Université libre de Bruxelles

info-f-403

Impementation of a parser for the Imp language

Stéphane Sercu

November 27th, 2017

1 Introduction

This report describes the modification made on the grammar of the Imp language in order to implement an LL(1) parser. The resulting parser makes use of the lexical analyzer (and thus, of the symbols and lexical units) developed during the first part of this project[1].

The report starts by describing the different steps and manipulations applied on the grammar that lead to the final LL(1) grammar. Then, it describes how the action table of the parser was built. Finally, it presents the implementation.

2 Transformation of the Imp grammar

This section explains the manipulations applied on the initial Imp grammar to

1. make sure there aren't any unreachable and/or unproductive variables
2. make it unambiguous
3. remove left-recursion and factor it.

2.1 Removing unproductive/unreachable variables

Since the removal of unproductive variables can lead to new unreachable ones, the first thing to do is find and remove the the unproductive variables.

To do so, the algorithm presented on the page 108 of the notes for this course was used to find the productive variables with the intend to deduce the ones that aren't productive and remove from the production rules the ones that contain those unproductive variables.

However, it appeared that all the symbols are productive.

The unreachable variables were checked similarly, using the algorithm presented on the page 109 in the notes. Though, this showed that there were no unreachable variables either.

2.2 Handling the priority and associativity of the operators

The problem of the priority and the associativity of the operators was addressed through the introduction of new variables. This way, an arithmetic expression (<ExprArith>) is managed as a sum/difference of a product (<Prod>) which is itself managed as a product/division of atomic expressions (<Atom>) (which can be either constants, variables, other expressions or opposites of atomic expressions. This hierarchy reflects perfectly the priorities of the operations.

[1]The lexical analyzer was slightly modified to reject the numbers starting with 0.

<ExprArith>	→ <ExprArith> `+` <Prod>
	→ <ExprArith> `-` <Prod>
	→ <Prod>
<Prod>	→ <Prod> `*` <Atom>
	→ <Prod> `/` <Atom>
	→ <Atom>
<Atom>	→ [VarName]
	→ [Number]
	→ `-`<Atom>
	→ `(` <ExprArith> `)`
<Cond>	→ <Cond> `or` <BoolAnd>
	→ <BoolAnd>
<BoolAnd>	→ <BoolAnd> `and` <BoolAtom>
	→ <BoolAtom>
<BoolAtom>	→ `not` <BoolAtom>
	→ <ExprArith> <Comp> <ExprArith>

Table 1: Modified chuck of the grammar taking care of the priority & associativity of the operations.

The same concept stands for the boolean operators `or`, `and` and `not` through the variables <Cond>, <BoolAnd> and <BoolAtom>.
The chunk of grammar affected by those changes is presented in table 1.

2.3 Removing left-recursion & Factoring

The algorithms exposed during the 5th practical session of this course were used to remove the left-recursion and apply the factorization.

Those two last manipulations lead to the final grammar used by the parser and presented in table 2.

3 Building the action table

This section describes how the action table was built. It starts by showing the calculations of the Firsts & Follows and then presents the action table.

3.1 Firsts Follows

To build the action table, the $\text{First}^1(\alpha)$ of the right-hand side of each construction rules $(A \rightarrow \alpha)$ of the grammar are needed. Those are presented, with the detail of their computations, in the table 3.

Additionnaly, each time ϵ apears in the $\text{First}^1(\alpha)$, the Follow^1 of the left-hand side of the corresponding construction rule is needed. Those Follows are listed in the table 4.

3.2 Action table

// To actually build the action table, the algorithm presented in the 6th practical session was manually executed. The result is presented in table 5.

(1)	<Program>	→ `begin` <Code> `end`
(2)	<Code>	→ ϵ
(3)		→ <InstList>
(4)	<InstList>	→ <Instruction> <InstListSuffix>
(5)	<InstListSuffix>	→ ϵ
(6)		→ `;` <InstList>
(7)	<Instruction>	→ <Assign>
(8)		→ <If>
(9)		→ <While>
(10)		→ <For>
(11)		→ <Print>
(12)		→ <Read>
(13)	<Assign>	→ [VarName] `:=` <ExprArith>
(14)	<ExprArith>	→ <EA1><EA2>
(15)	<EA1>	→ <Prod>
(16)	<EA2>	→ `+` <Prod><EA2>
(17)		→ `-` <Prod><EA2>
(18)		→ ϵ
(19)	<Prod>	→ <P1><P2>
(20)	<P1>	→ <Atom>
(21)	<P2>	→ `*` <Atom><P2>
(22)		→ `/` <Atom><P2>
(23)		→ ϵ
(24)	<Atom>	→ [VarName]
(25)		→ [Number]
(26)		→ `-`<Atom>
(27)		→ `(` <ExprArith> `)`
(28)	<Cond>	→ <C1><C2>
(29)	<C1>	→ <BoolAnd>
(30)	<C2>	→ `or` <BoolAnd><C2>
(31)		→ ϵ
(32)	<BoolAnd>	→ <BA1><BA2>
(33)	<BA1>	→ <BoolAtom>
(34)	<BA2>	→ `and` <BoolAtom><BA2>
(35)		→ ϵ
(36)	<BoolAtom>	→ `not` <BoolAtom>
(37)		→ <ExprArith> <Comp> <ExprArith>
(38)	<If>	→ `if` <Cond> `then` <Code> <IfSuffix>
(39)	<IfSuffix>	→ `endif`
(40)		→ `else` <Code> `endif`
(41)	<Comp>	→ `=`
(42)		→ `>=`
(43)		→ `>`
(44)		→ `<=`
(45)		→ `<`
(46)		→ `<>`
(47)	<While>	→ `while` <Cond> `do` <Code> `done`
(48)	<For>	→ `for` [VarName] `from` <ExprArith> <ForSuffix>
(49)	<ForSuffix>	→ `by` <ExprArith> `to` <ExprArith> `do` <Code> `done`
(50)		→ `to` <ExprArith> `do` <Code> `done`
(51)	<Print>	→ `print(`[VarName]`)`
(52)	<Read>	→ `read(`[VarName]`)`

Table 2: Final grammar

$A \to \alpha$	First(α)
(1)	First(begin) = begin
(2)	First(ϵ) = ϵ
(3)	First(<InstList>) = First(<Instruction>) = (see (4)) = [VarName] if while for print read
(4)	First(<Instruction>) = First(<Assign>) ∪ First(<If>) ∪ First(<While>) ∪ First(<For>) ∪ First(<Print>) ∪ First(<Read>) = [VarName] if while for print read
(5)	ϵ
(6)	First(;) = ;
(7)	First(<Assign>) = [VarName]
(8)	First(<If>) = if
(9)	First(<While>) = while
(10)	First(<For>) = for
(11)	First(<Print>) = print
(12)	First(<Read>) = read
(13)	First([VarName]) = [VarName]
(14)	First(<EA1>) = First(<Prod>) = First(<P1>) = First(<Atom>) = [VarName] [Number] - (
(15)	First(<Prod>) = (see (14)) = [VarName] [Number] - (
(16)	+
(17)	-
(18)	ϵ
(19)	First(<P1>) = (see (14)) = [VarName] [Number] - (
(20)	First(<Atom>) = [VarName] [Number] - (
(21)	*
(22)	/
(23)	ϵ
(24)	[VarName]
(25)	[Number]
(26)	-
(27)	(
(28)	First(<C1>) = First(<BoolAnd>) = First(<BA1>) = First(<BoolAtom>) = not ∪ First(<ExprArith>) = (see (37)) = not [VarName] [Number] - (
(29)	First(<BoolAnd>) = (see (28)) = not [VarName] [Number] - (
(30)	or
(31)	ϵ
(32)	First(<BA1>) = (see (28)) = not [VarName] [Number] - (
(33)	First(<BoolAtom>) = (see (28)) = not [VarName] [Number] - (
(34)	and
(35)	ϵ
(36)	not
(37)	First(<ExprArith>) = First(<EA1>) = (see (14)) = [VarName] [Number] - (
(38)	if
(39)	endif
(40)	else
(41)	=
(42)	>=
(43)	>
(44)	<=
(45)	<
(46)	<>
(47)	while
(48)	for
(49)	by
(50)	to
(51)	print
(52)	read

Table 3: Required Firsts

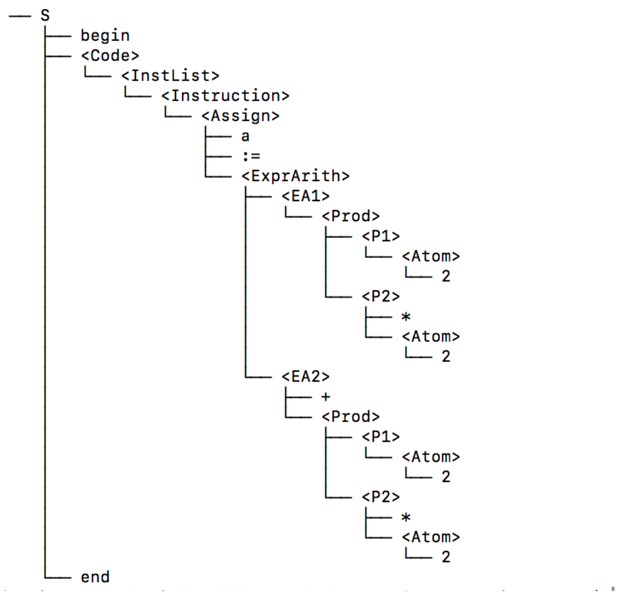
$A \to \alpha$	Follow(A)
(2)	Follow(<Code>) = end done endif ∪ First(<IfSuffix>) = end done endif else
(5)	Follow(<InstListSuffix>) = Follow(<InstList>) = Follow(<Code>) = (see (2)) = end done endif else
(18)	Follow(<EA2>) = Follow(<ExprArith>) =) do to ∪ Follow(<Assign>) ∪ First(<Comp>) ∪ Follow(<BoolAtom>) ∪ First(<ForSuffix>) = (see (*) and (**)) =) do to by ; end done endif else = >= > <= < <> or then and
(23)	Follow(<P2>) = Follow(<Prod>) = First(<EA2>) ∪ Follow(<EA2>) ∪ Follow(<EA1>) = First(<EA2>) ∪ Follow(<EA2>) = (see (18)) = + -) do to by ; end done endif else = >= > <= < <> or then and
(31)	Follow(<C2>) = Follow(<Cond>) = then do
(35)	Follow(<BA2>) = Follow(<BoolAnd>) = First(<C2>) ∪ Follow(<C2>) ∪ Follow(<C1>) = or ∪ Follow(<C2>) = (see (31)) = or then do
(*)	Follow(<Assign>) = Follow(<Instruction>) = First(<InstListSuffix>) ∪ Follow(<InstList>) = ; ∪ Follow(<Code>) ∪ Follow(<InstListSuffix>) = ; endif else done end
(**)	Follow(<BA2>) = Follow(<BoolAnd>) = (see (35)) = or then do

Table 4: Required Follows

	begin	end	;	:=	-	*	/	+	(	)	or	and	not	if	then	endif	else	=	>=	>	<=	<	<>
<Program>	1																						
<Code>		2												3		2	2						
<InstList>														4									
<InstListSuffix>		5	6													5	5						
<Instruction>														8									
<Assign>																							
<ExprArith>					14				14														
<EA1>					15				15														
<EA2>		18	18		17			16		18	18	18			18	18	18	18	18	18	18	18	18
<Prod>					19				19														
<P1>					20				20														
<P2>		23	23		23	21	22	23		23	23	23			23	23	23	23	23	23	23	23	23
<Atom>					26				27														
<Cond>					28				28				28										
<C1>					29				29				29										
<C2>											30				31								
<BoolAnd>					32				32				32										
<BA1>					33				33				33										
<BA2>											35	34			35								
<BoolAtom>					37				37				36										
<If>														38									
<IfSuffix>																39	40						
<Comp>																		41	42	43	44	45	46
<While>																							
<For>																							
<ForSuffix>																							
<Print>																							
<Read>																							

Table 5: Action table

	while	do	done	for	to	from	by	print	read	[VarName]	[Number]
<Program>											
<Code>	3		2	3				3	3	3	
<InstList>	4			4				4	4	4	
<InstListSuffix>			5								
<Instruction>	9			10				11	12	7	
<Assign>										13	
<ExprArith>										14	14
<EA1>										15	15
<EA2>		18	18		18		18				
<Prod>										19	19
<P1>										20	20
<P2>		23	23		23		23				
<Atom>										24	25
<Cond>										28	28
<C1>										29	29
<C2>		31									
<BoolAnd>										32	32
<BA1>										33	33
<BA2>		35									
<BoolAtom>										37	37
<If>											
<IfSuffix>											
<Comp>											
<While>	47										
<For>				48							
<ForSuffix>					50		49				
<Print>								51			
<Read>									52		



```
── S
   ├── begin
   ├── <Code>
   │   └── <InstList>
   │       └── <Instruction>
   │           └── <Assign>
   │               ├── a
   │               ├── :=
   │               └── <ExprArith>
   │                   ├── <EA1>
   │                   │   └── <Prod>
   │                   │       ├── <P1>
   │                   │       │   └── <Atom>
   │                   │       │       └── 2
   │                   │       └── <P2>
   │                   │           ├── *
   │                   │           └── <Atom>
   │                   │               └── 2
   │                   └── <EA2>
   │                       ├── +
   │                       └── <Prod>
   │                           ├── <P1>
   │                           │   └── <Atom>
   │                           │       └── 2
   │                           └── <P2>
   │                               ├── *
   │                               └── <Atom>
   │                                   └── 2
   └── end
```

Figure 1: An example of tree produced by the parser.

4 Implementing the parser

As said in the introduction, the parser uses the lexical analyzer built during the first part of this project. It was slightly improved by deprecating the numbers starting with 0.

4.1 The parser

The class `Parser` is pretty simple. It retrieves the set of symbols from the lexer and then go through them one by one (using its `readNextSymbol()` method). It can also look at the next symbol (without consuming it) using the `getNextSymbol()` method.

There is one method for each variable. Those methods are called recursively according to the current and the next symbol (in accordance with the action table described earlier).

Each call to one of those methods adds a node to the parse tree and a rule id in the derivation sequence.

4.2 Parse tree

Beside the derivation sequence, the parser is also capable of building the parse tree. It uses the `Java-Console-Tree-Builder`[2] library to draw it in the console.

The figure 1 shows, as an example, the the piece of art resulting from the parsing of the source file `symetricExpression.imp`[3].

5 Tests

The parser was tested using valid and invalid Imp codes. Each "functionality" of the language was tested as independently as possible, starting with basic and then more complex codes. The derivation sequence and the parsing tree of each one of the codes was analyzed and compared to the expected result.

In particular:

- The parsing of arithmetic expressions was tested using valid and invalid, simple and more complex arithmetic expressions. Those tests were focused on the priorities of the operation.

[2] `https://github.com/nathanielove/Java-Console-Tree-Builder`, last visit: 27 Nov. 2017
[3] The content of this program is a simple assignation: `a := 2*2+2*2`

- The parsing of if, for and while blocks was tested independently with each possible configuration.
- Invalid codes were tried to test the coherence of the error messages.

6 Conclusion

During the second part of this project, an LL(1) parser, based on an unambiguous and factorized grammar, was successfully implemented. It's able to print the derivation sequence and a parser tree.