Problem 1

Contributor: Shivanshi Nagar Degree: 3

(i)

C_constant: Language of all integer and floating point constants.

int_const: Integer constants can be Octal, decimal, or hexadecimal and they should have a suffix.

oct_int: Octal integers start with 0 and can consist of [0-7] zero or more times.

dec_int: Decimal integers start with a non zero digit [1-9] followed by [0-9] zero or more times.

hex_int: Hexadecimal integers start with ‘0x’ or ‘0X’ followed by one hexadecimal digit (decimal digits or [a-f] or [A-F]) followed by hexadecimal digits zero or more times.

oct_digit: Any digit 0, 1, 2, 3, 4, 5, 6, or 7 represents an octal (base 8) digit.

nonzero_digit: Any digit 1, 2, 3, 4, 5, 6, 7, 8, 9 is a nonzero digit.

dec_digit: A decimal digit may be a nonzero digit or 0 (base 10).

hex_digit: A hex digit may be a decimal digit or a, b, c, d, e, or f (or any of those letters, capitalized, i.e. A, B, C, D, E, F) (base 16).

int_suffix: Integer suffix can be one of the following: epsilon (nothing), u or U (unsigned), l or L (long), ll or LL (long long), any combination of long and unsigned, any combination of long long and unsigned. Represents language of integer types.

u_suffix: either u or U; represents unsigned type.

l_suffix: either l or L; represents a long type.

ll_suffix: either ll or LL; represents long long type.

fp_const: Floating point constants can be decimal or hexadecimal, both consisting of their respective digits, with the addition of exponents. Exponents start with either +, - or epsilon (nothing), and have at least one decimal digit. May (optionally) be followed by f or F (to denote floating point number) or l or L (to denote long number).

dec_fp: Floating point decimal numbers may be one or more decimal digits followed by an exponent expression (E or e then an exponent; exponential notation). They may also be represented as one of more decimal digits with a floating point dot specified somewhere within the digits; an exponent expression at the end is optional.

hex_fp: Floating point hex numbers may start with 0x or 0X and have any number of hex digits, potentially including a floating point dot specified somewhere within the digits, followed by p or P and then an exponent term.

exponent: Exponents may optionally have a sign in front (+ or -), followed by 1 or more decimal digits. These represent exponents placed at the end of exponential expressions.

(ii) The automata depicts the grammar of all possible constants, integers and floating points described in part (i). Every number that is a correct representation integer or floating points constants will be accepted by this automata. The automata is equivalent to the grammar of part (i).

Example

0x4fd36671 green highlight:

C_constant -> int_const
Int_const -> hex_int int_suffix
Hex_int -> 0x hex_digit hex_digit* -> 0x hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit
Hex_digit -> dec_digit | a | b| c | d | e | f | A | B | C | D | E | F -> dec_digit -> nonzero_digit -> 4
Hex_digit -> dec_digit | a | b| c | d | e | f | A | B | C | D | E | F -> f
Hex_digit -> dec_digit | a | b| c | d | e | f | A | B | C | D | E | F -> d
Hex_digit -> dec_digit | a | b| c | d | e | f | A | B | C | D | E | F -> dec_digit -> nonzero_digit -> 3
Hex_digit -> dec_digit | a | b| c | d | e | f | A | B | C | D | E | F -> dec_digit -> nonzero_digit -> 6
Hex_digit -> dec_digit | a | b| c | d | e | f | A | B | C | D | E | F -> dec_digit -> nonzero_digit -> 6
Hex_digit -> dec_digit | a | b| c | d | e | f | A | B | C | D | E | F -> dec_digit -> nonzero_digit -> 7
Hex_digit -> dec_digit | a | b| c | d | e | f | A | B | C | D | E | F -> dec_digit -> nonzero_digit -> 1
Int_suffix -> epsilon

Example
0.001 in orange highlight:
C_constant -> fp_const
Fp_const -> dec_fp (f | F | l | L | epsilon) -> dec_fp
Dec_fp -> dec_digit * (. dec_digit | dec_digit .) dec_digit * ((E|e) exponent | epsilon)
-> dec_digit . dec_digit dec_digit dec_digit
Dec_digit -> 0 | nonzero_digit -> 0
Dec_digit -> 0 | nonzero_digit -> 0
Dec_digit -> 0 | nonzero_digit -> 0
Dec_digit -> 0 | nonzero_digit -> nonzero_digit -> 1
-> 0.001

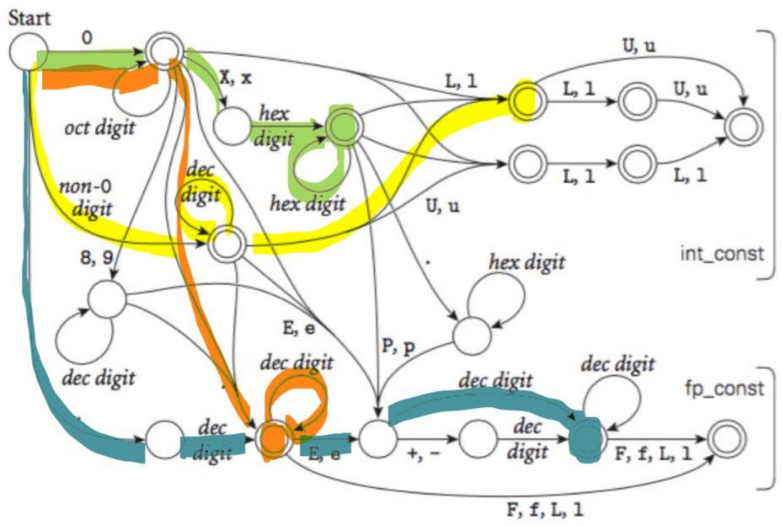
Example
23456L yellow highlight:
C_constant -> int_const
Int_const -> dec_int int_suffix
Dec_int -> nonzero_digit dec_digit* -> nonzero_digit dec_digit dec_digit dec_digit dec_digit
Nonzero_digit -> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
-> 2
Dec_digit -> 0 | nonzero_digit -> nonzero_digit -> 3
Dec_digit -> 0 | nonzero_digit -> nonzero_digit -> 4
Dec_digit -> 0 | nonzero_digit -> nonzero_digit -> 5
Dec_digit -> 0 | nonzero_digit -> nonzero_digit -> 6
Int_suffix -> l_suffix
L_suffix -> L

Example
.2E3 blue highlight:
C_constant -> fp_const

Fp_const -> dec_fp (f | F | l | L | epsilon) -> dec_fp
Dec_fp -> dec_digit * (. dec_digit | dec_digit .) dec_digit * ((E|e) exponent | epsilon)
-> . dec_digit E exponent
Dec_digit -> nonzero_digit
Nonzero_digit -> 2
Exponent -> (+ | - | epsilon) dec_digit dec_digit *
-> epsilon dec_digit
Dec_digit -> nonzero_digit
Nonzero_digit -> 3

Inconsistencies (extra credit):

1. The grammar listed allows for floating point hex constants that start with 0x or 0X, may contain any number of hex digits, may contain a dot (dot must come after a hex digit or before a hex digit), and may be followed by p or P with some exponent value after. The automaton requires a hex digit to immediately follow 0X or 0x, meaning numbers like 0xp+1 would be accepted by the grammar but not by the automaton. A simple fix would be to include the p/P transition for the state following the X/x transition.
2. Additionally, the automaton does not allow for numbers like 0x.fff since a hex digit must immediately follow the 0x according to the automaton. This could be fixed by adding a dot transition from the state following the X/x transition, which leads to a new state, which has a required hex digit transition to the state that has a loop for hex digit. This would allow floating point hex numbers that do not contain any hex digits prior to the floating point dot.
3. The automaton allows for any number of oct digits followed by a dot, E, e, X, x, L, l, U, or u. Of these, the X and x can produce invalid strings (e.g. 012321xf would lead to an accepting state). The key to fixing this is to only allow the X/x as the second character in the input, to denote hex numbers. On the state following the initial “0” transition, keep all transitions except for the loop on octal digit. Create a new state which this state can transition to by reading an octal digit. Mark this new state as accepting, copy over all transitions from the previous state (except the X/x transition), and add a loop on this state for octal digit. The net effect being that numbers containing more than 1 octal digit cannot be followed by x or X.



0x4fd3 6671

0.001

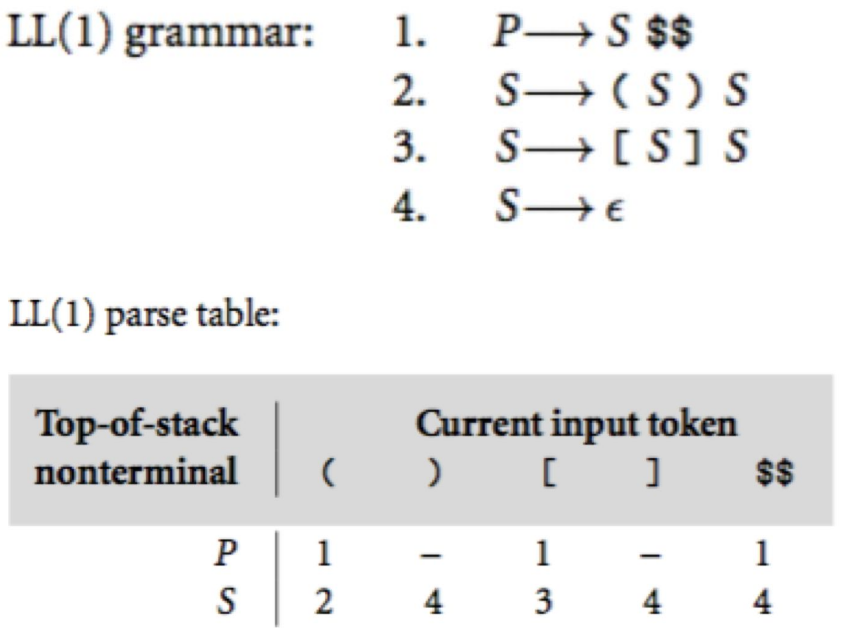
Long → 23456

0.2×10^3

Problem 2

Contributor: Shivanshi Nagar Degree: 3

LL(1)



LL(1) grammar:

1. $P \longrightarrow S$ \$\$
2. $S \longrightarrow (S) S$
3. $S \longrightarrow [S] S$
4. $S \longrightarrow \epsilon$

LL(1) parse table:

Top-of-stack nonterminal	Current input token				
	(	)	[	]	\$\$
P	1	–	1	–	1
S	2	4	3	4	4

([([()])])

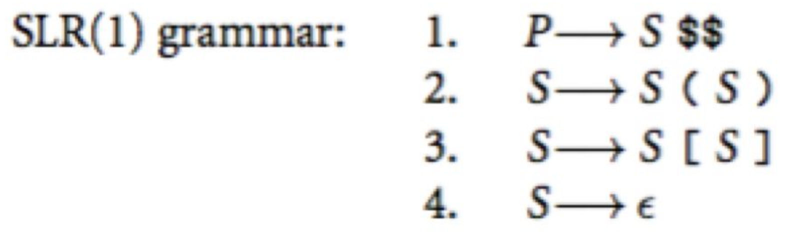
Parse Stack	Input	Comments
P	([([()])])\$\$	
S\$\$	([([()])])\$\$	Predict P -> S \$\$
(S)S\$\$	([([()])])\$\$	Predict S -> (S)S
S)S\$\$	[([()])])\$\$	Match (
[S]S)S\$\$	[([()])])\$\$	Predict S-> [S]S
S]S)S\$\$	([()])])\$\$	Match [
(S)S]S)S\$\$	([()])])\$\$	Predict S -> (S)S
S)S]S)S\$\$	[()])])\$\$	Match (

[S]S)S]S)S$$	[()])])$$	Predict S-> [S]S
S]S)S]S)S$$	()])])$$	Match [
(S)S]S)S]S)S$$	()])])$$	Predict S -> (S)S
S)S]S)S]S)S$$	)])])$$	Match (
)S]S)S]S)S$$	)])])$$	Predict S -> Epsilon
S]S)S]S)S$$	])])$$	Match)
]S)S]S)S$$	])])$$	Predict S -> Epsilon
S)S]S)S$$	)])$$	Match]
)S]S)S$$	)])$$	Predict S -> Epsilon
S]S)S$$	])$$	Match)
]S)S$$	])$$	Predict S -> Epsilon
S)S$$	)$$	Match]
)S$$	)$$	Predict S -> Epsilon
S$$	$$	Match)
$$	$$	Predict S -> Epsilon

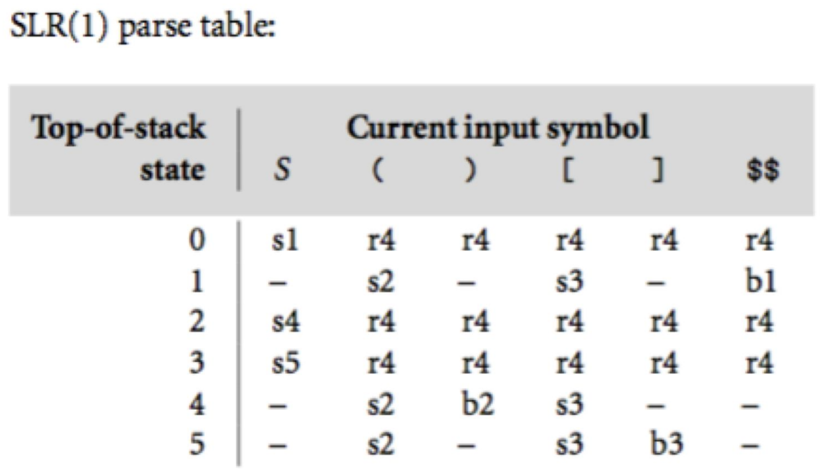
()[]()[]()

Parse Stack	Input	Comments
P	()[]()[]()$$	
S$$	()[]()[]()$$	Predict P -> S $$
(S)S$$	()[]()[]()$$	Predict S -> (S)S
S)S$$	)[]()[]()$$	Match (
)S$$	)[]()[]()$$	Predict S -> epsilon
S$$	[]()[]()$$	Match)
[S]S$$	[]()[]()$$	Predict S -> [S]S

S]S$$	]()[]()$$	Match [
]S$$	]()[]()$$	Predict S -> epsilon
S$$	()[]()$$	Match]
(S)S$$	()[]()$$	Predict S -> (S)S
S)S$$	)[]()$$	Match (
)S$$	)[]()$$	Predict S -> epsilon
S$$	[]()$$	Match)
[S]S$$	[]()$$	Predict S -> [S]S
S]S$$	]()$$	Match [
]S$$	]()$$	Predict S -> epsilon
S$$	()$$	Match]
(S)S$$	()$$	Predict S -> (S)S
S)S$$	)$$	Match (
)S$$	)$$	Predict S -> epsilon
S$$	$$	Match)
$$	$$	Predict S -> Epsilon


SLR(1) grammar:

1. $P \longrightarrow S\ \$\$$
2. $S \longrightarrow S\ (\ S\ )$
3. $S \longrightarrow S\ [\ S\ ]$
4. $S \longrightarrow \epsilon$


SLR(1) parse table:

Top-of-stack state	Current input symbol					
	S	(	)	[	]	$$
0	s1	r4	r4	r4	r4	r4
1	–	s2	–	s3	–	b1
2	s4	r4	r4	r4	r4	r4
3	s5	r4	r4	r4	r4	r4
4	–	s2	b2	s3	–	–
5	–	s2	–	s3	b3	–

([([()])])

Parse Stack	Input	Comments
0	([([()])])$$	
0	S([([()])])$$	Reduce by S -> epsilon
0S1	([([()])])$$	Shift S
0S1(2	[([()])])$$	Shift (
0S1(2	S[([()])])$$	Reduce by S -> epsilon
0S1(2S4	[([()])])$$	Shift S
0S1(2S4[3	([()])])$$	Shift [
0S1(2S4[3	S([()])])$$	Reduce by S -> epsilon
0S1(2S4[3S5	([()])])$$	Shift S
0S1(2S4[3S5(2	[()])])$$	Shift (

0S1(2S4[3S5(2	S[()])])$$	Reduce by S -> epsilon
0S1(2S4[3S5(2S4	[()])])$$	Shift S
0S1(2S4[3S5(2S4[3	()])])$$	Shift [
0S1(2S4[3S5(2S4[3	S()])])$$	Reduce by S -> epsilon
0S1(2S4[3S5(2S4[3S5	()])])$$	Shift S
0S1(2S4[3S5(2S4[3S5(2	)])])$$	Shift (
0S1(2S4[3S5(2S4[3S5(2	S)])])$$	Reduce by S -> epsilon
0S1(2S4[3S5(2S4[3S5(2S4	)])])$$	Shift S
0S1(2S4[3S5(2S4[3	])])$$	Shift and reduce by S->(S)S
0S1(2S4[3S5(2S4[3	S])])$$	Reduce by S -> epsilon
0S1(2S4[3S5(2S4[3S5	])])$$	Shift S
0S1(2S4[3S5(2	)])$$	Shift and reduce by S->[S]S
0S1(2S4[3S5(2	S)])$$	Reduce by S -> epsilon
0S1(2S4[3S5(2S4	)])$$	Shift S
0S1(2S4[3	])$$	Shift and reduce by S->(S)S
0S1(2S4[3	S])$$	Reduce by S -> epsilon
0S1(2S4[3S5	])$$	Shift S
0S1(2	)$$	Shift and reduce by S->[S]S
0S1(2	S)$$	Reduce by S -> epsilon
0S1(2S4	)$$	Shift S
0	$$	Shift and reduce by S->(S)S
0	S$$	Shift by S -> Epsilon
0S1	$$	Shift S
0	P	Shift and reduce by P -> S$$
done		

()[]()[]()

Parse Stack	Input	Comments
0	()[]()[]()$$	
0	S()[]()[]()$$	Reduce by S -> epsilon
0S1	()[]()[]()$$	Shift S
0S1(2	)[]()[]()$$	Shift (
0S1(2	S)[]()[]()$$	Reduce by S -> epsilon
0S1(2S4	)[]()[]()$$	Shift S
0	[]()[]()$$	Shift and reduce by S-> (S)S
0	S[]()[]()$$	Reduce by S -> epsilon
0S1	[]()[]()$$	Shift S
0S1[3	]()[]()$$	Shift [
0S1[3	S]()[]()$$	Reduce by S -> epsilon
0S1[3S5	]()[]()$$	Shift S
0	()[]()$$	Shift and reduce by S->[S]S
0	S()[]()$$	Reduce by S -> epsilon
0S1	()[]()$$	Shift S
0S1(2	)[]()$$	Shift (
0S1(2	S)[]()$$	Reduce by S -> epsilon
0S1(2S4	)[]()$$	Shift S
0	[]()$$	Shift and reduce by S-> (S)S
0	S[]()$$	Reduce by S -> epsilon
0S1	[]()$$	Shift S
0S1[3	]()$$	Shift [

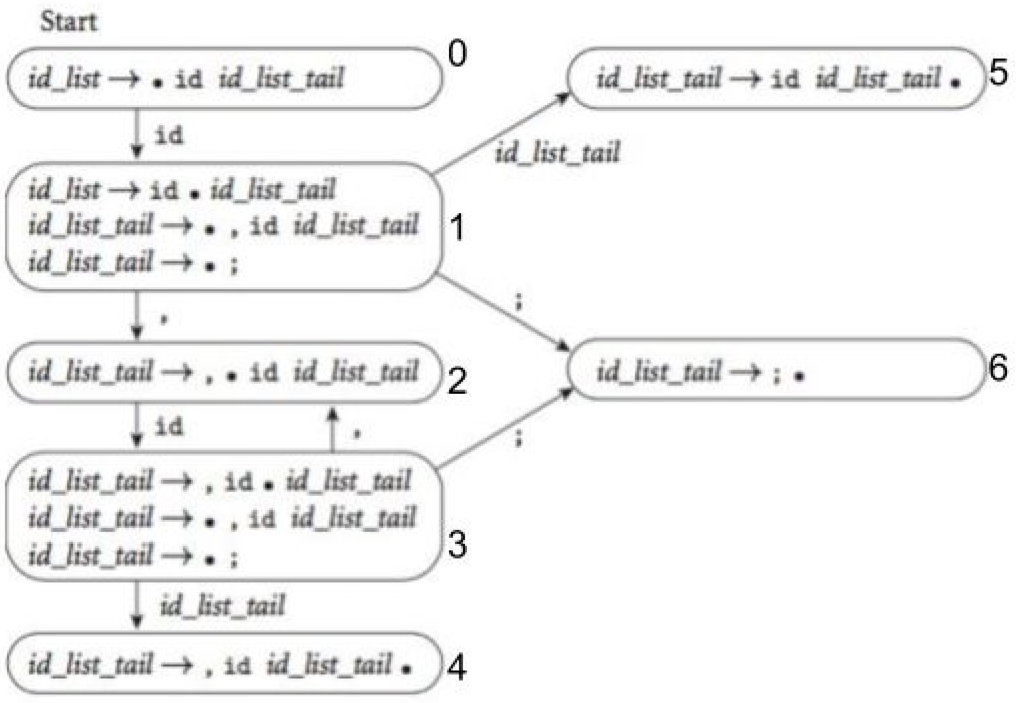
0S1[3	S]()$$	Reduce by S -> epsilon
0S1[3S5	]()$$	Shift S
0	()$$	Shift and reduce by S->[S]S
0	S()$$	Reduce by S -> epsilon
0S1	()$$	Shift S
0S1(2	)$$	Shift (
0S1(2	S)$$	Reduce by S -> epsilon
0S1(2S4	)$$	Shift S
0	$$	Shift and reduce by S-> (S)S
0	S$$	Shift by S -> Epsilon
0S1	$$	Shift S
0	P	Shift and reduce by P -> S$$
done		

Problem 3

Contributor: Stephen Hansen Degree: 3
Verify via several representative examples that the automaton copied below is a CFSM for the grammar of PLP Example 2.20 and that it can be used for parsing bottom-up with zero tokens of look-ahead.

Exercise 2.20:
id_list -> id id_list_tail
id_list_tail -> , id id_list_tail
id_list_tail -> ;

Automaton:


Start
0
id_list → • id id_list_tail
id
1
id_list → id • id_list_tail
id_list_tail → • , id id_list_tail
id_list_tail → • ;
id_list_tail
5
id_list_tail → id id_list_tail •
;
,
2
id_list_tail → , • id id_list_tail
6
id_list_tail → ; •
id
,
;
3
id_list_tail → , id • id_list_tail
id_list_tail → • , id id_list_tail
id_list_tail → • ;
id_list_tail
4
id_list_tail → , id id_list_tail •

Note that a dot appears at the end of an item in only three states, and there are no other items in these states. Lookahead is never required to resolve a shift-reduce conflict; the grammar is thus LR(0).

I have added some numbering to each state in the CFSM for better explanations of the representative examples.

The string “id ;” starts at state 0, parses “id” and goes to state 1 (shifting id onto stack), then parses “;” (shifting ; onto stack) and goes to state 6. There is no more input and state 6 contains a single transition where the dot is at the end, so the string is accepted by the grammar. The “;” at the top of the stack is reduced to id_list_tail. Going back to state 1, the stack contents “id id_list_tail” (id_list_tail being a nonterminal) are still valid input to the CFSM, as they lead to state 5 (where again, the dot in the transition is at the end). The top of the stack “id id_list_tail” must be reduced to “id_list”, which takes the parser back to state 0 (the start), and concludes the bottom-up parsing. In total, the following actions are done:

Shift “id”
Shift “;”
Reduce “;” to “id_list_tail”
Reduce “id id_list_tail” to “id_list”

The string “id , id ;” starts at state 0, parses “id” and goes to state 1 (shifting id onto stack), parses “,” and goes to state 2 (shifting , onto stack), parses “id” and goes to state 3 (shifting id onto stack), parses “;” and goes to state 6 (shifting ; onto stack). There is no more input and state 6 contains a single transition where the dot is at the end, so the string is accepted by the grammar. We reduce “;” to “id_list_tail” (going back to state 3, which a valid transition from state 3 to 4 with id_list_tail exists, so this reduction is valid (the stack as input would lead to state 4, an accepting state). Again, state 4 would mark the end of input and there is a transition with the dot at the end, so this is valid). We then reduce “, id id_list_tail” to “id_list_tail” going back to state 1 (the stack as input now leads to state 5). Finally, we reduce “id id_list_tail” to id_list and arrive back at state 0. The following actions are done by the parser:

Shift “id”
Shift “,”
Shift “id”
Shift “;”
Reduce “;” to “id_list_tail”
Reduce “, id id_list_tail” to “id_list_tail”
Reduce “id id_list_tail” to “id_list”

As a final accepted example, consider “id , id , id ;”. The state machine would go through states 0 -> 1 -> 2 -> 3 -> 2 -> 3 -> 6 and accept the string. The parser would generate the following actions:

Shift “id”
Shift “,”
Shift “id”
Shift “,”
Shift “id”
Shift “;”
Reduce “;” to “id_list_tail”
Reduce “, id id_list_tail” to “id_list_tail”
Reduce “, id id_list_tail” to “id_list_tail”
Reduce “id id_list_tail” to “id_list”

For some examples that get rejected by the CFSM, first note that any string containing a token other than “id”, “,”, or “;” is rejected automatically (no transition arrows are defined for these symbols). Taking the string “id” to start, we transition from state 0 to state 1 but state 1 contains three productions, none of which have the dot at the end of the right-hand side, so state 1 cannot accept the string. Likewise, the string “id ,” is not accepted as we transition from state 0 to state 1 to state 2, but state 2 contains no productions that have a dot at the end of the right-hand side. String “id , id” would go to state 3 from state 2, but again state 3 contains no productions that have the dot at the end of the right-hand side. The only strings that are accepted are those ending in “;” and those ending in the nonterminal “id_list_tail” (when reducing). The semicolon transition is only defined for states that have just parsed an id. In effect, these examples show that the CFSM pictured does indeed reflect the grammar of PLP Example 2.20, accepting all strings that represent lists of ids, ids separated by commas and ending the list with a semicolon.

Regarding parsing bottom-up with zero tokens of lookahead. The examples above simulate a LR(0) grammar using shift and reduce to construct the parse tree from bottom up. In essence, this CFSM proves that the grammar requires no additional lookahead to determine which action to take at any given moment. At any moment during parsing, there is only one applicable action that the parser may take for this grammar. In states 4, 5, and 6, the states contain a production where the dot is at the end of the right-hand side. A reduction is necessary for these states and since each of these states only contains one production each, then the choice of reduction is forced for each state. In states 0, 1, 2, and 3, when shifting symbols, there is only one unique transition per symbol when shifting, which forces a deterministic traversal of the CFSM (and thus only one valid choice of shift per step). The parser is forced to shift the entire string into the stack prior to doing reductions (the parse tree always branches right, so if we are parsing bottom-up then we need to reach the end of the input to get the leaves), and each reduction is forced by its corresponding state (4, 5, or 6). There is no ambiguity and never any conflict between shift and reduce actions when choosing a next action; the action is always forced and no additional tokens of lookahead are necessary at any given moment to determine which action to take. Thus this CFSM can be used to parse bottom-up with no shift-reduce conflicts arising, reducing the need to utilize lookahead to determine actions.