IMPLEMENTATION OF PYTHON PROGRAM TO FIND LR(K) TRANSITION STATES AND PARSE TABLE

Shanmukha Sudha Kiran .T -211FA04003; RaviKiran .K-211FA04043; Deepthi .U -211FA04046; Sahil Raj -211FA04677

BATCH-17,3Rd B. TECH, CSE BRANCH, VFSTR DEEMED TO BE UNIVERSITY

ABSTRACT:

In this paper, we begin by introducing the concept of Augmented Grammar, LR(0) states, and transitions. We then proceed to illustrate how these concepts are utilized to construct a parsing table, which proves to be instrumental in the verification of input strings.

T1 QUESTION:

Consider the following expression grammar: Terminals = {num, +, *, $} Non-Terminals = {E' , E} Rules = E' → E$; E → num; E → E + E; E → E * E Start Symbol = E'

a. Develop Augmented Grammar

b. Create the LR(0) states and transitions for this grammar

c. Since this grammar is ambiguous, it is not LR(k), for any k. We can, however, create a parse table for the grammar, using precedence rules to determine when to shift, and when to reduce. Assuming that * binds more strongly than +, and both are left-associative, create a parse table using the LR(0) states and transitions.

d. Test your parse table with the following strings: num * num + num num + num * num num + num + num e. Now create an LR parse table for this grammar, this time assuming that + binds more strongly than *, and both are right-associative. Test your parse table with the following strings:

INTRODUCTION:

AUGMENTED GRAMMAR:

An augmented grammar is any grammar whose productions are augmented with

conditions expressed using features**.** Features may be associated with any nonterminal symbol in a derivation.

LR(k):

The LR parser is an efficient bottom-up syntax analysis technique that can be used for a large class of context-free grammar. This technique is also called LR **(**k) parsing.

1) L stands for the left to right scanning
2) R stands for rightmost derivation in reverse

   Where ,K stands for no. of input symbols of lookahead.

K=0,1

Where K stands for Look ahead values.

PARSING TABLE:

A parsing table is a tool used in predictive parsing, which is a top-down parsing technique. It is a table that maps non-terminal symbols and lookahead tokens to parsing actions. The parsing table addresses the whole parsing calculation in a plain configuration, with clear mappings between non-terminal images, lookahead tokens, and parsing activities.

ALGORITHM:

1. Start
2. Define a class `Item` to represent production rules in the form `lhs -> rhs` with a dot indicating the position of parsing.
3. Define a class `Grammar` that takes the start symbol and a set of production rules.
4. Implement a method `closure` in the `Grammar` class to calculate the closure of a set of items. This method is used to find the items that can be derived from the current items by applying productions.
5. Implement a method `goto` in the `Grammar` class to calculate the transition from one set of items to another based on a given symbol. This is used to create transitions in the finite automaton.
6. Implement a method `items` in the `Grammar` class to generate the LR(0) parser automaton. This method iterates over states and symbols to generate closures and transitions.
7. Define functions `print_states` and `print_transitions` to print the generated states and transitions.
8. Define the grammar rules for a sample expression language in the `rules` dictionary.
9. Create an instance of the `Grammar` class with the start symbol and rules.
10. Generate states and transitions for the LR(0) parser automaton.
11. Print the generated states and transitions.
12. Stop

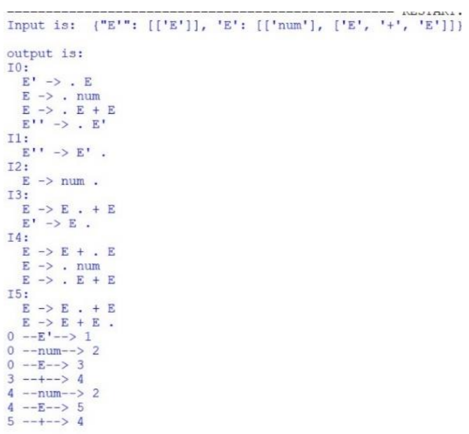
SOURCE CODE:

```
from collections import deque
class Item:
    def _init_(self, lhs, rhs, dot):
        self.lhs = lhs
        self.rhs = rhs
        self.dot = dot
    def _eq_(self, other):
        return self.lhs == other.lhs and self.rhs == other.rhs and self.dot == other.dot
    def _hash_(self):
        return hash((self.lhs, tuple(self.rhs), self.dot))
    def _str_(self):
        rhs = list(self.rhs)
        rhs.insert(self.dot, '.')
        return f"{self.lhs} -> {' '.join(rhs)}"
class Grammar:
    def _init_(self, start_symbol, rules):
        self.start_symbol = start_symbol
        self.rules = rules
    def closure(self, items):
        queue = deque(items)
        closure = set(items)
        while queue:
            item = queue.popleft()
            if item.dot < len(item.rhs) and item.rhs[item.dot] in self.rules:
                for rule in self.rules[item.rhs[item.dot]]:
                    new_item = Item(item.rhs[item.dot], rule, 0)
                    if new_item not in closure:
                        closure.add(new_item)
                        queue.append(new_item)
        return frozenset(closure)
    def goto(self, items, symbol):
        next_items = set()
        for item in items:
            if item.dot < len(item.rhs) and
item.rhs[item.dot] == symbol:
                next_items.add(Item(item.lhs,
item.rhs, item.dot + 1))
        return self.closure(next_items)
    def items(self):
        symbols = set()
        for lhs in self.rules:
            symbols.add(lhs)
            for rule in self.rules[lhs]:
                symbols.update(rule)


        start_item = Item(self.start_symbol +
"'", [self.start_symbol], 0)
        start_closure =
self.closure([start_item])
        queue = deque([start_closure])
        closures = {start_closure: 0}
        transitions = {}
        index = 1
        while queue:
            closure = queue.popleft()
            for symbol in symbols:
                next_closure = self.goto(closure,
symbol)
```

```
                if next_closure:
                    if next_closure not in closures:
closures[next_closure] = index
                        index += 1
queue.append(next_closure)
transitions[closures[closure], symbol] =
closures[next_closure]
        return closures.keys(), transitions
def print_states(states):
    for i, state in enumerate(states):
        print(f"I{i}:")
        for item in state:
            print(f"  {item}")
def print_transitions(transitions):
    for key in transitions:
        print(f"{key[0]} --{key[1]}-->
{transitions[key]}")


rules = {
    "E'": [["E", "*", "E"]],
    "E": [["num"], ["E", "+", "E"]]
}
print("input is: ",rules)
print()
print("output is: ")
grammar = Grammar("E'", rules)
states, transitions = grammar.items()
print_states(states)
print_transitions(transitions)
```

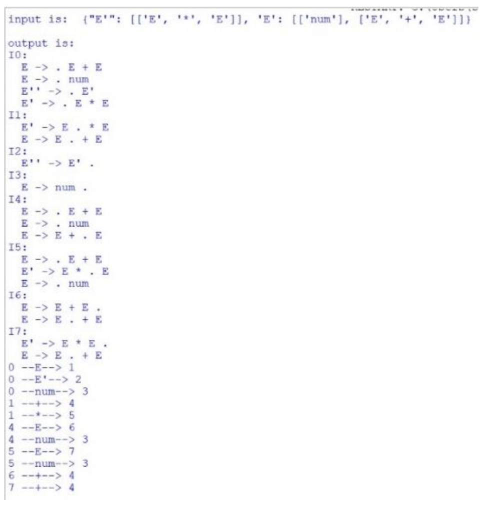
OUTPUTS:

1)



```
Input is:  {"E'": [['E']], 'E': [['num'], ['E', '+', 'E']]}

output is:
I0:
  E' -> . E
  E -> . num
  E -> . E + E
  E'' -> . E'
I1:
  E'' -> E' .
I2:
  E -> num .
I3:
  E -> E . + E
  E' -> E .
I4:
  E -> E + . E
  E -> . num
  E -> . E + E
I5:
  E -> E . + E
  E -> E + E .
0 --E'--> 1
0 --num--> 2
0 --E--> 3
3 --+--> 4
4 --num--> 2
4 --E--> 5
5 --+--> 4
```

2)



```
input is:  {"E'": [['E', '*', 'E']], 'E': [['num'], ['E', '+', 'E']]}

output is:
I0:
  E -> . E + E
  E -> . num
  E'' -> . E'
  E' -> . E * E
I1:
  E' -> E . * E
  E -> E . + E
I2:
  E'' -> E' .
I3:
  E -> num .
I4:
  E -> . E + E
  E -> . num
  E -> E + . E
I5:
  E -> . E + E
  E' -> E * . E
  E -> . num
I6:
  E -> E + E .
  E -> E . + E
I7:
  E' -> E * E .
  E -> E . + E
0 --E--> 1
0 --E'--> 2
0 --num--> 3
1 --+--> 4
1 --*--> 5
4 --E--> 6
4 --num--> 3
5 --E--> 7
5 --num--> 3
6 --+--> 4
7 --+--> 4
```

TIME COMPLEXITY:

Overall, the time complexity of the entire LR(0) parsing automaton construction can be approximated as O(N * M), where N is the number of states and M is the average number of items and production rules in each state. This is a simplified estimation, as factors like the specific grammar, the number of symbols, and the structure of production rules can affect the actual time complexity.

For your specific grammar and set of rules, the time complexity can vary based on the number of states and items in the closure sets, as well as the number of symbols in the grammar.

CONCLUSION:

In conclusion, our exploration of Augmented Grammar, LR(0) states, transitions, and the construction of a parsing table highlights their significance in parsing and verifying input strings. The parsing table serves as a valuable tool for efficiently determining the validity of various input strings according to the grammar rules defined. This research enhances our understanding of formal language theory and parsing techniques, contributing to the broader field of computational linguistics and compiler design.

REFERENCE:

1) https://www.geeksforgeeks.org/lr-parser/
2) https://www.javatpoint.com/lr-parser
3) https://www.codingninjas.com/studio/library/lr-parsing