ECOLE POLYTECHNIQUE DE BRUXELLES

2023-2024

INFO-F403 - Introduction to Language Theory and Compiling

PASCALMAISPRESQUE
Project – Part 2

Sacha DUYNSLAEGER

Yen Nhi NGUYEN

Gilles GEERAERTS

Léonard BRICE

Mathieu SASSOLAS

November 2023

1 PASCALMAISPRESQUE - Project Part II

1.1 Introduction

A compiler for PASCALMAISPRESQUE is what this project aims to design and write. This simple imperative language's grammar has reserved keywords and lexical units.

This second part of the project is to write the parser of the PASCALMAISPRESQUE compiler started with the first part of the project.

1.2 Code Execution

To compile all the files necessary for running the code, simply do :

```
make
```

To run all the tests, simply do :

```
make tests
```

Then, if you want to run the parser with your own files, this can be done with two methods. To do so, you first need to enter the `dist/` directory.
The first method runs the code without creating a tex file representing the parse tree, and the second one creates a tex file representing the parse tree :

— `java -jar part2.jar sourceFile.pmp`

— `java -jar part2.jar -wt sourceFile.tex sourceFile.pmp`

The generated parse trees can be found in the directory : `More/LaTexTrees/`.

1.3 Modified grammar

Multiple transformations were applied to the PASCALMAISPRESQUE context-free grammar provided in the work description document : the elimination of useless symbols that do not result in any terminals, the removal of ambiguity to allow for only one derivation possible per input while respecting the priority and associativity of the operations, the left-recursion removal to make the grammar deterministic, meaning the parser will eventually always read a terminal, and the left-factorization to allow the parser to make a deterministic decision.

1.3.1 Removal of unproductive variables

First, we compute the set of productive symbols : Prod :{=, <, +, – (binary), *, /, if, else, and, or, begin, end, ..., while, print, read, [VarName], [Number], – (unary), (,), {, }, then, do, :=}.

Secondly, rule [34] from the given grammar tells that <*READ*> is productive too because its right-hand side is the only one to contain *read*. And we do it successively to distinguish which symbols are productive from the ones which are not.

In conclusion, the rule [9] from the given grammar contains an unproductive variable because <for> does not produce any terminal symbol.

1.3.2 Removal of unreachable variables

We start with the first symbol, which is <*Program*>. ⇒ Reach = {<*Program*>}
With rule [1], *begin*, <*Code*>, and *end* are also reachable.
⇒ Reach = {<*Program*>, *begin*, <*Code*>, *end*}
And we do it successively to distinguish which symbols are reachable from the ones which are not.

At the end, no unreachable variable was detected.

1.3.3 Non-ambiguous grammar

Secondly, to build a parser for the grammar (which is a deterministic program), we need to eliminate any ambiguity within the grammar. This can be achieved by modifying the grammar to ensure that only a single derivation tree is generated. Consequently, the consideration of operator priority and associativity becomes paramount in this process.

To do so, some intermediate variables were introduced in the grammar to enforce a hierarchy between the operators, and that results in moving elements down the derivation trees. An example is provided in the next section.

1.3.4 Removal of the left-recursion

Some of the rules from the given grammar contain left recursion : the variable from the left-hand side of the rule is the first symbol of the rule in its right-hand side, as seen in rule [25] from the project instructions. To remove it, the left recursion was turn into right recursion by introducing intermediate variables.

For example, the rules [25] to [28] from the grammar from the project instructions :

[25]	<Cond>	→	<Cond> and <Cond>
[26]		→	<Cond> or <Cond>
[27]		→	{ <Cond> }
[28]		→	<SimpleCond>

were transform by taking into account the priority of the AND over the priority of the OR and removing the left recursion with the introduction of <Cond'>, <CondInter> and <CondAND'> to obtain :

[30]	<Cond>	→	<CondAND><Cond'>
[31]	<Cond'>	→	or <CondAND><Cond'>
[32]		→	ϵ
[33]	<CondAND>	→	<CondInter><CondAND'>
[34]	<CondAND'>	→	and <CondInter><CondAND'>
[35]		→	ϵ
[36]	<CondInter>	→	{ <Cond>}
[37]		→	<SimpleCond>

This method was repeated until there were no more left recursion.

1.3.5 Left-factorisation

In the grammar from the work description document, there are some rules that have the same left-hand side and a common prefix in the right-hand side.

For example, <InstList> is here the common left-hand side and <Instruction> is the common prefix in the right-hand side.

[4]	<InstList>	→	<Instruction>
[5]		→	<Instruction> ... <InstList>

To factorize the common prefix, <InstFactorized> is a new state that was introduced.

[4]	<InstList>	→	<Instruction><InstFactorized>
[5]	<InstFactorized>	→	... <InstList>
[6]		→	ϵ

This method was repeated until there were no more factorisable rules.

1.3.6 Rules of the modified grammar

After applying all these modifications, here is the modified grammar :

[1]	<Program>	→	begin <Code>end
[2]	<Code>	→	ϵ
[3]		→	<InstList>
[4]	<InstList>	→	<Instruction><InstFactorized>
[5]	<InstFactorized>	→	... <InstList>
[6]		→	ϵ
[7]	<Instruction>	→	<Assign>
[8]		→	<If>
[9]		→	<While>
[10]		→	<Print>
[11]		→	<Read>
[12]		→	begin <InstList>end
[13]	<Assign>	→	[VarName] := <ExprArith>
[14]	<ExprArith>	→	<ExpGenProdDiv><EA'>
[15]	<EA'>	→	+ <ExpGenProdDiv><EA'>
[16]		→	- <ExpGenProdDiv><EA'>
[17]		→	ϵ
[18]	<ExpGenProdDiv>	→	<InterExp><ExpProdDiv>
[19]	<ExpProdDiv>	→	* <InterExp><ExpProdDiv>
[20]		→	/ <InterExp><ExpProdDiv>
[21]		→	ϵ
[22]	<InterExp>	→	<Atom>
[23]		→	(<ExprArith>)
[24]		→	- <InterExp>
[25]	<Atom>	→	[VarName]
[26]		→	[Number]
[27]	<If>	→	if <Cond>then <Instruction><IfFactorized>
[28]	<IfFactorized>	→	else <Instruction>
[29]		→	ϵ
[30]	<Cond>	→	<CondAND><Cond'>
[31]	<Cond'>	→	or <CondAND><Cond'>
[32]		→	ϵ
[33]	<CondAND>	→	<CondInter><CondAND'>
[34]	<CondAND'>	→	and <CondInter><CondAND'>
[35]		→	ϵ
[36]	<CondInter>	→	{ <Cond>}
[37]		→	<SimpleCond>
[38]	<SimpleCond>	→	<ExprArith><Comp><ExprArith>
[39]	<Comp>	→	=
[40]		→	<
[41]	<While>	→	while <Cond>do <Instruction>
[42]	<Print>	→	print ([VarName])
[43]	<Read>	→	read ([VarName])

TABLE 1.1: Modified grammar rules

1.4 Analysis of the grammar

The $FIRST^1$ and the $FOLLOW^1$ sets were first computed for all the variables to determine if the grammar is LL(1). The action table of an LL(1) parser for the transformed grammar was then build to write the parser.

NOTE : afterwards, since it is only necessary to check the $First^1$ and $FOLLOW^1$ sets to determine if the grammar is LL(1), these two will be noted by FIRST and FOLLOW respectively in the computations of sections 1.4.1, 1.4.2, and 1.4.3.

1.4.1 FIRST sets

For a given variable A, $FIRST^1$(A) is used to determine the set of all possible first terminal symbols that can be derived from each variable A of the grammar.

<Program> :	{begin}	
<Code> :	{Epsilon, FIRST(<InstList>)}	= {Epsilon, begin, [Varname], if, while, print, read }
<Instlist> :	{FIRST(<Instruction>)}	= {begin, [Varname], if, while, print, read}
<InstFactorized> :	{..., Epsilon}	
<Instruction> :	{begin, FIRST(<Assign>, <If>, <While>, <Print>, <Read>)}	= {begin, [Varname], if, while, print, read}
<Assign> :	{[VarName]}	
<ExprArith> :	{FIRST(<ExpGenProdDiv>)}	= {(, -, [Varname], [Number]} (- is - unary)
<EA'> :	{+, -, Epsilon} (- is – binary)	
<ExpGenProdDiv> :	{FIRST(<InterExp>)}	= {(, -, [Varname], [Number]} (- is - unary)
<ExpProdDiv> :	{*, /, Epsilon}	
<InterExp> :	{(, -, FIRST(<Atom>)}	= {(, -, [Varname], [Number]} (- is - unary)
<Atom> :	{[VarName], [Number]}	
<If> :	{if}	
<IfFactorized> :	{else, Epsilon}	
<Cond> :	{FIRST(<CondAND>)}	= {{, (, -, [Varname], [Number]} (- is - unary)
<Cond'> :	{or, Epsilon}	
<CondAND> :	{FIRST(<CondInter>)}	= {{, (, -, [Varname], [Number]} (- is - unary)
<CondAND'> :	{and, Epsilon}	
<CondInter> :	{{, FIRST(<SimpleCond>)}	= {{, (, -, [Varname], [Number]} (- is - unary)
<SimpleCond> :	{FIRST(<ExprArith>)}	= {(, -, [Varname], [Number]} (- is - unary)
<Comp> :	{=, <}	
<While> :	{while}	
<Print> :	{print}	
<Read> :	{read}	

TABLE 1.2: $FIRST^1 sets$

1.4.2 FOLLOW sets

For a given variable A, $FOLLOW^1$(A) is the set of all possible first terminal symbols that are generated after a word derived from A.

<Program> :	{$}	
<Code> :	{end}	
<Instlist> :	{end}	
<InstFactorized> :	{FOLLOW(<InstList>)}	= {end}
<Instruction> :	{FIRST(<InstFactorized>, <IfFactorized>), FOLLOW(<IfFactorized>, <While>)}	= {..., end, else}
<Assign> :	{FOLLOW(<Instruction>)}	= {..., end, else}
<ExprArith> :	{), FIRST(<Comp>), FOLLOW(<Assign>, <SimpleCond>)}	= {), =, <, ..., end, else, and, or, then, }, do}
<EA'> :	{FOLLOW(<ExprArith>)}	= {), =, <, ..., end, else, and, or, then, }, do}
<ExpGenProdDiv> :	{FIRST(<EA'>)}	= {), =, <, ..., end, else, +, -, and, or, then, }, do}
<ExpProdDiv> :	{FOLLOW(<ExpGenProdDiv>)}	= {), =, <, ..., end, else, +, -, and, or, then, }, do}
<InterExp> :	{FIRST(<ExpProdDiv>)}	= {), =, <, ..., end, else, *, /, +, -, and, or, then, }, do}
<Atom> :	{FOLLOW(<InterExp>)}	= {), =, <, ..., end, else, *, /, +, -, and, or, then, }, do}
<If> :	{FOLLOW(<Instruction>)}	= {..., end, else}
<IfFactorized> :	{FOLLOW(<If>)}	= {..., end, else}
<Cond> :	{then, }, do}	
<Cond'> :	{FOLLOW(<Cond>)}	= {then, }, do}
<CondAND> :	{FIRST(<Cond'>)}	= {or, then, }, do}
<CondAND'> :	{FOLLOW(<CondAND>)}	= {or, then, }, do}
<CondInter> :	{FIRST(<CondAnd'>)}	= {and, or, then, }, do}
<SimpleCond> :	{FOLLOW(<CondInter>)}	= {and, or, then, }, do}
<Comp> :	{FIRST(<ExprArith>)}	= {(, -, [Varname], [Number]} (- is - unary)
<While> :	{FOLLOW(<Instruction>)}	= {..., end, else}
<Print> :	{FOLLOW(<Instruction>)}	= {..., end, else}
<Read> :	{FOLLOW(<Instruction>)}	= {..., end, else}

TABLE 1.3: $FOLLOW^1 sets$

1.4.3 (Almost) LL(1) grammar and action table

With the implication that says that a grammar is not LL(1) if the grammar is not SLL(1), we can determine if the grammar is LL(1) by verifying if there is no common element for the first terminal symbol generated by all the different rules of the same variable.

Here is an example : for <Code>, the rules where it appears on the left-hand side are :

[2]	<Code>	→	ϵ
[3]		→	<InstList>

By looking at the FIRST and FOLLOW sets computed before,

$FIRST^1(\epsilon FOLLOW^1(<Code>))$ = {end}

$FIRST^1(<InstList> FOLLOW^1(<Code>))$ = {begin, [Varname], if, while, print, read}

$\Rightarrow FIRST^1(\epsilon FOLLOW^1(<Code>)) \cap FIRST^1(<InstList> FOLLOW^1(<Code>)) = \emptyset$

This implies that there is no common element for the first terminal generated by all the different rules containing the variable <Code> in the left-hand side.

And we do it successively for all the variables to know if the grammar is SLL(1), and thus LL(1).

Note : $First^1$ and $FOLLOW^1$ are noted FIRST and FOLLOW in the computations.

With the table 1.4 containing the computations steps, we see that the grammar is not SLL(1), and thus not LL(1) because of the terminal *else* from the rules [23] and [24] of the issued grammar (even with the modified grammar). When there are multiple nested conditions and only one *else* statement, it becomes confusing to know which condition the *else* statement is connected to.

NOTE : Because the action table was already built when rule 23 was modified, we followed the instructions given by Mr. Mathieu Sassolas and attached the *else* to the closest possible *if*.

NOTE : Another solution is to have an *else* (terminal) regardless of whether it is actually followed by an instruction.

Table 1.5 presents the action table from the modified grammar. Built based on all the relations between a given variable and a terminal (FIRST set and if there is an ϵ, also the FOLLOW set of the variables), the action table tells which rule to applied if the next symbol read is X.

<InstFactorized>	$FIRST(\epsilon FOLLOW(< InstFactorized >))$ = {end} $FIRST(... < InstList > FOLLOW(< InstFactorized >))$ = {...} $\{end\} \cap \{...\} = \emptyset$
<Instruction>	$FIRST(< Assign > FOLLOW(< Instruction >))$ = {[VarName]} $FIRST(< If > FOLLOW(< Instruction >))$ = {if} $FIRST(< While > FOLLOW(< Instruction >))$ = {while} $FIRST(< Print > FOLLOW(< Instruction >))$ = {print} $FIRST(< Read > FOLLOW(< Instruction >))$ = {read} $FIRST(begin < InstList > endFOLLOW(< Instruction >))$ = {begin} $\{[VarName]\} \cap \{if\} \cap \{while\} \cap \{print\} \cap \{read\} \cap \{begin\} = \emptyset$
<EA'>	$FIRST(+ < ExpGenProdDiv >< EA' > FOLLOW(< EA' >))$ = {+} $FIRST(- < ExpGenProdDiv >< EA' > FOLLOW(< EA' >))$ = {-} $FIRST(\epsilon FOLLOW(< EA' >))$ = {), =, <, . . ., end, else, and, or, then, }, do} $\{+\} \cap \{-\} \cap \{), =, <, ..., end, else, and, or, then, \}, do\} = \emptyset$
<ExpProdDiv>	$FIRST(* < InterExp >< ExpProdDiv > FOLLOW(< ExpProdDiv >))$ = {*} $FIRST(/ < InterExp >< ExpProdDiv > FOLLOW(< ExpProdDiv >))$ = {/} $FIRST(\epsilon FOLLOW(< ExpProdDiv >))$ = {), =, <, . . ., end, else, +, -, and, or, then, }, do} $\{*\} \cap \{/\} \cap \{), =, <, ..., end, else, +, -, and, or, then, \}, do\} = \emptyset$
<InterExp>	$FIRST(< Atom > FOLLOW(< InterExp >))$ = {[VarName], [Number]} $FIRST((< ExprArith >) FOLLOW(< InterExp >))$ = {(} $FIRST(- < InterExp > FOLLOW(< InterExp >))$ = {-} $\{[VarName], [Number]\} \cap \{(\} \cap \{-\} = \emptyset$
<Atom>	$FIRST([VarName] FOLLOW(< Atom >))$ = {[VarName]} $FIRST([Number] FOLLOW(< Atom >))$ = {[Number]} $\{[VarName]\} \cap \{[Number]\} = \emptyset$
<IfFactorized>	$FIRST(else < Instruction > FOLLOW(< IfFactorized >))$ = {else} $FIRST(\epsilon FOLLOW(< IfFactorized >))$ = {. . ., end, else} $\{else\} \cap \{..., end, else\} = \{else\}$
<Cond'>	$FIRST(or < CondAND >< Cond' > FOLLOW(< Cond' >))$ = {or} $FIRST(\epsilon FOLLOW(< Cond' >))$ = {then, }, do} $\{or\} \cap \{then, \}, do\} = \emptyset$
<CondAND'>	$FIRST(and < CondInter >< CondAND' > FOLLOW(< CondAND' >))$ = {and} $FIRST(\epsilon FOLLOW(< CondAND' >))$ = {or, then, }, do} $\{and\} \cap \{or, then, \}, do\} = \emptyset$
<CondInter>	$FIRST(\{< Cond >\} FOLLOW(< CondInter >))$ = {{} $FIRST(< SimpleCond > FOLLOW(< CondInter >))$ = {(, -, [Varname], [Number]} $\{\{\} \cap \{(, -, [Varname], [Number]\} = \emptyset$
<Comp>	$FIRST(= FOLLOW(< Comp >))$ = {=} $FIRST(< FOLLOW(< Comp >))$ = {<} $\{=\} \cap \{<\} = \emptyset$

TABLE 1.4: Computations to determine if the grammar is SLL(1)

	=	<	+	– (binary)	*	/	if	else	and	or	begin	end	…	while	print	read	[VarName]	[Number]	– (unary)	(	)	{	}	then	do	:=
<Program>											[1]															
<Code>							[3]				[3]	[2]		[3]	[3]	[3]	[3]									
<InstList>							[4]				[4]			[4]	[4]	[4]	[4]									
<InstFactorized>												[6]	[5]													
<Instruction>							[8]				[12]			[9]	[10]	[11]	[7]									
<Assign>																	[13]									
<ExprArith>																	[14]	[14]	[14]	[14]						
<EA'>	[17]	[17]	[15]	[16]				[17]	[17]	[17]		[17]	[17]									[17]		[17]	[17]	[17]
<ExpGenProdDiv>																	[18]	[18]	[18]		[18]					
<ExpProdDiv>	[21]	[21]	[21]	[21]	[19]	[20]		[21]	[21]	[21]		[21]	[21]									[21]		[21]	[21]	[21]
<InterExp>																	[22]	[22]	[24]	[23]						
<Atom>																	[25]	[26]								
<If>							[27]																			
<IfFactorized>								[28]				[29]	[29]													
<Cond>																	[30]	[30]	[30]	[30]		[30]				
<Cond'>										[31]														[32]	[32]	[32]
<CondAND>																	[33]	[33]	[33]			[33]				
<CondAND'>									[34]	[35]														[35]	[35]	[35]
<CondInter>																	[37]	[37]	[37]	[37]		[36]				
<SimpleCond>																	[38]	[38]	[38]	[38]						
<Comp>	[39]	[40]																								
<While>														[41]												
<Print>															[42]											
<Read>																[43]										

TABLE 1.5: Action Table

1.5 Test Files

Aside from the Euclid.pmp file provided at the launch of the project, other input files were written to test different cases *(cfr section 2)* :

— **A file that does not respect the grammar.** *(cfr read_alone.pmp)*
This is a file that does not start with *begin*. An error message is printed in the terminal : "Syntax Error occurred when reading the token 'read' at line 2. Lexical Unit BEG was expected, but READ was found."

— **A list with 2 instructions.** *(cfr readNprintCODE.pmp)*
This file produces on stdout the sequence of rule numbers (according to the new modified grammar) that represents the leftmost derivation of the input. Its parse tree (fig. 2.1) is also build and written in a LaTex file in the subfile LaTexTrees, situated in the file More (when adding -wt readNprint.tex to the command).

— **The priority of the operation.** *(cfr calculus.pmp)*
The minus unary has a higher priority than the multiplication and the subtraction, and is thus below the * and the minus (binary) in the parse tree, as seen in the parse tree 2.3.

— **A nested *if* with an *else* statement for each *if*.** *(cfr nestedDoubleElse.pmp)*
With nested if, we expect that the first else should be linked to the second if, and the second else to the first if. This is the case, as seen in the parse tree 2.4.

— **A *while* with an *and*.** *(cfr condAnd.pmp)*
This test is to ensure that the execution code within the loop occurs solely when both conditions are satisfied. This is indeed the case, as shown by the parse tree 2.5.

2 Test files

2.1 read_alone.pmp

```
  read(a)...
** should return an error because it does not start with begin and doesn't
end with end
```

This file produces the output : "Syntax Error occurred when reading the token 'read' at line 2. Lexical Unit BEG was expected, but READ was found."

2.2 readNprintCODE.pmp

```
begin
  read(a)...
  print(a)
end
```

This file produces the output : 1 3 4 11 43 5 4 10 42 6

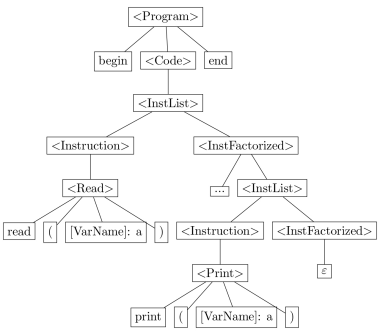

FIGURE 2.1: readNprintCODE.pmp parse tree

2.3 euclid.pmp

```
'' Euclid's algorithm ''
begin
  read(a)...
  read(b)...
  while 0 < b do
    begin
      c := b...
      while b < a+1 do       ** computation of modulo
        a := a-b...
      b := a...
      a := c
    end...
  print(a)
end
```

This file produces the output : 1 3 4 11 43 5 4 11 43 5 4 9 41 30 33 37 38 14 18 22 26 21 17 40 14 18 22 25 21 17 35 32 12 4 7 13 14 18 22 25 21 17 5 4 9 41 30 33 37 38 14 18 22 25 21 17 40 14 18 22 25 21 15 18 22 26 21 17 35 32 7 13 14 18 22 25 21 16 18 22 25 21 17 5 4 7 13 14 18 22 25 21 17 5 4 7 13 14 18 22 25 21 17 6 5 4 10 42 6

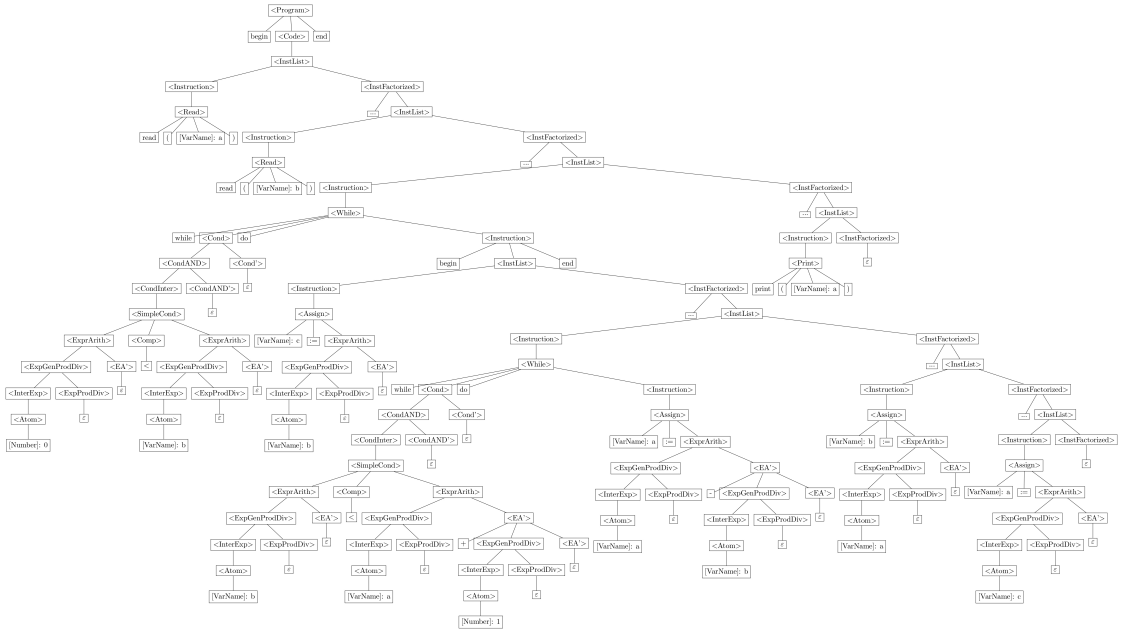

FIGURE 2.2: euclid.pmp parse tree

2.4 calculus.pmp

```
begin
   a := 1+1...
   b := 2 - 1...
   c := -1...
   d := 1*2...
   e := 1/2...
   f := 1*-2...
   g := -1 - -2
end
```

This file produces the output :

1 3 4 7 13 14 18 22 26 21 15 18 22 26 21 17 5 4 7 13 14 18 22 26 21 16 18 22 26 21 17 5 4 7 13 14 18 24 22 26 21 17 5 4 7 13 14 18 22 26 19 22 26 21 17 5 4 7 13 14 18 22 26 20 22 26 21 17 5 4 7 13 14 18 22 26 19 24 22 26 21 17 5 4 7 13 14 18 24 22 26 21 16 18 24 22 26 21 17 6

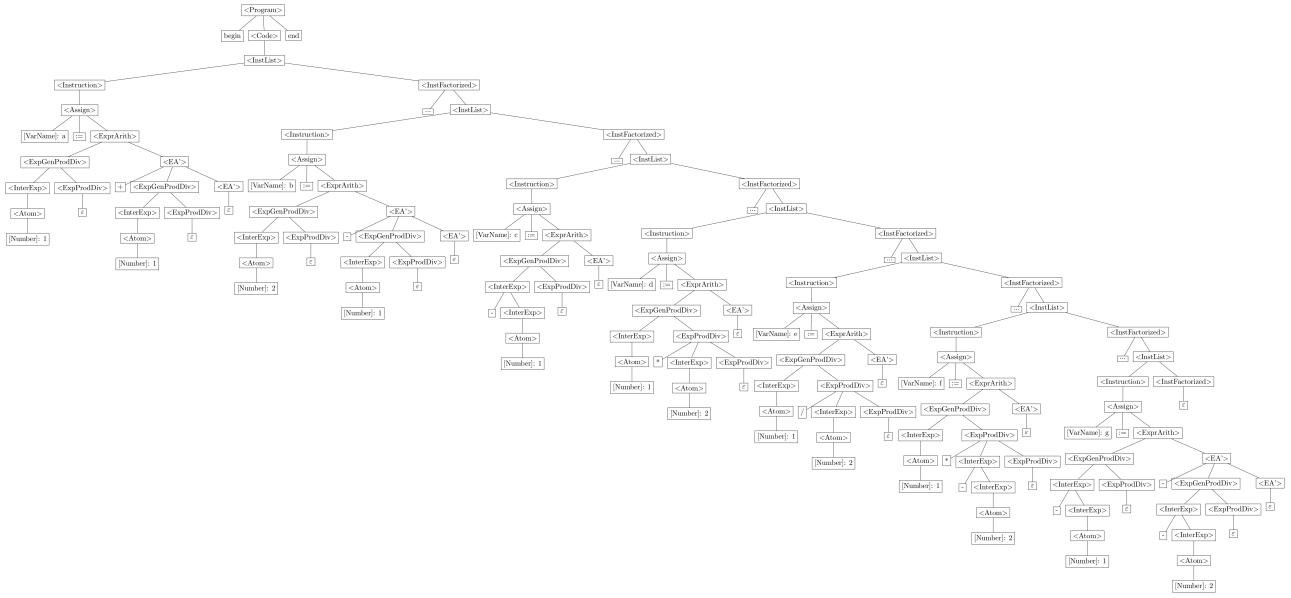

FIGURE 2.3: calculus.pmp parse tree

2.5 nestedDoubleElse.pmp

```
begin
    read(a)...
    if 1 < a then
```

```
        begin
            if 2 < a then
                print(a)
            else
                a := 2
        end
    else
        a:= 3...
    print(a)
end

** Here, because the action table was built on an ambiguous/erroneous rule,
** the first else should be linked to the second if,
** and the second else to the first if
```

This file produces the output : 1 3 4 11 43 5 4 8 27 30 33 37 38 14 18 22 26 21 17 40 14 18 22 25 21 17 35 32 12 4 8 27 30 33 37 38 14 18 22 26 21 17 40 14 18 22 25 21 17 35 32 10 42 28 7 13 14 18 22 26 21 17 6 28 7 13 14 18 22 26 21 17 5 4 10 42 6

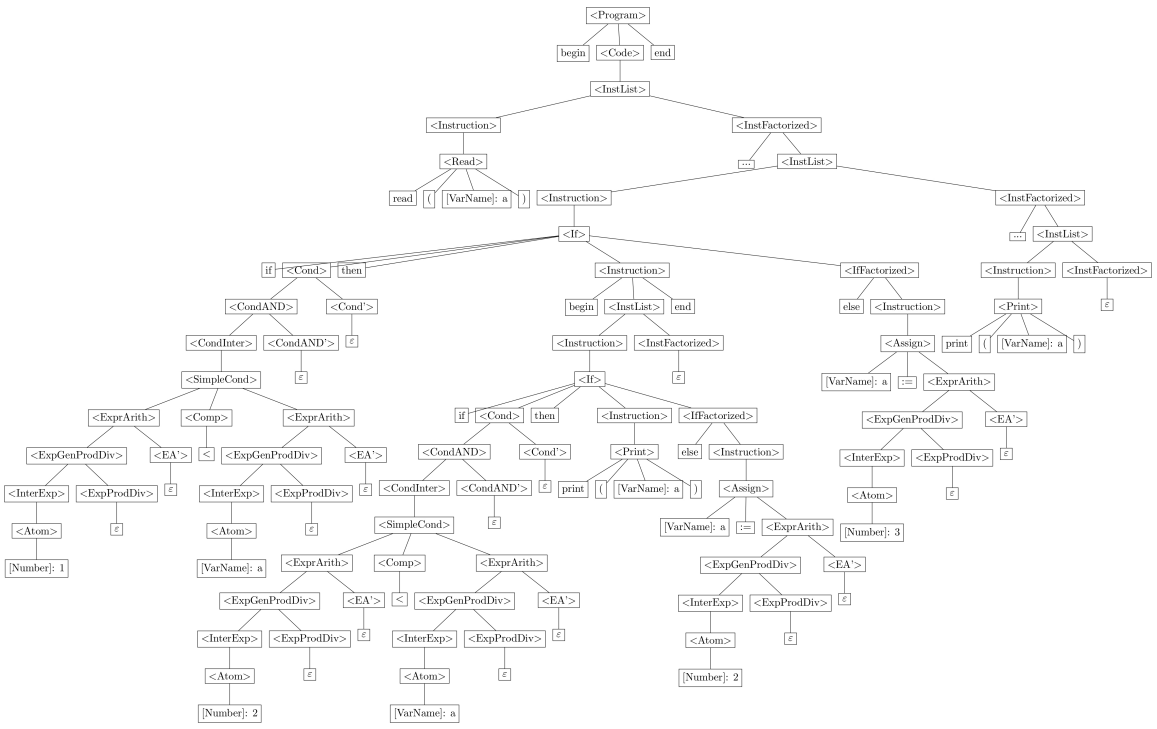

FIGURE 2.4: NestedDoubleElse.pmp parse tree

2.6 condAnd.pmp

```
begin
  read(a)...
  while 0 < a  and a < 5 do
    begin
      if {1 < a} then
        begin
          print(a)...
          b := a
        end
    end...
  print(a)
end
```

This file produces the output : 1 3 4 11 43 5 4 9 41 30 33 37 38 14 18 22 26 21 17 40 14 18 22 25 21 17 34 37 38 14 18 22 25 21 17 40 14 18 22 26 21 17 35 32 12 4 8 27 30 33 36 30 33 37 38 14 18 22 26 21 17 40 14 18 22 25 21 17 35 32 35 32 12 4 10 42 5 4 7 13 14 18 22 25 21 17 6 29 6 5 4 10 42 6

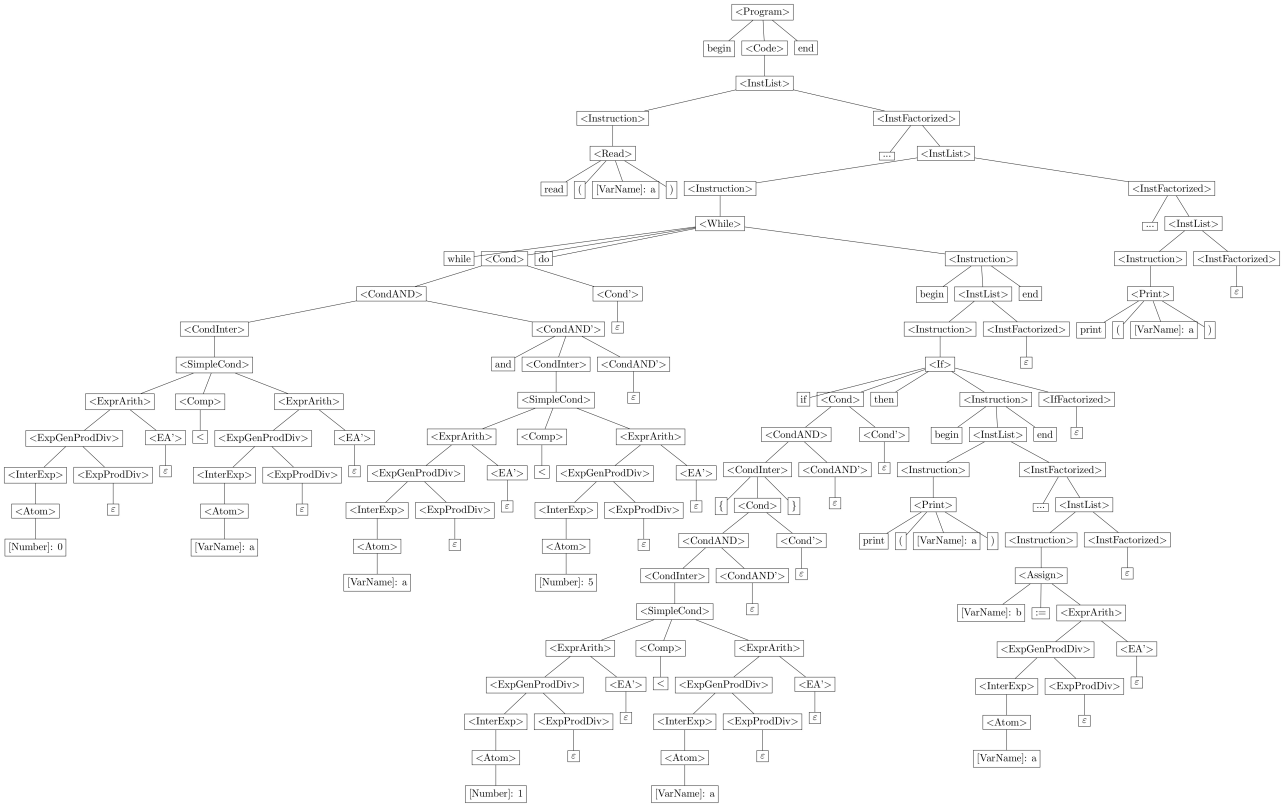

FIGURE 2.5: condAnd.pmp parse tree