Introduction to language theory and compiling
Project – Part 2

Antoine Passemiers
Alexis Reynouard

November 21, 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 considering a $LL(1)$ parser, 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.

1.1 Removing useless rules and variables

1.1.1 Unreachable variables

Unreachable variables are variables 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.

1.1.2 Unproductive variables

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.

1.2 Removing ambiguity

1.2.1 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>

1.2.2 Operator associativity

1.3 Removing left-recursion and applying factorization

1.3.1 Left-recursion

1.3.2 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

1.4 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] )`