M-IRIFS

INFO-F403 : Introduction to language theory and compiling

Project Part2

Pierre Léchaudé

Chahine Mabrouk Bouzouita

Gilles Geeraerts

Mrudula Balachander

Mathieu Sassolas

24/11/2024

Table des matières

1 Part 1 : Lexer

1.1 LexicalAnalyzer

1.1.1 Regular expressions

— VARNAME = [a-z][a-zA-Z0-9]*

For a token to be recognized as a VARNAME, it has to begin with a lowercase letter (*[a-z]*). Next, it may have as much that we want of alphanumeric caracters (*[a-zA-Z0-9]**).

— NUMBER = [1-9][0-9]*|0

For a token to be recognized as a NUMBER, it can not begin with a 0 (*[1-9]*) but can be followed by any numbers between 0 and 9 (*[0-9]**). Or it can also be just the number 0 (*|0*) followed by nothing.

— PROGNAME = [A-Z][a-zA-Z]*[_][a-zA-Z]*

For a token to be recognized as a PROGNAME, it has to begin with an uppercase letter (*[A-Z]*). Next, it can be any letter (lower or uppercase) (*[a-zA-Z]**) as much time as we want, but there as to be an underscore character somewhere in the PROGNAME (*[_]*).

— WHITESPACE = $[\backslash t \backslash r \backslash n]+$

We've had a `WHITESPACE` macro that recognized when there is a return of line (\n), a tabulation (\t), or a carriage return (\r). This macro will be very helpful when analyzing an input file. The + sign means that there can be one or more of the recognized tokens (\t, \r, \n).

1.1.2 States

We begin in the initial state (<YYINITAIL>). In this state, we check every token to see if it matches a particular regular expression (cf 1.1.1). If we found a dollar sign ($) or a double exclamation point (!!), we are in a comment zone where we don't need to take the following tokens into account (until the comment section is closed). Because there are two types of comment (long and short), we chose to create two new comment states :

— <LONG_COMMENT>

  When we are in this state, we check every token following the "!!". If it's another "!!" we go back to initial sates because the comment section closed. For every other token in between, we simply ignore them and stay in the long comment state.

— <SHORT_COMMENT>

  When we are in this state. We switch back to initial state if we encounter a newline (\n). As long as we stay on the same line, we simply ignore all the characters encountered and stay in the short comment states.

1.2 Main File

The `main` function processes an input file, scans it using a lexical analyzer, and performs the following steps :

1. Reads the input file specified in the command-line arguments.
2. Initializes a lexical analyzer to identify and print each symbol in the file.
3. Builds a symbol table, mapping variable names to the lines where they are first encountered.
4. Prints the completed symbol table at the end of the analysis.
5. Handles errors if the file is missing or cannot be opened.

If the input file is not found or invalid, the program outputs an error message and exits.

1.3 Makefile

The purpose of the Makefile is to automate the process of compiling, testing, and cleaning up files in our project. By using a Makefile, we can run all necessary tasks with simple commands, reducing the potential for errors and improving productivity.

The Makefile in our project defines several targets, each responsible for a specific task :

— `all` : This is the default target. It compiles all the necessary Java files, generates the lexical analyzer from the JFlex specification, and creates a JAR file that can be used to execute the program. This is the most common target and allows us to build the entire project with a single command.

— `compile` : This target ensures that all Java source files are compiled. It is a dependency for other targets, such as running tests or generating the JAR file.

— `tests` : This target runs the project on a set of input test files. It automates the process of executing the program on each test file located in the `test/test1/` directory. For each test file, the program is executed and the output is displayed, making it easy to validate the behavior of the lexical analyzer.

— `run` : This target allows the program to be executed from the generated JAR file with a specific input file (by default, `Euclid.gls`). It simplifies running the program manually by using the JAR file, which bundles all the compiled classes into a single archive.

— `clean` : This target removes all intermediate files generated during the compilation process, including `.class` files and the generated lexical analyzer Java file. This is useful for maintaining a clean working directory.

— `javadoc` : this target generates the java documentation for the project.

— `javadoclean` : this target removes all the java documentation files created with the *javadoc* target.

Overall, the Makefile streamlines the development workflow by automating the steps of compilation, testing, and execution. With simple commands like `make all`, `make tests`, `make clean`, `make javadoc` and `make javadoclean` we can efficiently manage our project's lifecycle without needing to manually run individual commands.

1.4 Tests

The `tests` target in the Makefile automates running the program on several test files from the `test/test1/` directory. It executes the lexical analyzer for each test file, displaying the recognized tokens and their corresponding lexical units, the program reports unrecognized symbols while continuing to process valid tokens.

`test_comments.gls` test the short and long comment and address the missing closing comments.

`test_whitespaces.gls` confirms that the analyzer handles whitespaces without affecting token recognition.

In `test_illegal_symbol.gls`, the program highlights some issues with the code. Words are not always recognized as complete units, which can lead to unexpected results.

```
IllegalSymbol ->
Unrecognized symbol: I
token: llegalSymbol     lexical unit: VARNAME
expected -> Unrecognized symbol: IllegalSymbol

5oph1e ->
token: 5                lexical unit: NUMBER
token: oph1e            lexical unit: VARNAME
expected -> Unrecognized symbol: 5oph1e

00 ->
token: 0                lexical unit: NUMBER
token: 0                lexical unit: NUMBER
expected -> Unrecognized symbol: 00
```

2 Part 2 : Parser

2.1 Grammar transformation

2.1.1 Unproductive or unreachable variables

After We applied the corresponding algorithm, we found out that the expression <CALL> was reachable but unproductive because it leads to nowhere. We thus removed it from the grammar.

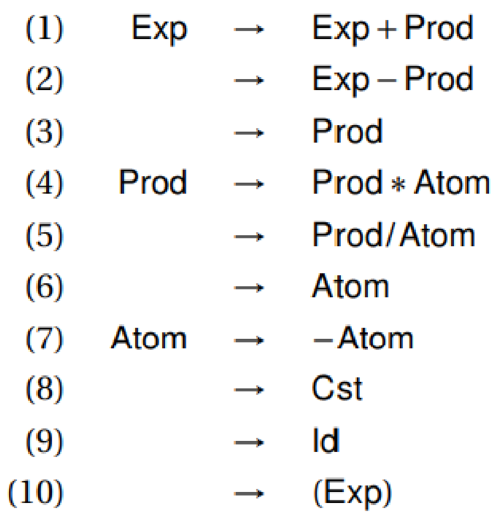
2.1.2 Ambiguity and priority

In order to implement the order priority required (FIG2.1), we needed to adjust the grammar such that it respects the priority rules.

Operators	**Associativity**
− (unary)	right
⋆, /	left
+, − (binary)	left
==, <=, <	left
−>	right

FIGURE 2.1: Priority table

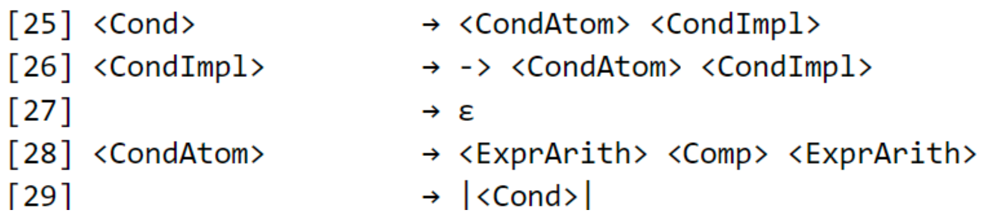
As seen in the lecture notes (CF section 4.4.4), we applied the following modifications (see fig 2.2). the idea is the same as in the exercise sessions, I.e by forcing the grammar the create additions of product and products of atoms. Here we can observe that this part of the grammar is already in LL(1) (left-recursion and Factorisation).



(1)	Exp	→	Exp + Prod
(2)		→	Exp − Prod
(3)		→	Prod
(4)	Prod	→	Prod ∗ Atom
(5)		→	Prod / Atom
(6)		→	Atom
(7)	Atom	→	−Atom
(8)		→	Cst
(9)		→	Id
(10)		→	(Exp)

FIGURE 2.2: first modifications

We must also implement the right priority on the "->" symbol. To do that, we made several modifications to the <COND> expression (see on fig2.3). By adding the <CONDIMPL> expression, we are able to implement the right associativity to the "->" operator.

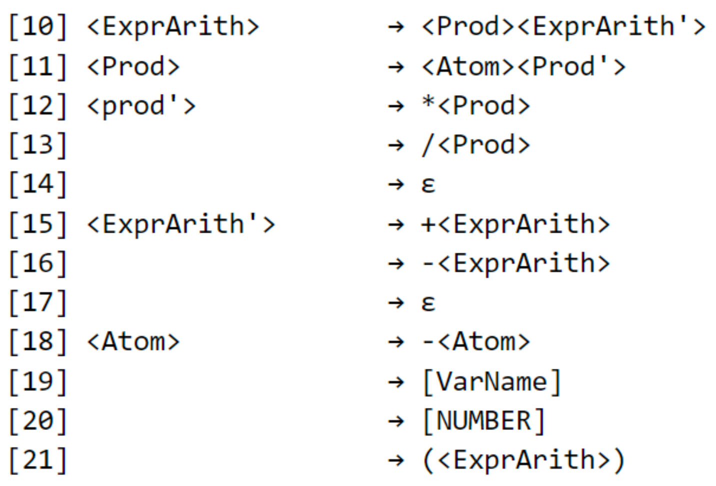


[25]	<Cond>	→ <CondAtom> <CondImpl>
[26]	<CondImpl>	→ -> <CondAtom> <CondImpl>
[27]		→ ε
[28]	<CondAtom>	→ <ExprArith> <Comp> <ExprArith>
[29]		→ |<Cond>|

FIGURE 2.3: condition modification

2.1.3 Left-recursion and factorisation

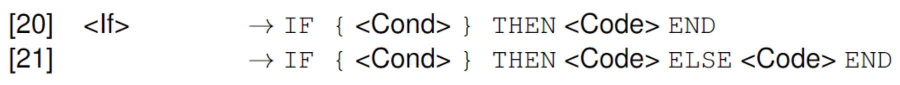
In order to make our grammar LL(1), we first removed the left-recursion in our grammar by using the corresponding algorithm (cf lecture notes section 4.4.4). We obtain this result (fig 2.4) :



```
[10] <ExprArith>        → <Prod><ExprArith'>
[11] <Prod>             → <Atom><Prod'>
[12] <prod'>            → *<Prod>
[13]                    → /<Prod>
[14]                    → ε
[15] <ExprArith'>       → +<ExprArith>
[16]                    → -<ExprArith>
[17]                    → ε
[18] <Atom>             → -<Atom>
[19]                    → [VarName]
[20]                    → [NUMBER]
[21]                    → (<ExprArith>)
```

FIGURE 2.4:

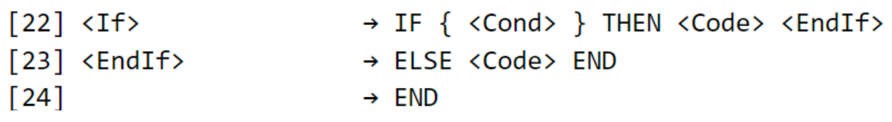
We then focussed on the IF expressions where a factoring problem occurs. In fact, here the grammar contains at least two rules with the same left-hand side, and a common prefix in the right-hand side (see fig 2.5) :



```
[20]  <If>        → IF { <Cond> } THEN <Code> END
[21]              → IF { <Cond> } THEN <Code> ELSE <Code> END
```

FIGURE 2.5: Initial IF expression

As it has been done in the erxrecise sessions (ex 4.5), we made the following modifications (see fig 2.6) :

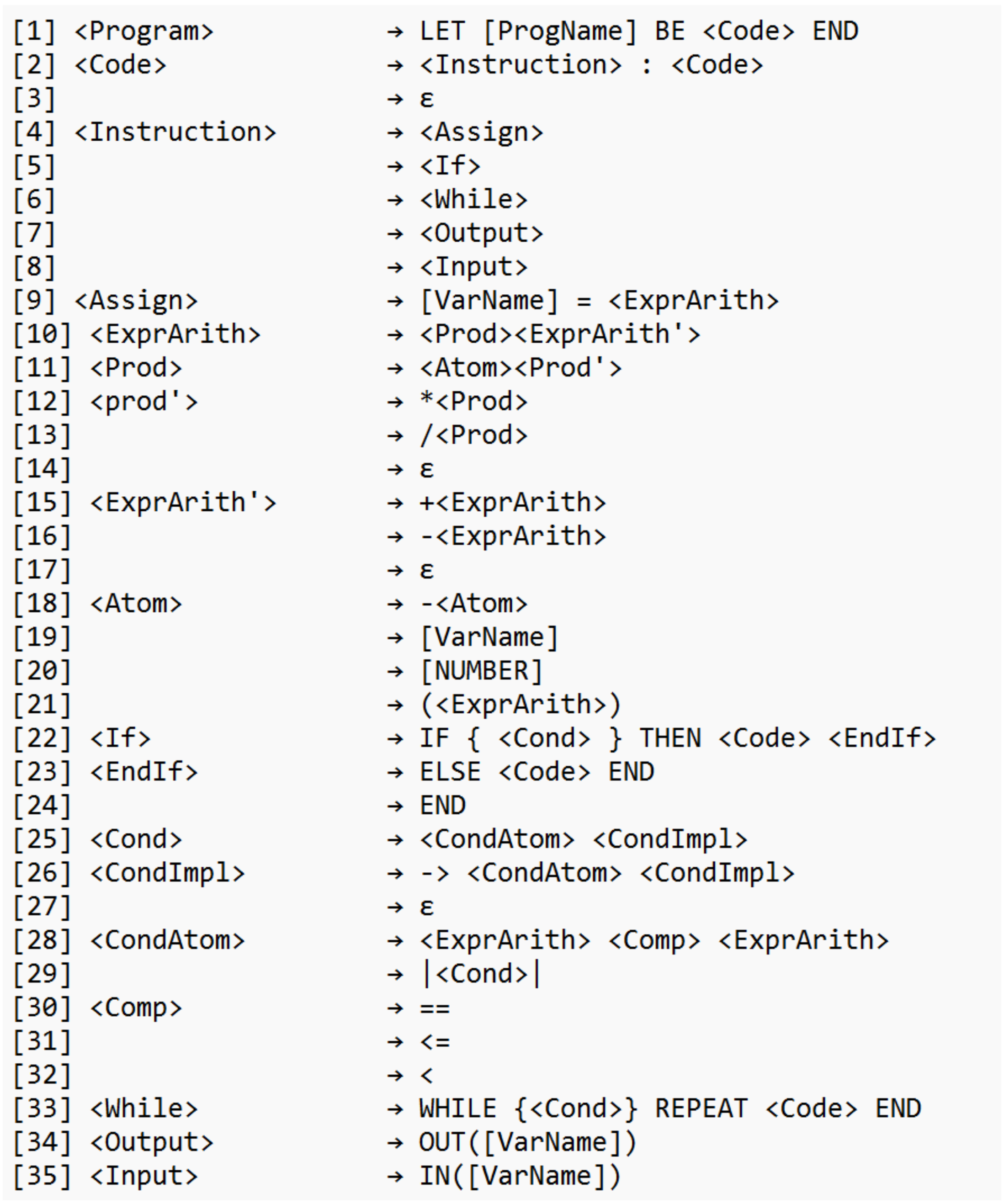


```
[22] <If>          → IF { <Cond> } THEN <Code> <EndIf>
[23] <EndIf>       → ELSE <Code> END
[24]               → END
```

FIGURE 2.6: Final IF expression

2.1.4 Final grammar

By summing up all the modifications mentioned before, we obtain this grammar :

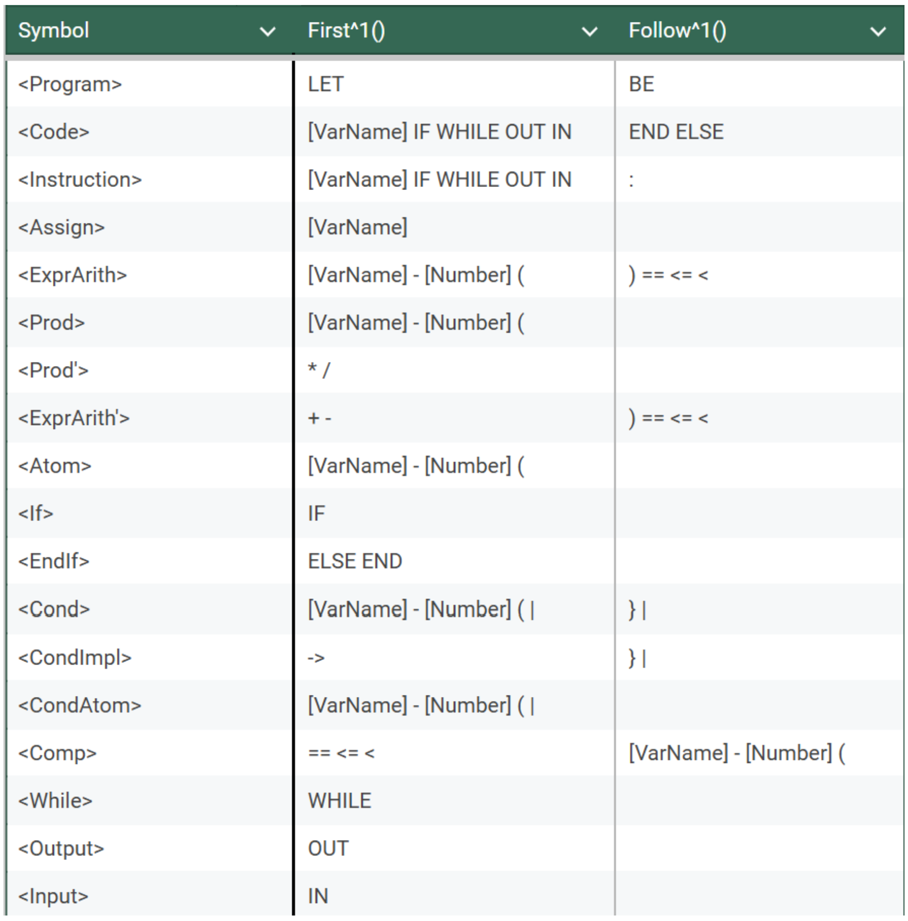


```
[1] <Program>           → LET [ProgName] BE <Code> END
[2] <Code>              → <Instruction> : <Code>
[3]                     → ε
[4] <Instruction>       → <Assign>
[5]                     → <If>
[6]                     → <While>
[7]                     → <Output>
[8]                     → <Input>
[9] <Assign>            → [VarName] = <ExprArith>
[10] <ExprArith>        → <Prod><ExprArith'>
[11] <Prod>             → <Atom><Prod'>
[12] <prod'>            → *<Prod>
[13]                    → /<Prod>
[14]                    → ε
[15] <ExprArith'>       → +<ExprArith>
[16]                    → -<ExprArith>
[17]                    → ε
[18] <Atom>             → -<Atom>
[19]                    → [VarName]
[20]                    → [NUMBER]
[21]                    → (<ExprArith>)
[22] <If>               → IF { <Cond> } THEN <Code> <EndIf>
[23] <EndIf>            → ELSE <Code> END
[24]                    → END
[25] <Cond>             → <CondAtom> <CondImpl>
[26] <CondImpl>         → -> <CondAtom> <CondImpl>
[27]                    → ε
[28] <CondAtom>         → <ExprArith> <Comp> <ExprArith>
[29]                    → |<Cond>|
[30] <Comp>             → ==
[31]                    → <=
[32]                    → <
[33] <While>            → WHILE {<Cond>} REPEAT <Code> END
[34] <Output>           → OUT([VarName])
[35] <Input>            → IN([VarName])
```

FIGURE 2.7: Final LL(1) GILLES grammar

2.2 Action table

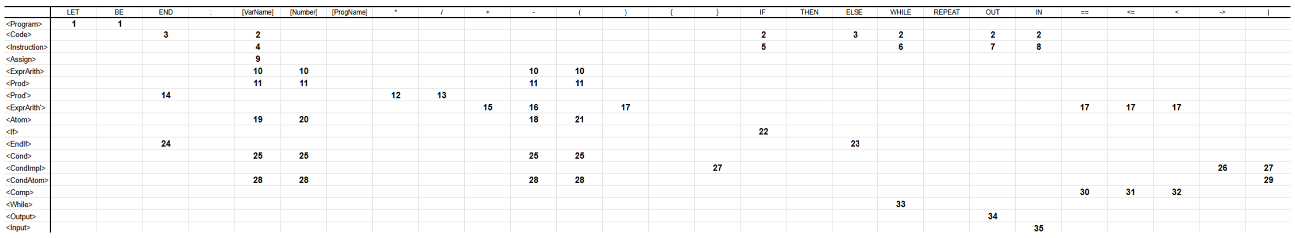
2.2.1 First and follow



Symbol	First^1()	Follow^1()
<Program>	LET	BE
<Code>	[VarName] IF WHILE OUT IN	END ELSE
<Instruction>	[VarName] IF WHILE OUT IN	:
<Assign>	[VarName]	
<ExprArith>	[VarName] - [Number] (	) == <= <
<Prod>	[VarName] - [Number] (	
<Prod'>	* /	
<ExprArith'>	+ -	) == <= <
<Atom>	[VarName] - [Number] (	
<If>	IF	
<EndIf>	ELSE END	
<Cond>	[VarName] - [Number] (|	} |
<CondImpl>	->	} |
<CondAtom>	[VarName] - [Number] (|	
<Comp>	== <= <	[VarName] - [Number] (
<While>	WHILE	
<Output>	OUT	
<Input>	IN	

FIGURE 2.8: First and follow set of the grammar variables

2.2.2 Action table



	LET	BE	END	:	[VarName]	[Number]	[ProgName]	*	/	+	-	(	)	{	}	IF	THEN	ELSE	WHILE	REPEAT	OUT	IN	==	<=	<	->	|
<Program>	1	1																									
<Code>			3		2											2		3	2		2	2					
<Instruction>					4											5			6		7	8					
<Assign>					9																						
<ExprArith>					10	10					10	10															
<Prod>					11	11					11	11															
<Prod'>			14					12	13																		
<ExprArith'>										15	16		17										17	17	17		
<Atom>					19	20					18	21															
<If>																22											
<EndIf>			24															23									
<Cond>					25	25					25	25															
<CondImpl>															27											26	27
<CondAtom>					28	28					28	28															29
<Comp>																							30	31	32		
<While>																			33								
<Output>																					34						
<Input>																						35					

FIGURE 2.9: Action table

2.3 Recursive-Descent Parser

2.3.1 Overview of the Parser Structure

The parser processes input tokens recursively based on grammar rules. Each method corresponds to a specific grammar rule.

— **Program Rule :** Parses the entire program structure.

— **Code Rule :** Parses sequences of instructions or handles epsilon.

— **Instruction Rule :** Parses individual instructions like `IF`, `WHILE`, or assignments.

— **Utility Methods :** `match()` ensures tokens conform to expected grammar, while `nextToken()` retrieves the next input token.

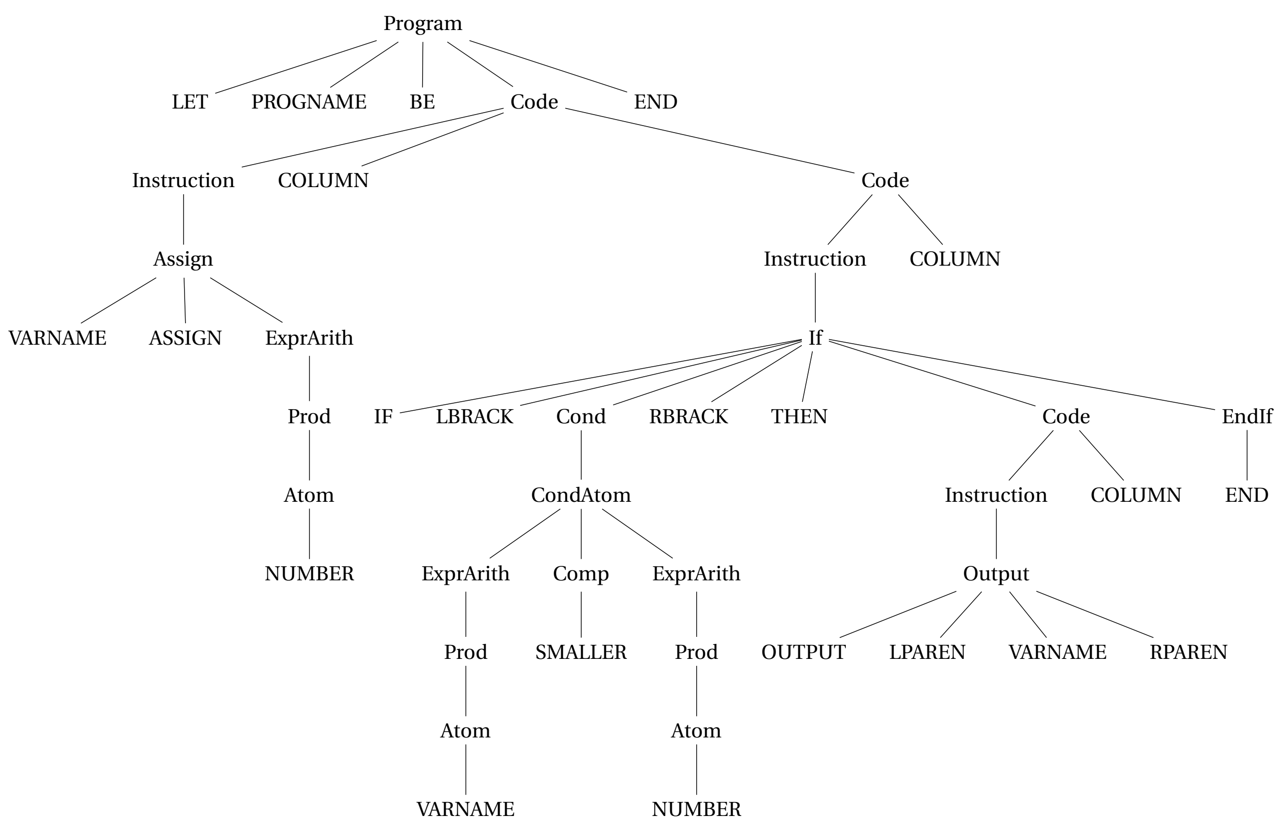
2.3.2 Recursive Parsing Workflow

Below is an example of how recursive parsing builds the tree for a simple program :

```
LET Prog_name BE x = 5 : IF {x < 10} THEN OUT(x): END: END
```

Parse Tree Example

The corresponding parse tree would look like this (Epsilon nodes are not explicit here for the visual comprehension of the example) :



2.3.3 Key Recursive Calls

— `Program()` calls `Code()`.

— `Code()` calls `Instruction()` and itself for sequences.

— `Instruction()` dispatches specific methods (e.g., `If()`, `While()`).

— Each method follows the grammar, producing terminal or epsilon nodes.

2.3.4 Tests

To validate the implementation, test files with intentional grammar errors were created. These files ensure the system correctly detects and reports syntax issues. Each file targets a specific error type, such as missing tokens or misplaced operators, and the output is compared to expected error messages to confirm proper error handling.