Introduction to language theory and compiling Project – Part 2

PARDON Mathieu DIAZ Y SUAREZ Esteban

November 2021

1 Introduction

The aim of this project is to implement a parser for the language alCOl with the help of the scanner we've already implemented in another project. To do so, we had to modify the given gramar. We also had to implement the action table of the language.

2 Modified grammar

The grammar contained no unproductive variable nor unreachable symbol so no changes had to be made at this level. However, changes have been made to take into account arithmetic operators priority (the rule $< ExprArith > \rightarrow < ExprArith >< Op >< ExprArith >$ did not allow to consider a difference between priorities). Changes have also been made to remove left-recursion of the rules deriving symbol *ExprArith*. Finally, derivation of *if* derivation rule has been left factorized to avoid having two rules deriving *if* with same prefix on the right hand.

Resulting grammar is presented in the first listing.

Listing 1: Modified AlCOl grammar

1	$<Program>$	$\rightarrow$	$begin\ <Code>\ end$
2	$<Code>$	$\rightarrow$	ε
3		$\rightarrow$	$<InstList>$
4	$<InstList>$	$\rightarrow$	$<Instruction>\ <InstListEnd>$
5	$<InstListEnd>$	$\rightarrow$	ε
6		$\rightarrow$	$;\ <InstList>$
7	$<Instruction>$	$\rightarrow$	$<Assign>$
8		$\rightarrow$	$<If>$
9		$\rightarrow$	$<While>$
10		$\rightarrow$	$<For>$
11		$\rightarrow$	$<Print>$
12		$\rightarrow$	$<Read>$
13	$<Assign>$	$\rightarrow$	$[VarName]\ :=\ <ExprArith>$
14	$<ExprArith>$	$\rightarrow$	$<Prod><ExprArith'>$
15	$<ExprArith'>$	$\rightarrow$	$+<Prod><ExprArith'>$
16		$\rightarrow$	$-<Prod><ExprArith'>$
17		$\rightarrow$	ε

18	*<Prod>*	$\rightarrow$	*<Atom><Prod'>*
19	*<Prod'>*	$\rightarrow$	**<Atom><Prod'>*
20		$\rightarrow$	*/<Atom><Prod'>*
21		$\rightarrow$	ε
22	*<Atom>*	$\rightarrow$	*−<Atom>*
23		$\rightarrow$	*[Number]*
24		$\rightarrow$	*[VarName]*
25		$\rightarrow$	*(<ExprArith>)*
26	*<If>*	$\rightarrow$	*if <Cond> then <Code> <ElseClause> endif*
27	*<ElseClause>*	$\rightarrow$	*else <Code>*
28		$\rightarrow$	ε
29	*<Cond>*	$\rightarrow$	*not <Cond>*
30		$\rightarrow$	*<SimpleCond>*
31	*<SimpleCond>*	$\rightarrow$	*<ExprArith> <Comp> <ExprArith>*
32	*<Comp>*	$\rightarrow$	*=*
33		$\rightarrow$	*>*
34		$\rightarrow$	*<*
35	*<While>*	$\rightarrow$	*while <Cond> do <Code> endwhile*
36	*<For>*	$\rightarrow$	*for [VarName] from <ExprArith> by <ExprArith> to <ExprArith> d*
37	*<Print>*	$\rightarrow$	*print([VarName])*
38	*<Read>*	$\rightarrow$	*read([VarName])*

Grammar is LL(1) because left-recursion have been removed (derivation of *If* rules), no rules have common prefix in right-hand side (*If* rules also) and there is no rule that a look-ahead of the first element on the input couldn't resolve.

Since the classes of LL(1) and strong LL(1) grammars coincide, we can prove the above grammar is LL(1) by applying the test verifying it is strong LL(1). Test available here once it will be finished.

3 Action table

The action table is a two-dimensional table. The elements indexed in the lines are the potentials tops of stacks, the elements indexed in the columns are the potential look-ahead. The cells contains the action of the grammar the parser must perform if it matchs the look-ahead and the corresponding top of stack.
Bellow the action table helps us to know if we should match or accept terminals when they occur at the top of the stack.

aa	VARNAME	NUMBER	BEG	END	SEMICOLON	ASSIGN	LPAREN	RPAREN	MINUS	PLUS	TIMES	DIVIDE	IF	THEN	ENDIF	ELSE	NOT	EQUAL	GREATER	SMALLER	WHILE	DO	ENDWHILE	FOR	FROM	BY	TO	ENDFOR	PRINT	READ	END
Program			1																												
Code	3			2									3		2	2					3		2	3				2	3	3	
InstList	4												4								4			4					4	4	
InstListEnd				5	6										5	5							5					5			
Instruction	7												8								9			10					11	12	
Assign	13																														
ExprArith	14	14					14		14																						
ExprArith'				17	17			17	16	15					17	17		17	17	17			17			17		17			
Prod	18	18					18	18																							
Prod'				21	21			21	21	21	19	20			21	21		21	21	21			21			21		21			
Atom	24	23					25		22																						
If													26																		
ElseClause															28	27															
Cond	30	30					30		30								29														
SimpleCond	31	31					31		31																						
Comp																		32	33	34											
While																					35										
For																								36							
Print																													37		
Read																														38	
Read	M																														
Read		M																													
Read			M																												
Read				A																											
Read					M																										
Read						M																									
Read							M																								
Read								M																							
Read									M																						
Read										M																					
Read											M																				
Read												M																			
Read													M																		
Read														M																	
Read															M																
Read																M															
Read																	M														
Read																		M													
Read																			M												
Read																				M											
Read																					M										
Read																						M									
Read																							M								
Read																								M							
Read																									M						
Read																										M					
Read																											M				
Read																												M			
Read																													M		
Read																														M	
END																															M

4 Parser implementation

We used three Hashmaps: one to convert the rule's index in the grammar to its derivations, a second one to convert the variables to their line's index in the action table and a last one to convert the terminal to the column's index in the action table.
We implemented the parser using a recursive descent function. The tokens from the scanner where stored in a list.
We used a match Table and another table that tells us if we should match or accept terminals when they occur at the top of the stack as we already said in a previous chapter. As we can see in the action table, some tokens

5 Testing set

We tested our parser on different set of codes. The results were not the ones we expected. The small sets without conditions and loop work perfectly but not the ones with those two elements.

6 Non-implemented features

Only available datatype is Integer (no float, no strings...) PODs / containers
No function implementation
No exponentiation operator
We've not implemented the Parse Tree printing feature.