CSL316 - LANGUAGE PROCESSOR

ASSIGNMENT-2

Name: **Sariputi Naga Swathi**
Enrolment Number: **BT19CSE098**

Grammar Assignment: Given the grammar in the file, generate the parsing table also output FIRST and FOLLOW sets for each non-terminal:
Roll Numbers ending with: 1, 3, 7: LL
The input file format will be as below (Where EPS stands for EPSILON):
##
TERMINALS ( )
NONTERMINALS S
S -> (S)
S -> EPS
##

Solution :
As my enrollment number ends with 8, I have written a program for SLR.

Imported argparse package to get input from the user through command line arguments. Created an object of ArgumentParser as the parser and then added arguments with the add_argument method. The first argument is -g which is an optional argument that stores true if the user gives this argument in command, the second argument is for taking input file and the third argument is -input which is an optional argument that stores true if the user gives this argument in command. Then to get grammar from a given input file, created an object of the Grammar class by passing the input file as an argument to the constructor.
Then to get first, follow, parse table of given grammar for SLR, created object of SLRParser class by passing the grammar as an argument to the constructor and called print_info method.
If the user gives -g then we generate transition states in graph form by calling the generate_automaton() method.
If the user gives -input then takes the input string from the user and print the parser table for that input.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('grammar_file', type=argparse.FileType('r'), help='text file to be used as grammar')
parser.add_argument('-g', action='store_true', help='generate automaton')
parser.add_argument('-input', action='store_true', help='input to parse')
args = parser.parse_args()
G = Grammar(args.grammar_file.read())
slr_parser = SLRParser(G)
slr_parser.print_info()
if args.input:
    print("Enter input string")
    tokens=input()
    results = slr_parser.LR_parser(tokens)
    print("Input string ",tokens,"parse table")
    slr_parser.print_LR_parser(results)
if args.g:
    slr_parser.generate_automaton()
```

Grammar :

constructor __init__() is used to read input from the file and created productions, terminals, non-terminals, and a start symbol. Input is read line to line and then productions are created from that line, if there '|' in the line, we will split there and create another production.And also while reading the line we check the characters in it. If it is a Capital letter we will add it into the non-terminals set, otherwise into terminals set, except for '->' and epsilon('^').

We will be doing partition at '->' and storing the left part in the head, and the right part in the bodies

```
class Grammar:
    def __init__(self, grammar_str):
        self.grammar_str = '\n'.join(filter(None, grammar_str.splitlines()))
        self.grammar = {}
        self.start = None
        self.terminals = set()
        self.nonterminals = set()
        for production in list(filter(None, grammar_str.splitlines())):
            head, _, bodies = production.partition(' -> ')
            if not head.isupper():
                raise ValueError(
                    f'\'{head} -> {bodies}\': Head \'{head}\' is not capitalized to be treated as a nonterminal.')
            if not self.start:
                self.start = head
            self.grammar.setdefault(head, set())
            self.nonterminals.add(head)
            bodies = {tuple(body.split()) for body in ' '.join(bodies.split()).split('|')}
            for body in bodies:
                if '^' in body and body != ('^',):
                    raise ValueError(f'\'{head} -> {" ".join(body)}\': Null symbol \'^\' is not allowed here.')
                self.grammar[head].add(body)
                for symbol in body:
                    if not symbol.isupper() and symbol != '^':
                        self.terminals.add(symbol)
                    elif symbol.isupper():
                        self.nonterminals.add(symbol)
        self.symbols = self.terminals | self.nonterminals
```

SLRParser:

constructor _init_:

We will create an object of a Grammar class, by passing the "start' -> start" string and adding this string to our grammar. Generated first and follow sets of the grammar. Then in the action set, we added terminals and a '$' and in the goto set, we added non-terminals except the start symbol.

Then add these two sets to get parse_table_symbols, and then call construct_table() function to generate a parse table.

```
class SLRParser:
    def __init__(self, G):
        self.G_prime = Grammar(f"{G.start}' -> {G.start}\n{G.grammar_str}")
        self.max_G_prime_len = len(max(self.G_prime.grammar, key=len))
        self.G_indexed = []
        for head, bodies in self.G_prime.grammar.items():
            for body in bodies:
                self.G_indexed.append([head, body])
        self.first, self.follow = first_follow(self.G_prime)
        self.C = self.items(self.G_prime)
        self.action = list(self.G_prime.terminals) + ['$']
        self.goto = list(self.G_prime.nonterminals - {self.G_prime.start})
        self.parse_table_symbols = self.action + self.goto
        self.parse_table = self.construct_table()
```

FIRST AND FOLLOW:

The logic for finding first:
First, we will take non-terminal, and if in its production if the first character is a terminal and add this terminal to first of this non-terminal else if the character is nonterminal add first of this non-terminal to first. If the first of this non-terminal contain epsilon, check for the next character and repeat.
The logic for finding follow:
First, to start non-terminal, we add $ to follow.
So to find follow of any non-terminal, if next to it is non-terminal, find first of it and add it to follow of this non-terminal else if next is terminal then add it to follow directly. Here if we get the first of the next non-terminal as epsilon instead of adding it to follow we move to the next symbol and repeat the same. If the right-side of the non-terminal is empty then we add follow of the leftmost non-terminal to follow.

```
def first_follow(G):
    def union(set_1, set_2):
        set_1_len = len(set_1)
        set_1 |= set_2
        return set_1_len != len(set_1)
    first = {symbol: set() for symbol in G.symbols}
    first.update((terminal, {terminal}) for terminal in G.terminals)
    follow = {symbol: set() for symbol in G.nonterminals}
    follow[G.start].add('$')
    while True:
        updated = False
        for head, bodies in G.grammar.items():
            for body in bodies:
                for symbol in body:
                    if symbol != '^':
                        updated |= union(first[head], first[symbol] - set('^'))
                        if '^' not in first[symbol]:
                            break
                    else:
                        updated |= union(first[head], set('^'))
                else:
                    updated |= union(first[head], set('^'))
                aux = follow[head]
                for symbol in reversed(body):
                    if symbol == '^':
                        continue
                    if symbol in follow:
                        updated |= union(follow[symbol], aux - set('^'))
                    if '^' in first[symbol]:
                        aux = aux | first[symbol]
                    else:
                        aux = first[symbol]
        if not updated:
            return first, follow
```

Parse Table:

Here in order to create a parse table, first we find items of transition states by items method in which closure method is called for placing dots appropriately.
So as this is a bottom-up approach first we put a dot for every production at the start of the right side(after partition) and we have to create the next states by moving the dot forward with the goto method.Then using these states, and action and goto we can construct a parsing table for SLR grammar based on terminals and decide whether it is a shift or reduce or state transition.

```
    def CLOSURE(self, I):
        J = I
        while True:
            item_len = len(J)
            for head, bodies in J.copy().items():
                for body in bodies.copy():
                    if '.' in body[:-1]:
                        symbol_after_dot = body[body.index('.') + 1]
                        if symbol_after_dot in self.G_prime.nonterminals:
                            for G_body in self.G_prime.grammar[symbol_after_dot]:
                                J.setdefault(symbol_after_dot, set()).add(
                                    ('.',) if G_body == ('^',) else ('.',) + G_body)
            if item_len == len(J):
                return J
    def GOTO(self, I, X):
        goto = {}
        for head, bodies in I.items():
            for body in bodies:
                if '.' in body[:-1]:
                    dot_pos = body.index('.')
                    if body[dot_pos + 1] == X:
                        replaced_dot_body = body[:dot_pos] + (X, '.') + body[dot_pos + 2:]
                        for C_head, C_bodies in self.CLOSURE({head: {replaced_dot_body}}).items():
                            goto.setdefault(C_head, set()).update(C_bodies)
        return goto
```

```
    def items(self, G_prime):
        C = [self.CLOSURE({G_prime.start: {('.', G_prime.start[:-1])}})]
        while True:
            item_len = len(C)
            for I in C.copy():
                for X in G_prime.symbols:
                    goto = self.GOTO(I, X)
                    if goto and goto not in C:
                        C.append(goto)
            if item_len == len(C):
                return C
```

```
    def construct_table(self):
        parse_table = {r: {c: '' for c in self.parse_table_symbols} for r in range(len(self.C))}
        for i, I in enumerate(self.C):
            for head, bodies in I.items():
                for body in bodies:
                    if '.' in body[:-1]:
                        symbol_after_dot = body[body.index('.') + 1]
                        if symbol_after_dot in self.G_prime.terminals:
                            s = f's{self.C.index(self.GOTO(I, symbol_after_dot))}'
                            if s not in parse_table[i][symbol_after_dot]:
                                if 'r' in parse_table[i][symbol_after_dot]:
                                    parse_table[i][symbol_after_dot] += '/'
                                parse_table[i][symbol_after_dot] += s
                    elif body[-1] == '.' and head != self.G_prime.start:  # CASE 2 b
                        for j, (G_head, G_body) in enumerate(self.G_indexed):
                            if G_head == head and (G_body == body[:-1] or G_body == ('^',) and body == ('.',)):
                                for f in self.follow[head]:
                                    if parse_table[i][f]:
                                        parse_table[i][f] += '/'
                                    parse_table[i][f] += f'r{j}'
                                break
                    else:
                        parse_table[i]['$'] = 'acc'
            for A in self.G_prime.nonterminals:
                j = self.GOTO(I, A)
                if j in self.C:
                    parse_table[i][A] = self.C.index(j)
        return parse_table
```

print_table():

In this function, we will Print all the requirements such as Augmented Grammar, Terminals, Non-Terminals, Symbols, Parsing Table.

```
    def print_info(self):
        def fprint(text, variable):
            print(f'{text:>12}: {", ".join(variable)}')
        def print_line():
            print(f'+{("-" * width + "+") * (len(list(self.G_prime.symbols) + ["$"]))}')
        def symbols_width(symbols):
            return (width + 1) * len(symbols) - 1
        print('AUGMENTED GRAMMAR:')
        for i, (head, body) in enumerate(self.G_indexed):
            print(f'{i:>{len(str(len(self.G_indexed) - 1))}}: {head:>{self.max_G_prime_len}} -> {" ".join(body)}')
        print()
        fprint('TERMINALS', self.G_prime.terminals)
        fprint('NONTERMINALS', self.G_prime.nonterminals)
        fprint('SYMBOLS', self.G_prime.symbols)
        print('\nFIRST:')
        for head in self.G_prime.grammar:
            print(f'{head:>{self.max_G_prime_len}} = {{ {", ".join(self.first[head])} }}')
        print('\nFOLLOW:')
        for head in self.G_prime.grammar:
            print(f'{head:>{self.max_G_prime_len}} = {{ {", ".join(self.follow[head])} }}')
        width = max(len(c) for c in {'ACTION'} | self.G_prime.symbols) + 2
        for r in range(len(self.C)):
            max_len = max(len(str(c)) for c in self.parse_table[r].values())
            if width < max_len + 2:
                width = max_len + 2
        print('\nPARSING TABLE:')
        print(f'+{"-" * width}+{"-" * symbols_width(self.action)}+{"-" * symbols_width(self.goto)}+')
        print(f'|{"":{width}}|{"ACTION":^{symbols_width(self.action)}}|{"GOTO":^{symbols_width(self.goto)}}|')
        print(f'|{"STATE":^{width}}+{("-" * width + "+") * len(self.parse_table_symbols)}')
        print(f'|{"":^{width}}|', end=' ')
        for symbol in self.parse_table_symbols:
            print(f'{symbol:^{width - 1}}|', end=' ')
        print()
        print_line()
        for r in range(len(self.C)):
            print(f'|{r:^{width}}|', end=' ')
            for c in self.parse_table_symbols:
                print(f'{self.parse_table[r][c]:^{width - 1}}|', end=' ')
            print()
```

generate_automaton():

We create an object of the Digraph class. Here we will do automation to create a graph, representing all states and transitions, using the Items, Terminals, Non-Terminals, parse table. Then by using matplotlib library, we will plot the graph.

```
    def generate_automaton(self):
        automaton = Digraph('automaton', node_attr={'shape': 'record'})
        for i, I in enumerate(self.C):
            I_html = f'<<I>I</I><SUB>{i}</SUB><BR/>'
            for head, bodies in I.items():
                for body in bodies:
                    I_html += f'<I>{head:>{self.max_G_prime_len}}</I> &#8594;'
                    for symbol in body:
                        if symbol in self.G_prime.nonterminals:
                            I_html += f' <I>{symbol}</I>'
                        elif symbol in self.G_prime.terminals:
                            I_html += f' <B>{symbol}</B>'
                        else:
                            I_html += f' {symbol}'
                    I_html += '<BR ALIGN="LEFT"/>'
            automaton.node(f'I{i}', f'{I_html}>')
        for r in range(len(self.C)):
            for c in self.parse_table_symbols:
                if isinstance(self.parse_table[r][c], int):
                    automaton.edge(f'I{r}', f'I{self.parse_table[r][c]}', label=f'<<I>{c}</I>>')
                elif 's' in self.parse_table[r][c]:
                    i = self.parse_table[r][c][self.parse_table[r][c].index('s') + 1:]
                    if '/' in i:
                        i = i[:i.index('/')]
                    automaton.edge(f'I{r}', f'I{i}', label=f'<<B>{c}</B>>' if c in self.G_prime.terminals else c)
                elif self.parse_table[r][c] == 'acc':
                    automaton.node('acc', '<<B>accept</B>>', shape='none')
                    automaton.edge(f'I{r}', 'acc', label='$')
        s = Source(str(automaton), filename="test.gv", format="png")
        #s.view()
        img = mpimg.imread('test.gv.png')
        imgplot = plt.imshow(img)
        plt.show()
```

Input string check using LR Parser:

First, we split the given string into input string concatenated with $ and stored in buffer and then Initiated stack, symbols, and results then run a while loop to check if the given input string is accepted by the grammar or not. Here in loop, if the symbol is not recognized then we stop the loop and add an error message in results and this implies is string is not accepted else we check for input acceptance using parse table and if it is not in parse table then input string is not accepted by grammar else if '/' in parse table then reduce conflict at that state else if parse table starts with 's' or 'r' for that character then we append the state transition to results with shift or reduce respectively and at last if true for all characters then we append 'acc' which indicates string accepted.

```
    def LR_parser(self, w):
        buffer = f'{w} $'.split()
        pointer = 0
        a = buffer[pointer]
        stack = ['0']
        symbols = ['']
        results = {'step': [''], 'stack': ['STACK'] + stack, 'symbols': ['SYMBOLS'] + symbols, 'input': ['INPUT'],'action': ['ACTION']}
        step = 0
        while True:
            s = int(stack[-1])
            step += 1
            results['step'].append(f'({step})')
            results['input'].append(' '.join(buffer[pointer:]))
            if a not in self.parse_table[s]:
                results['action'].append(f'ERROR: unrecognized symbol {a}')
                break
            elif not self.parse_table[s][a]:
                results['action'].append('ERROR: input cannot be parsed by given grammar')
                break
            elif '/' in self.parse_table[s][a]:
                action = 'reduce' if self.parse_table[s][a].count('r') > 1 else 'shift'
                results['action'].append(f'ERROR: {action}-reduce conflict at state {s}, symbol {a}')
                break
            elif self.parse_table[s][a].startswith('s'):
                results['action'].append('shift')
                stack.append(self.parse_table[s][a][1:])
                symbols.append(a)
                results['stack'].append(' '.join(stack))
                results['symbols'].append(' '.join(symbols))
                pointer += 1
                a = buffer[pointer]
            elif self.parse_table[s][a].startswith('r'):
                head, body = self.G_indexed[int(self.parse_table[s][a][1:])]
                results['action'].append(f'reduce by {head} -> {" ".join(body)}')
                if body != ('^',):
                    stack = stack[:-len(body)]
                    symbols = symbols[:-len(body)]
                stack.append(str(self.parse_table[int(stack[-1])][head]))
                symbols.append(head)
                results['stack'].append(' '.join(stack))
                results['symbols'].append(' '.join(symbols))
            elif self.parse_table[s][a] == 'acc':
                results['action'].append('accept')
                break
        return results
```

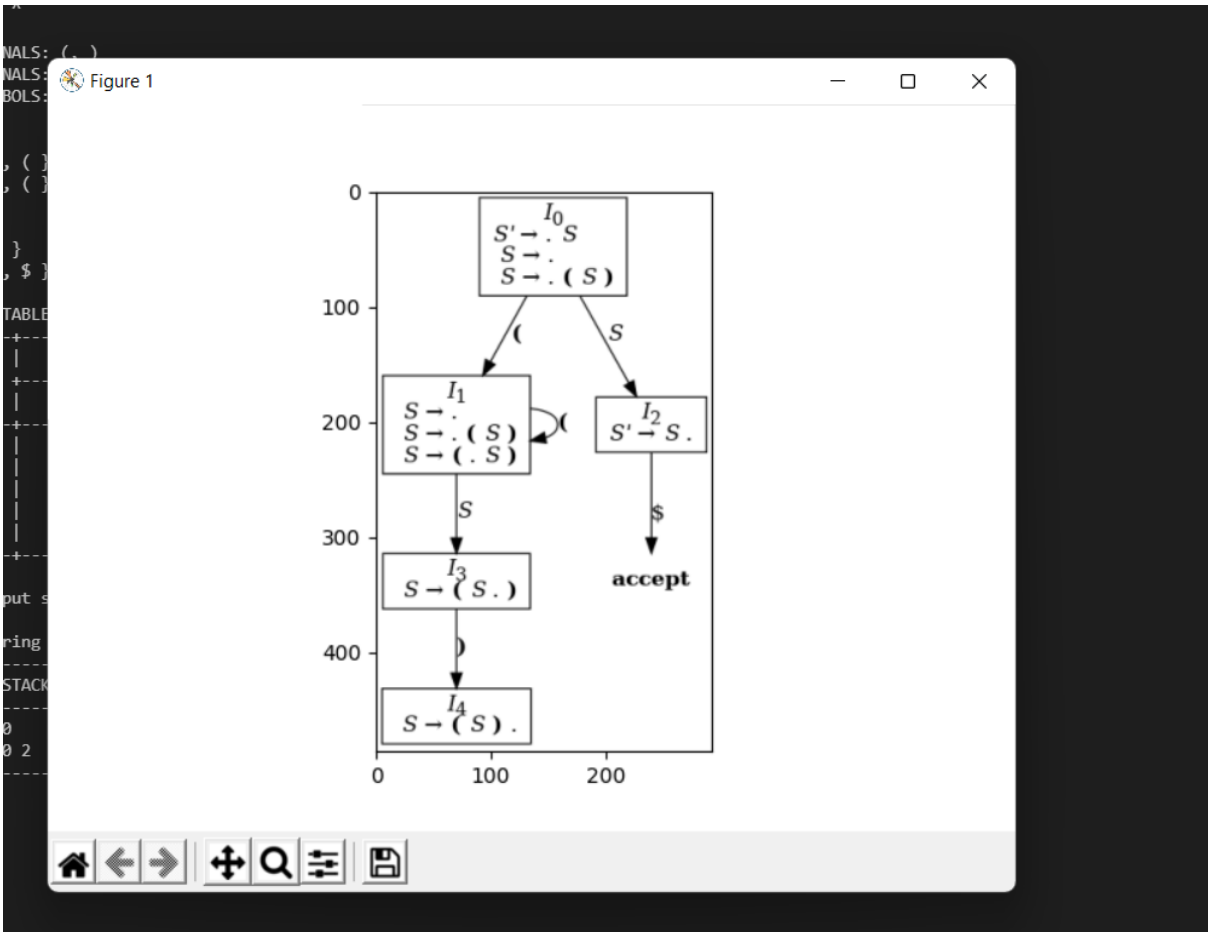
print_LR_parser():

Here we will print input string parsing for the given grammar in the table format.

```
    def print_LR_parser(self, results):
        def print_line():
            print(f'{"".join(["+" + ("-" * (max_len + 2)) for max_len in max_lens.values()])}+')
        max_lens = {key: max(len(value) for value in results[key]) for key in results}
        justs = {'step': '>', 'stack': '', 'symbols': '', 'input': '>', 'action': ''}
        print_line()
        print(''.join([f'| {history[0]:^{max_len}} ' for history, max_len in zip(results.values(), max_lens.values())]) + '|')
        print_line()
        for i, step in enumerate(results['step'][:-1], 1):
            print(''.join([f'| {history[i]:{just}{max_len}} ' for history, just, max_len in
                           zip(results.values(), justs.values(), max_lens.values())]) + '|')
        print_line()
```

OUTPUT:

```
PS G:\6sem\lp\A2\SLR\py-slr> python SLRParser.py -g text.txt -input
AUGMENTED GRAMMAR:
0: S' -> S
1:  S -> ( S )
2:  S -> ^

   TERMINALS: ), (
NONTERMINALS: S, S'
     SYMBOLS: ), (, S, S'

FIRST:
S' = { ^, ( }
 S = { ^, ( }

FOLLOW:
S' = { $ }
 S = { $, ) }

PARSING TABLE:
+--------+--------------------------+--------+
|        |          ACTION          |  GOTO  |
| STATE  +--------+--------+--------+--------+
|        |   )    |   (    |   $    |   S    |
+--------+--------+--------+--------+--------+
|   0    |   r2   |   s1   |   r2   |   2    |
|   1    |   r2   |   s1   |   r2   |   3    |
|   2    |        |        |  acc   |        |
|   3    |   s4   |        |        |        |
|   4    |   r1   |        |   r1   |        |
+--------+--------+--------+--------+--------+

Enter input string
( )
Input string  ( ) parse table
+-----+---------+---------+-------+----------------------+
|     |  STACK  | SYMBOLS | INPUT |        ACTION        |
+-----+---------+---------+-------+----------------------+
| (1) | 0       |         | ( ) $ | shift                |
| (2) | 0 1     |  (      |   ) $ | reduce by S -> ^     |
| (3) | 0 1 3   |  ( S    |   ) $ | shift                |
| (4) | 0 1 3 4 |  ( S )  |     $ | reduce by S -> ( S ) |
| (5) | 0 2     |  S      |     $ | accept               |
+-----+---------+---------+-------+----------------------+
PS G:\6sem\lp\A2\SLR\py-slr>
```



Invalid Input string:

```
Enter input string
( ) ( )
Input string  ( ) ( ) parse table
+-----+---------+---------+-----------+-------------------------------------------------+
|     |  STACK  | SYMBOLS |   INPUT   |                     ACTION                      |
+-----+---------+---------+-----------+-------------------------------------------------+
| (1) | 0       |         | ( ) ( ) $ | shift                                           |
| (2) | 0 1     |  (      |   ) ( ) $ | reduce by S -> ^                                |
| (3) | 0 1 3   |  ( S    |   ) ( ) $ | shift                                           |
| (4) | 0 1 3 4 |  ( S )  |     ( ) $ | ERROR: input cannot be parsed by given grammar  |
+-----+---------+---------+-----------+-------------------------------------------------+
PS G:\6sem\lp\A2\SLR\py-slr>
```

—----------THE END—----------