Introduction to language theory and compiling
Project – Part 2

Antoine Passemiers
Alexis Reynouard

November 25, 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 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 accessed using composed rules from grammar G.

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

All variables are accessible.

1.2.2 Unproductive variables

A rule is productive if all symbols in the right-hand part of this rule 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:

$$<\text{ExprArithP1}> \rightarrow <\text{ExprArthP1}> + <\text{ExprArithP2}> \tag{1.1}$$

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:

$$A \Rightarrow_G^* w \text{ where } w \in T^* \tag{1.2}$$

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:

- TODO

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

All variables are productive.

1.3 Removing ambiguity

1.3.1 Operator priority

<CondP1> → <CondP1> and <CondP2>
→ <CondP2>
<CondP2> → <CondP2> or <CondP3>
→ <CondP3>
<CondP3> → not <SimpleCond>
→ <SimpleCond>

<Assign> → [VarName] := <ExprArithP1>
<SimpleCond> → <ExprArithmP1> <Comp> <ExprArithP1>
→ TODO ???
<ExprArithP1> → <ExprArithP1> + <ExprArithP2>
→ <ExprArithP1> - <ExprArithP2>
→ <ExprArithP2>
<ExprArithP2> → <ExprArithP2> * <ExprArithP3>
→ <ExprArithP2> / <ExprArithP3>
→ <ExprArithP3>
<ExprArithP3> → [VarName]
→ [number]
→ (<ExprArithP3>)
→ - <ExprArithP3>

1.3.2 Operator associativity

1.4 Removing left-recursion and applying factorization

1.4.1 Left-recursion

1.4.2 Factorization

<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> → ϵ
[3] <> → <InstList>
[4] <InstList> → <Instruction>
[5] <> → <Instruction> `;` <InstList>
[6] <Instruction> → <Assign>
[7] <> → <If>
[8] <> → <While>
[9] <> → <For>
[10] <> → <Print>
[11] <> → <Read>
[12] <Assign> → `[VarName]` `:=` <ExprArithP1>
[13] <If> → `if` <Cond> `then` <Code> <IfTail>
[14] <IfTail> → `endif`
[15] <> → `else` <Code> `endif`
[16] <For> → `for [VarName] from` <ExprArith> <ForTail>
[17] <ForTail> → <ExprArith> `to` <ExprArith> `do` <Code> `done`
[18] <> → <ExprArith> `do` <Code> `done`
[19] <Comp> → `=`
[20] <> → `>=`
[21] <> → `>`
[22] <> → `<=`
[23] <> → `<`
[24] <> → `<>`
[25] <While> → `while` <Cond> `do` <Code> `done`
[26] <Print> → `print ( [VarName] )`
[27] <Read> → `read ( [VarName] )`