminal from the input buffer. Since we use only **one symbol of look-ahead**, we can conclude that the grammar is LL(1).

4 Implementation of the LL(1) parser

4.1 Theoretical considerations

The objective is to implement a **recursive descent LL(1) 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:

$$P_{Imp} = (\{q\}, T, V \cup T, \delta, q, \$, \phi) \tag{5}$$

- Because P_{Imp} accepts the grammar **by empty stack** and not by accepting state, it only requires the initial state to work. Let's call this state q. The set of states is thus {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.

We implemented the parser using a stack instead of hard coding each rule as a unique Java method. Instead of making **explicit recursive calls to functions**, each symbol that composes the right-hand part of the current rule is pushed on the stack. This yields the same results than what we could get after implementing a

4.2 Automating the process, but not too much

We think that 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.

4.3 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.

4.4 Action table

Class *ActionTable* has been implemented as a Map, 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. The action table for the imp language is given below. Strictly negative integers represent other types of actions: -1 stands for **match**, -2 stands for **accept** and -3 stands for **error**.

In source code, *LL1Parser* class contains a stack, a reference to the action table, and the root of the tree. Also, it possesses helper methods to export the parse to txt, javascript and latex.

We implemented our parser and parse tree as follows:

- Push the $ symbol on the stack and append the $ symbol to the input sequence of tokens. Then, push the start symbol `<Program>` on the stack.
- Look at the top of the stack, as well as to the next input token. Then combine the two symbols in a list and use it as a key to the action table. The action table returns an integer value that represents the action to apply. If the value is positive, then the parser **produces**. If the value is -1, then the parser does a **match**. Finally if the value is -2, it means that both symbols are $, and that the sequence of tokens has been **accepted by empty stack**. In all other cases, it produces a **syntax error**. Set the current node as the top of the stack.
- In case of a **produce**, use the integer value as the identifier of the rule to apply, then push all the symbols from the right-hand side of the rule on the stack. Also, push a symbol only if it is not ϵ. Create a new node containing the symbol and add it to the children list of the current node.
- In case of a **match**, pop the stack and look at the next token from the inputs.
- Repeat from second step, until an error occurs or the sequence of tokens gets accepted.

5 Parse tree

The parse tree is obviously built during parsing. Each *Node* object is composed of a grammar symbol that represents it, an identifier, a parent *Node* object, and a list of children. Because children are **not created in**

the right order, we implemented a *NodeComparator* to be able to compare nodes by identifiers. Every time a new child is added to a parent node, the latter sorts its children list using the *NodeComparator*. Once the tree is built, we compile it to Javascript to be able to display it in the browser.

6 How to use the parser

```
Arguments :
        (1) --ru <grammar file> -o <grammar output file>
        (2) --ll <grammar file> -o <grammar output file>
        (3) --at <ll1 unambiguous grammar file> -o <latex output file>
        (4) <ll1 unambiguous grammar file> <code> [-o <parse tree output file>]
(1) remove useless
(2) left factorization and removing of left recursion
(3) print action table
(4) save parse tree and output the rule used}
```

TODO: todo

7 Visualize parse trees

Because comparing rule identifiers from the command line to rules from a list can be tedious, **we recommand you to use the parse tree**. We used Go.js for visualization. To see the parse tree of the last Imp program that has been parsed, just open *more/trees/tree.html* in your browser (you will need an internet connection for that). It is pretty straightforward to compare your input Imp program with the parse tree since all terminals are aligned.