Compiler Project

2014 – 2015

HAZEM ABUMOSTAFA / HOSSAM KHALIL KHALIL

TABLE OF CONTENTS

PHASE 1: LEXICAL ANLYSIS

Definition:

Given a piece of code, this phase should analyze it and extracts a list of tokens found. This phase relies on a pre-made DFA transition table, which should be well-designed for reliable output.

Design:

THE CLASS:

We made a class and called it `Lexical`. The constructor of the class takes only the path of the DFA table (as plain-text file).

A lexical analyzer requires a transition table to use. We wanted to make that table code-independent, which will make future additions and edits much easier. To solve this problem, we designed our system to deal with a specific-formatted plain-text file containing the transitions.

The system initially takes the path of the table once, and analyze any piece of code afterwards with `analyze` function, using the same constructed table.

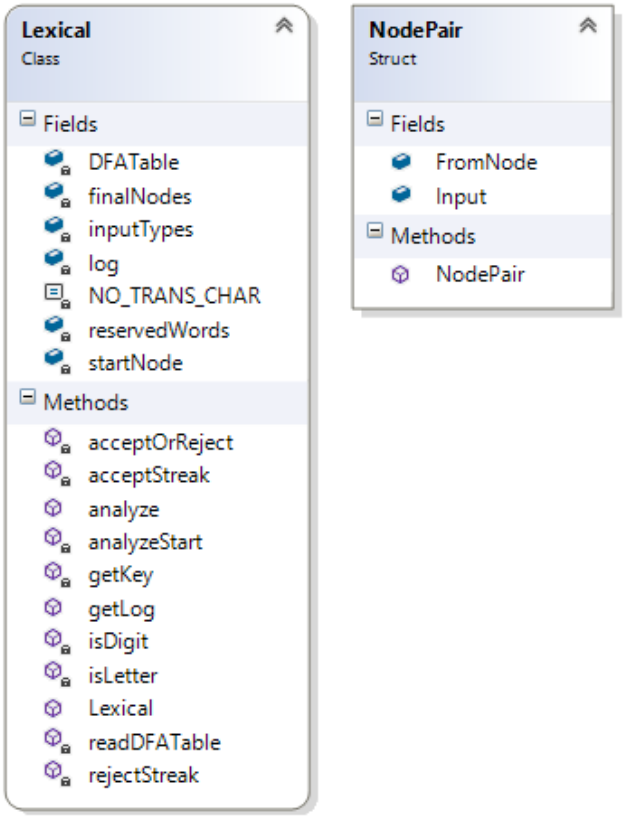
`Lexical` main functions are:

1. `readDFATable`: A private function that reads the content of the plain-text DFA transition table and store it in a private structured data object. This function is automatically called when a new object of `Lexical` is instantiated.

2. `analyze`: A public function that do the work of analyzing the code (converting code into tokens). The code is a `string` object, and is given as the only input parameter. Returns a `List` object of accepted tokens.

3. `getLog`: A public function that returns a summary (log history) of the last analyzed code by `analyze`. This is usually useful for testing the system. The log contains both accepted and rejected tokens along with their lexeme names.

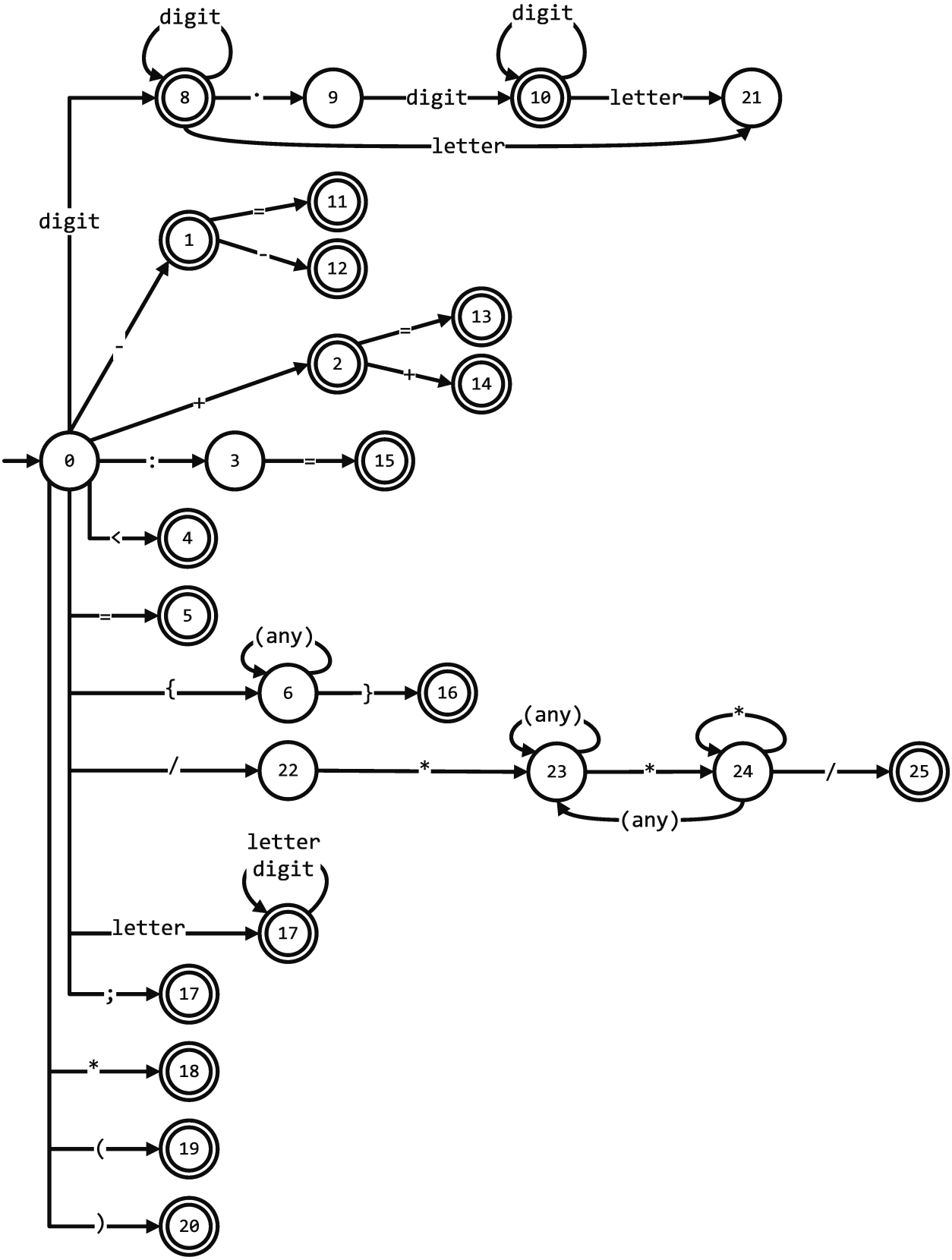
CLASS DIAGRAM:



DFA TABLE FORMAT:

(the start state)	0
(list of final states)	1|Minus Operator - 2|Comment
(list of reserved words)	if then else end write
(separator)	----------
(list of input characters)	letter digit - +
(separator)	----------
(corresponding state for each input, for all states, '-' indicates 'no transition' and coded as a constant)	0 7 8 1 2 1 - - 12 - 2 - - - 5 3 - - - -

DFA DIAGRAM:


digit
digit
8
.
9
digit
10
letter
21
letter
digit
1
=
11
-
12
-
2
=
13
+
14
+
0
:
3
=
15
<
4
=
5
(any)
{
6
}
16
(any)
*
/
22
*
23
*
24
/
25
(any)
letter
digit
letter
17
;
17
*
18
(
19
)
20

Output:

The expected output from this phase is a list of tokens found in the code. The `Lexical` class also offers log history showing the accepted tokens along with any error/unidentified tokens for testing purposes.

The system supports the following tokens:

- ID's
- Numbers
- Fraction numbers
- Minus operator (`-`)
- Plus operator (`+`)
- Less-than comparison (`<`)
- Equality comparison (`=`)
- Assign operator (`:=`)
- Augmented minus operation (`-=`)
- Augmented plus operation (`+=`)
- Decrement operator (`--`)
- Increment operator (`++`)
- C-like comments (`/*comment*/`)
- Curley braces comments (`{comment}`)
- End of statement sign (`;`)
- Multiplication operator (*)
- Left parenthesis
- Right parenthesis

And the following reserved words (as defined by Tiny-C language description):

`If`, `Then`, `Else`, `End`, `Write`, `Read`, `Repeat`, `Until`.

PHASE 2: SYNTAX AND SYMANTIC ANALYSIS

Definition:

Given a list of tokens, this phase should check the grammar of the input tokens and provide the user with a syntax tree.

After getting the Syntax tree it'll be passed on to validate the legality of the code (making sure the operations have the correct data types).

Design:

THE CLASS:

The class is `Syntax`. The constructor of the class takes only the list of accepted tokens.

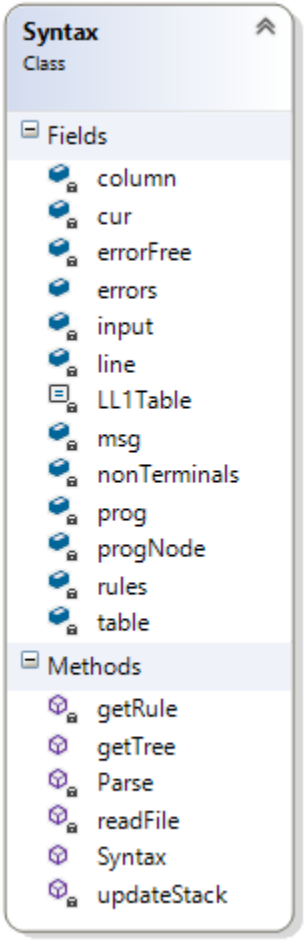
We chose to go with "LL1" approach in the syntax phase, and we stored the table in a text file to make it independent from the code.

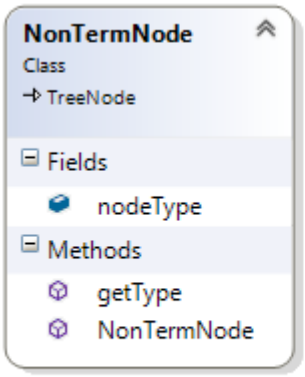
Like `Lexical`, `Syntax` also initially takes the path of the table once, then the system uses it to verify the stream of tokens and check that the code follow the language grammar.

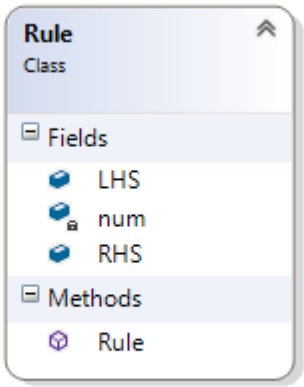
`Syntax` main functions are:

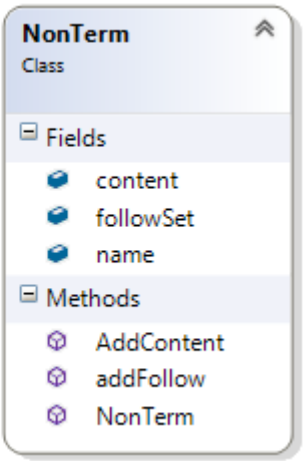
1. `readFile`: A private function that reads the content of the LL1 text file and stores the rules in a rule `SortedList`, stores the legal transitions in another `SortedList` with the non-terminal as the key and also stores the follow set so if there is an error it can handle it.
2. `Parse`: A private function in which the list of tokens in parsed by LL1 method using data loaded from `readFile`, using private helper functions as `getRule` which looks for the rule which matches the terminal in input, `updateStack` which updates the LL1 stack by adding new non-terminals when needed.

CLASS DIAGRAM:









LL1 TABLE:

T\NT	;	if	Then	end	else	repeat	until	id	:=	Read	write	<	>	=	+	-	*	/	(	)	number	$
Prg		1				1		1		1	1											
StSq’	2			3	3		3															3
St		4				5		6		7	8											
IfSt		9																				
If’				10	11																	
Repeat St						12																
Assign st								13														
Read st										14												
Write St											15											
Exp								16											16		16	
E’	18		18	18	18		18					17	17	17						18		18
Cop												19	20	21								
Smp exp								22											22		22	
Smp exp’	24		24	24	24		24					24	24	24	23	23				24		24
add Op															25	26						
Term								27											27		27	
Term’	29		29	29	29		29					29	29	29	29	29	28	28				29
mul Op																	30	31				
Factor								34											32		33	

Output:

The expected output from Syntax phase is a parse tree along with any error/unidentified to tell the user where to modify.

The expected output from Semantic phase is an annotated tree showing the user the evaluation of the code.

RESPOSIBILITY MATRIX:

Lexical Analysis	Hazem Hamdy AbuMostafa
Syntax Analysis	Hossam Khalil Khalil
Semantic Analysis	Hazem Hamdy AbuMostafa