Shift-Reduce Parsing

```
terminals = {'a', 'b'}
non_terminals = {'S'}
grammar = {
    ('S', ('a', 'S', 'b')),
    ('S', ())
}
```

Here, we define the terminals and non-terminals of the grammar, as well as the production rules in the form of a set of tuples. Here, we have a grammar with a single non-terminal symbol S, and two productions: S -> aSb and S -> epsilon (where epsilon represents the empty string).

```
stack = ['S']
input_str = list('aabbb')
```

We initialize the stack with the start symbol S and convert the input string to a list.

```
while stack:
    top = stack[-1]
    if top in terminals:
        if input_str and top == input_str[0]:
            stack.pop()
            input_str = input_str[1:]
        else:
            print('The input string is invalid.')
            break
    elif top in non_terminals:
        match_found = False
        for production in grammar:
            if production[0] == top:
                rhs = production[1]
                if not rhs:
                    stack.pop()
                    match_found = True
                    break
                if all([symbol in terminals for symbol in rhs]):
                    if rhs == input_str[:len(rhs)]:
                        stack.pop()
                        input_str = input_str[len(rhs):]
                        match_found = True
                        break
                elif all([symbol in terminals.union(non_terminals) for symbol in rhs]):
                    stack.pop()
                    stack.extend(reversed(rhs))
                    match_found = True
                    break
        if not match_found:
            print('The input string is invalid.')
            break
```

1. while stack: sets up a loop that runs as long as the stack is not empty. The loop will continue until either the stack is empty, or the input string is found to be invalid.
2. top = stack[-1] assigns the value of the top element in the stack to the variable top.
3. if top in terminals: checks if the top element is a terminal symbol. If it is, then the program checks whether the first character of the input string is the same as the top symbol. If it is, then the program removes the top element from the stack and removes the first character from the input string. If it isn't, the program prints a message indicating that the input string is invalid and exits the loop.
4. elif top in non_terminals: checks if the top element is a non-terminal symbol. If it is, then the program searches through the grammar rules for a rule that has the top symbol as the left-hand side. If a matching rule is found, the right-hand side (RHS) of the rule is assigned to the variable rhs.
5. if not rhs: checks if the rhs is empty. If it is, that means that the rule matches an empty string, so the program removes the top element from the stack.
6. if all([symbol in terminals for symbol in rhs]): checks if all the symbols in rhs are terminals. If they are, then the program checks if the first characters of the input string match rhs. If they do, the program removes the top element from the stack, and removes the characters in

rhs from the input string.

7. elif all([symbol in terminals.union(non_terminals) for symbol in rhs]): checks if all the symbols in rhs are either terminals or non-terminals. If they are, then the program removes the top element from the stack, and pushes the symbols in rhs onto the stack in reverse order.
8. if not match_found: checks if the program has found a matching production rule for the top symbol. If it hasn't, then the input string is invalid, so the program prints a message and exits the loop.
9. If the loop completes successfully, i.e., all symbols in the stack have been processed and the input string is empty, then the program prints a message indicating that the input string is valid. If the stack is not empty, or the input string is not empty, then the program prints a message indicating that the input string is invalid.

```
terminals = {'a', 'b'}
non_terminals = {'S'}
grammar = {
    ('S', ('a', 'S', 'b')),
    ('S', ())
}

stack = ['S']
input_str = list('aabbb')

while stack:
    top = stack[-1]
    if top in terminals:
        if input_str and top == input_str[0]:
            stack.pop()
            input_str = input_str[1:]
        else:
            print('The input string is invalid.')
            break
    elif top in non_terminals:
        match_found = False
        for production in grammar:
            if production[0] == top:
                rhs = production[1]
                if not rhs:
                    stack.pop()
                    match_found = True
                    break
                if all([symbol in terminals for symbol in rhs]):
                    if rhs == input_str[:len(rhs)]:
                        stack.pop()
                        input_str = input_str[len(rhs):]
                        match_found = True
                        break
                elif all([symbol in terminals.union(non_terminals) for symbol in rhs]):
                    stack.pop()
                    stack.extend(reversed(rhs))
                    match_found = True
                    break
        if not match_found:
            print('The input string is invalid.')
            break
if not stack and not input_str:
    print('The input string is invalid.')
else:
    print('The input string is valid.')
```

```
The input string is valid.
```

LR(0) Parser

```
# Define grammar and terminals
grammar = {
    ('S', ('(', 'S', ')')),
    ('S', ('i',)),
}

terminals = {'(', ')', 'i', '$'}
```

Define the grammar and terminals. The grammar defines two productions for non-terminal S, which are S -> (S) and S -> i, and the set of terminals includes (,), i, and $.

```
# Define LR(0) automaton
states = [
    {('S', ('.', '(', 'S', ')'))},
    {('S', ('(', '.', 'S', ')')), ('S', ('.', 'i'))},
    {('S', ('(', 'S', '.', ')'))},
]
```

Define the states of the LR(0) automaton. Each state is represented by a set of items. An item is a production with a dot . to indicate the current position in the right-hand side of the production. The states are defined such that the initial state has the item S -> . (S), and the final state has the item S -> (S) .

```
# Define transitions
transitions = {
    (0, '('): 1,
    (1, 'S'): 2,
    (2, ')'): 3,
    (2, ','): 4,
}
```

Define the transitions between the states of the automaton. Each transition is represented by a tuple of the form (current_state, symbol): next_state. The transitions are defined such that when the automaton is in state 0 and reads a (, it goes to state 1, and so on.

```
# Define ACTION and GOTO tables
ACTION = {
    0: {'(': 'S1'},
    1: {'(': 'S1', 'i': 'S2'},
    2: {')': 'S3', '$': 'accept'},
}

GOTO = {
    0: {'S': 1},
    1: {'S': 2},
}
```

Define the ACTION and GOTO tables for the LR(0) parser. The ACTION table defines the actions to take when the automaton is in a certain state and reads a certain symbol. The GOTO table defines the next state to transition to when the automaton is in a certain state and reduces a certain non-terminal. The tables are defined such that when the automaton is in state 0 and reads a (, it shifts to state 1 and pushes symbol (to the stack, and so on.

```
# Define input string
input_str = ['i', '(', 'i', ',', 'i', ')', '$']
```

```
# Initialize parser
stack = [0]
input_index = 0
```

Initialize the parser by setting the initial state of the automaton to 0 and the input index to 0. The stack is initially empty except for the initial state.

```
# Define input string
input_str = ['i', '(', 'i', ',', 'i', ')', '$']

# Initialize parser
stack = [0]
input_index = 0

# Parse input string
while True:
    state = stack[-1]
    symbol = input_str[input_index]
    action_value = ACTION[state].get(symbol)

    # Shift
    if action_value and action_value[0] == 'S':
        next_state = int(action_value[1:])
        stack.append(symbol)
```

```
        stack.append(next_state)
        input_index += 1

    # Reduce
    elif action_value and action_value[0] == 'R':
        production = grammar[int(action_value[1:])]
        lhs, rhs = production
        for _ in range(len(rhs)):
            stack.pop()
            stack.pop()
        next_state = GOTO[stack[-1]].get(lhs)
        stack.append(lhs)
        stack.append(next_state)

    # Error
    elif action_value == 'error':
        print('Parsing error: unable to match input symbol with stack symbol.')
        break

     # Accept
    else:
        print('Valid')
        break
```

1. Here, we define the input string, initialize the parser, and parse the input string.
2. We first initialize the parser by setting the stack to contain the start state (0), and setting the input index to 0.
3. We then enter a while loop that continues until we either reach the end of the input (accept), or encounter an error.
4. In each iteration of the while loop, we retrieve the current state and input symbol, and use them to look up the appropriate action in the ACTION table.
5. If the action is a shift (indicated by 'S' followed by a number), we add the input symbol and next state to the stack, and increment the input index.
6. If the action is a reduce (indicated by 'R' followed by a number), we pop off 2 * len(rhs) elements from the stack (since each symbol on the right-hand side of the production corresponds to both a symbol and a state), and push the non-terminal symbol and next state onto the stack.
7. If the action is 'error', we print an error message and break out of the while loop.
8. If the action is accept or anything else, we print an success message and break out of the while loop.

```
# Define grammar and terminals
grammar = {
    ('S', ('(', 'S', ')')),
    ('S', ('i',)),
}

terminals = {'(', ')', 'i', '$'}

# Define LR(0) automaton
states = [
    {('S', ('.', '(', 'S', ')'))},
    {('S', ('(', '.', 'S', ')')), ('S', ('.', 'i'))},
    {('S', ('(', 'S', '.', ')'))},
]

# Define transitions
transitions = {
    (0, '('): 1,
    (1, 'S'): 2,
    (2, ')'): 3,
    (2, ','): 4,
}

# Define ACTION and GOTO tables
ACTION = {
    0: {'(': 'S1'},
    1: {'(': 'S1', 'i': 'S2'},
    2: {')': 'S3', '$': 'accept'},
}

GOTO = {
    0: {'S': 1},
    1: {'S': 2},
}
```

```

# Define input string
input_str = ['i', '(', 'i', ',', 'i', ')', '$']

# Initialize parser
stack = [0]
input_index = 0

# Parse input string
while True:
    state = stack[-1]
    symbol = input_str[input_index]
    action_value = ACTION[state].get(symbol)

    # Shift
    if action_value and action_value[0] == 'S':
        next_state = int(action_value[1:])
        stack.append(symbol)
        stack.append(next_state)
        input_index += 1

    # Reduce
    elif action_value and action_value[0] == 'R':
        production = grammar[int(action_value[1:])]
        lhs, rhs = production
        for _ in range(len(rhs)):
            stack.pop()
            stack.pop()
        next_state = GOTO[stack[-1]].get(lhs)
        stack.append(lhs)
        stack.append(next_state)

    # error
    elif action_value == 'error':
        print('Parsing error: unable to match input symbol with stack symbol.')
        break

     # accept
    else:
        print('Valid')
        break

```

```
Valid
```

✓ 0s completed at 3:27 AM