CS39003 Compilers Laboratory
Autumn 2025
Assignment 3
Date of posting: 25-Aug-2025

SLR(1) Parsing

In this assignment, we continue with the same language of string operations as in Assignment 2 (on LL(1) parsing). Let us only change the concatenation operator to + instead of dot. (This is needed because we use dot in LR(0) items as the progress marker. Incidentally, Perl uses **.** as the string-concatenation operator, whereas Java and JavaScript use + for this purpose. For us, the difference is inconsequential.) The end-of-input symbol continues to be the new-line character **\n**. In order to avoid writing big non-terminal names, we adopt their first letters as their new names. That is, LINE, EXPR, ARG, REST, BASE, and XPNT will now be called L, E, A, R, B, and X. With these modifications, the grammar is restated below. The terminal symbols (tokens) are in red.

(0)	L′	⟶	L	FOLLOW(L′) = { \n }
(1)	L	⟶	**ID = E**	FOLLOW(L) = { \n }
(2)	E	⟶	A R	FOLLOW(E) = {) , \n }
(3)	R	⟶	ε	FOLLOW(R) = {) , \n }
(4)	R	⟶	+ E	
(5)	A	⟶	B X	FOLLOW(A) = { + ,) , \n }
(6)	X	⟶	ε	FOLLOW(X) = { + ,) , \n }
(7)	X	⟶	^ NUM	
(8)	B	⟶	STR	FOLLOW(B) = { ^ , + ,) , \n }
(9)	B	⟶	$ID	
(10)	B	⟶	(E)	

The LR(0) automaton for this grammar is given on the next page. SLR(1) parsing uses the FOLLOW function values of the non-terminals (these are listed above alongside the productions). The SLR(1) parsing table derived from the LR(0) automaton and from these FOLLOW values is given on the next-to-next page.

In this assignment, you are required to implement the **iterative SLR(1) parsing** algorithm using a stack. Proceed as instructed below. Maintain a symbol table as in all of the earlier assignments.

Tokens

Write a lex file to identify the tokens from the input. Indeed, all the tokens used in Assignment 2 are precisely the tokens in the current assignment too. Only change the concatenation operator from dot to plus. With only this one modification, your lex file for Assignment 2 should work. Continue to use the state RVALUE of lex (to disambiguate valid lines like `abc = abc`).

Recall that a token is a (type, value) pair. The types of the tokens are useful for parsing decisions. The values are used for performing the string operations. For this assignment, the values are needed only for the tokens ID, STR, $ID, and NUM. Assume that the values of ID, STR, and $ID are strings, whereas the values of NUM are integers.

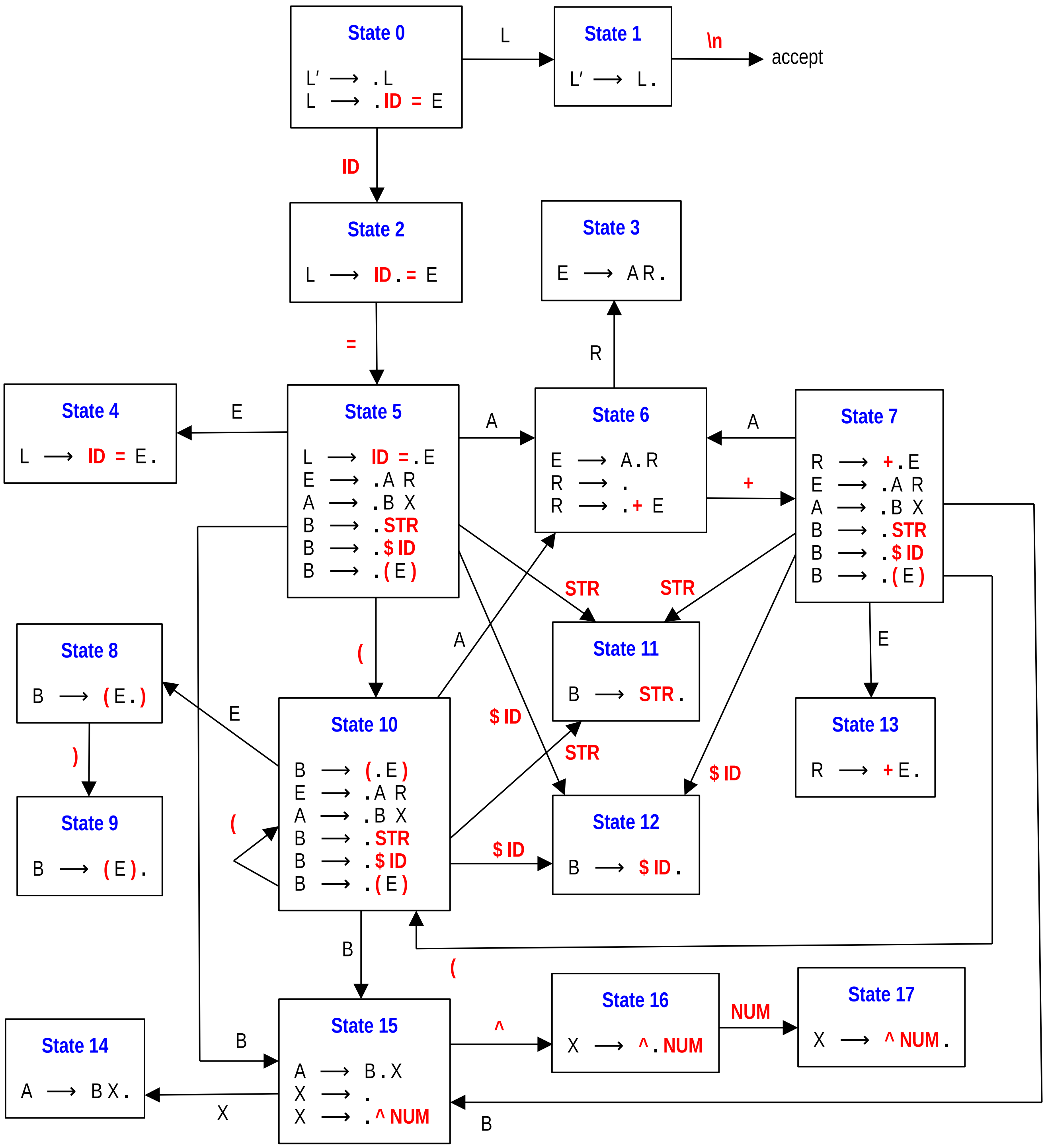
Stack elements

For making the parsing decisions, the parser uses the token types. However, these types need not be stored in the stack. Any LR parser maintains a stack of states. Every state (except the initial state) is associated with a unique grammar symbol (either terminal or non-terminal). It is not necessary to store these symbols in the stack. But for carrying out the operations on strings, some additional information (values) must be stored alongside the state numbers. For the four terminal symbols ID, STR, $ID, and NUM, this is the lexical value of the token as mentioned above. We need to store a string against each of E, A, R, and B, whereas we need to store an integer against the non-terminal X. To sum up, each element in the parse stack should be a structure consisting of the following two fields.

- The number of the current state (an **integer**, 0 – 17 in our example).
- A **union** of int and string data types (this is because the additional information stored for grammar symbols are not of homogeneous data types). For the operators = , + , and ^ , the punctuation symbols (,) , and \n , and for L (the start non-terminal), this union is not used, and can be left uninitialized.

Strictly follow the above organization of the stack elements. Understanding this will be useful in future assignments. You may implement the stack functions yourself or use standard library calls.

LR(0) Automaton for our Grammar



SLR(1) Parsing Table for our Grammar

STATE	ACTION										GOTO					
	ID	STR	$ ID	NUM	=	+	^	(	)	\n	L	E	A	R	B	X
0	s2										1					
1										ACC						
2					s5											
3									r2	r2						
4										r1						
5		s11	s12					s10				4	6		15	
6						s7			r3	r3				3		
7		s11	s12					s10				13	6		15	
8									s9							
9						r10	r10		r10	r10						
10		s11	s12					s10				8	6		15	
11						r8	r8		r8	r8						
12						r9	r9		r9	r9						
13									r4	r4						
14						r5			r5	r5						
15						r6	s16		r6	r6						14
16				s17												
17						r7			r7	r7						

Implementation of the SLR(1) parser

Parse each line of the input file one by one. Always keep the next input token available in a global variable. Parsing actions are based on the current state (stored at the top of the stack) and the next input token.

Start by pushing the start state 0 to the top of an empty parse stack. The start state is not associated with any grammar symbol, so the union in that element can be left uninitialized. Then continue the parsing loop until either the input is accepted, or a failure occurs. You may exit from the program in case of a parsing failure (there is no need to recover from failures). Each iteration of the parsing loop works as follows. Let *s* be the state at the top of the stack, and *a* the next input symbol.

- If the (*s*, *a*)-th entry of the above table is empty, report failure, and exit.
- If *s* is 1, and *a* is the new-line symbol, accept.
- If the table entry is a shift directive, consume the input symbol, and push the new state to the top of the stack. Depending on the type of the token just consumed from the input, the union in the stack top is appropriately populated or left uninitialized.
- If the table entry is a reduction directive, look at the production number in the directive (these numbers are as in the grammar stated near the beginning of this write-up). If the body of the production contains *k* grammar symbols, make *k* pops from the stack, and look at the state at the new top of the stack. Depending on this state and the head of the production used in the reduction step, find the new state from the GOTO section of the above table. Push that state to the top of the stack. Note however that this operation may be associated with some string computations. For example, if the reduction by E → A R is used in State 3, you retrieve the strings stored in the unions for A and R, concatenate them, and store the concatenated string in the union of E in the new top of the stack.

Error Handling

As in the last assignment, detect three types of errors (and nothing else): (i) invalid character by lex (ignore), (ii) undefined variable (use the empty string), and (iii) parse error (exit the program).

What to submit

Write a lex file to identify the tokens from the input (write only `yywrap()` in the user-code section of the lex file). The parser code should be written in a separate C/C++ file (like `SLRparser.c`). Also write a `makefile` with `compile`, `run`, and `clean` targets. Submit only the above three files in a single zip/tar/tgz archive.

Sample Output

Show every state transition, every shift and every reduction step, and the final assignment. In the sample output below, → stands for a push operation in the stack, and ← denotes a pop operation in the stack. The shift actions are shown as [s…], and the reduction actions as [r…]. In order to avoid very wide lines in the output, a new-line is printed after the completion of every reduction step.

```
$ cat input.txt
S1 = a

S2 = b^2 + c

S3 = ( $S1 + (b + c^4) ^2 ) ^ 3

S4 = (($S1 + d) ^ 3 + $S2 + d^2)^4 + (EF^2 + GH) ^ 5
$ make run
lex tokens.l
gcc SLRparser.c
./a.out input.txt
+++ Going to parse next statement
    0 [s2] -> 2 [s5] -> 5 [s11] -> 11 [r8] <- 5 -> 15
      [r6] -> 14
      [r5] <- 15 <- 5 -> 6
      [r3] -> 3
      [r2] <- 6 <- 5 -> 4
      [r1] <- 5 <- 2 <- 0 -> 1
      -> ACCEPT
+++ Stored S1 = a

+++ Going to parse next statement
    0 [s2] -> 2 [s5] -> 5 [s11] -> 11 [r8] <- 5 -> 15
      [s16] -> 16 [s17] -> 17 [r7] <- 16 <- 15 -> 14
      [r5] <- 15 <- 5 -> 6
      [s7] -> 7 [s11] -> 11 [r8] <- 7 -> 15
      [r6] -> 14
      [r5] <- 15 <- 7 -> 6
      [r3] -> 3
      [r2] <- 6 <- 7 -> 13
      [r4] <- 7 <- 6 -> 3
      [r2] <- 6 <- 5 -> 4
      [r1] <- 5 <- 2 <- 0 -> 1
      -> ACCEPT
+++ Stored S2 = bbc

+++ Going to parse next statement
    0 [s2] -> 2 [s5] -> 5 [s10] -> 10 [s12] -> 12 [r9] <- 10 -> 15
      [r6] -> 14
      [r5] <- 15 <- 10 -> 6
      [s7] -> 7 [s10] -> 10 [s11] -> 11 [r8] <- 10 -> 15
      [r6] -> 14
      [r5] <- 15 <- 10 -> 6
      [s7] -> 7 [s11] -> 11 [r8] <- 7 -> 15
      [s16] -> 16 [s17] -> 17 [r7] <- 16 <- 15 -> 14
      [r5] <- 15 <- 7 -> 6
      [r3] -> 3
      [r2] <- 6 <- 7 -> 13
      [r4] <- 7 <- 6 -> 3
      [r2] <- 6 <- 10 -> 8
      [s9] -> 9 [r10] <- 8 <- 10 <- 7 -> 15
      [s16] -> 16 [s17] -> 17 [r7] <- 16 <- 15 -> 14
      [r5] <- 15 <- 7 -> 6
      [r3] -> 3
      [r2] <- 6 <- 7 -> 13
      [r4] <- 7 <- 6 -> 3
      [r2] <- 6 <- 10 -> 8
      [s9] -> 9 [r10] <- 8 <- 10 <- 5 -> 15
      [s16] -> 16 [s17] -> 17 [r7] <- 16 <- 15 -> 14
      [r5] <- 15 <- 5 -> 6
      [r3] -> 3
      [r2] <- 6 <- 5 -> 4
      [r1] <- 5 <- 2 <- 0 -> 1
      -> ACCEPT
+++ Stored S3 = abccccbccccabccccbccccabccccbcccc

+++ Going to parse next statement
    0 [s2] -> 2 [s5] -> 5 [s10] -> 10 [s10] -> 10 [s12] -> 12 [r9] <- 10 -> 15
      [r6] -> 14
      [r5] <- 15 <- 10 -> 6
      [s7] -> 7 [s11] -> 11 [r8] <- 7 -> 15
      [r6] -> 14
      [r5] <- 15 <- 7 -> 6
      [r3] -> 3
      [r2] <- 6 <- 7 -> 13
      [r4] <- 7 <- 6 -> 3
      [r2] <- 6 <- 10 -> 8
      [s9] -> 9 [r10] <- 8 <- 10 <- 10 -> 15
      [s16] -> 16 [s17] -> 17 [r7] <- 16 <- 15 -> 14
      [r5] <- 15 <- 10 -> 6
      [s7] -> 7 [s12] -> 12 [r9] <- 7 -> 15
      [r6] -> 14
      [r5] <- 15 <- 7 -> 6
      [s7] -> 7 [s11] -> 11 [r8] <- 7 -> 15
      [s16] -> 16 [s17] -> 17 [r7] <- 16 <- 15 -> 14
      [r5] <- 15 <- 7 -> 6
      [r3] -> 3
      [r2] <- 6 <- 7 -> 13
      [r4] <- 7 <- 6 -> 3
      [r2] <- 6 <- 7 -> 13
      [r4] <- 7 <- 6 -> 3
      [r2] <- 6 <- 10 -> 8
      [s9] -> 9 [r10] <- 8 <- 10 <- 5 -> 15
      [s16] -> 16 [s17] -> 17 [r7] <- 16 <- 15 -> 14
      [r5] <- 15 <- 5 -> 6
      [s7] -> 7 [s10] -> 10 [s11] -> 11 [r8] <- 10 -> 15
      [s16] -> 16 [s17] -> 17 [r7] <- 16 <- 15 -> 14
      [r5] <- 15 <- 10 -> 6
      [s7] -> 7 [s11] -> 11 [r8] <- 7 -> 15
      [r6] -> 14
      [r5] <- 15 <- 7 -> 6
      [r3] -> 3
      [r2] <- 6 <- 7 -> 13
      [r4] <- 7 <- 6 -> 3
      [r2] <- 6 <- 10 -> 8
      [s9] -> 9 [r10] <- 8 <- 10 <- 7 -> 15
      [s16] -> 16 [s17] -> 17 [r7] <- 16 <- 15 -> 14
      [r5] <- 15 <- 7 -> 6
      [r3] -> 3
      [r2] <- 6 <- 7 -> 13
      [r4] <- 7 <- 6 -> 3
      [r2] <- 6 <- 5 -> 4
      [r1] <- 5 <- 2 <- 0 -> 1
      -> ACCEPT
+++ Stored S4 = adadadbbcddadadadbbcddadadadbbcddadadadbbcddEFEFGHEFEFGHEFEFGHEFEFGHEFEFGH

$
```