Parsa Aghaali

400521072

Report of Hw4

In this exercise, we are asked to write a program that takes a grammar and converts it into LL(1) grammar with the help of ChatGPT. The code is implemented by ChatGPT 3.5.

First, we should know that the definition of LL(1) grammar is as follows:

LL(1) Grammars

LL(1) GRAMMARS AND LANGUAGES. A context-free grammar $G = (V_T, V_N, S, P)$ whose parsing table has no multiple entries is said to be *LL*(1). In the name *LL*(1),

- the first *L* stands for scanning the input from **l**eft to right,
- the second *L* stands for producing a **l**eftmost derivation,
- and the 1 stands for using **one** input symbol of lookahead at each step to make parsing action decision.

A language is said to be *LL*(1) if it can be generated by a *LL*(1) grmmar. It can be shown that *LL*(1) grammars are

- not ambiguous and
- not left-recursive.

Moreover we have the following theorem to characterize *LL*(1) grammars and show their importance in practice.

Theorem 3 A context-free grammar $G = (V_T, V_N, S, P)$ is *LL*(1) if and if only if for every nonterminal A and every strings of symbols α, β such that $\alpha \neq \beta$ and $A \longmapsto \alpha \mid \beta$ we have

1. FIRST(α) $\cap$ FIRST(β) = $\emptyset$,
2. if $\alpha \overset{*}{\Longrightarrow} \varepsilon$ then FIRST(β) $\cap$ FOLLOW(A) = $\emptyset$.

Figure 1 from https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Syntax.html/node14.html

Explanation of the code

This function eliminate_left_recursion(rules) is designed to remove left recursion from a given set of grammar rules.

```
    new_rules = {}
```

This initializes an empty dictionary **new_rules**, which will hold the modified grammar rules without left recursion.

```
    for lhs, rhs_list in rules.items():
```

This loop iterates over each non-terminal symbol (**lhs**) and its corresponding list of productions (**rhs_list**) in the input grammar rules dictionary.

```
        direct_recursion = []
        non_recursion = []
```

These lines initialize two lists: **direct_recursion** to store productions with direct left recursion, and **non_recursion** to store productions without direct left recursion.

```
        for rhs in rhs_list:
```

This loop iterates over each production (**rhs**) in the list of productions for the current non-terminal symbol.

```
            if rhs.startswith(lhs):
                direct_recursion.append(rhs[len(lhs):].strip())
```

Here, it checks if the production starts with the same symbol as the left-hand side (**lhs**). If it does, it indicates direct left recursion. In that case, it appends the part of the production after the left-hand side symbol (excluding the left-hand side itself) to the **direct_recursion** list.

```
            else:
                non_recursion.append(rhs)
```

If the production does not start with the same symbol as the left-hand side, it means there is no direct left recursion, so the production is added to the **non_recursion** list.

```
        if direct_recursion:
            new_lhs = lhs + "'"
```

If there are productions with direct left recursion, a new non-terminal symbol is created by appending a prime (') to the original left-hand side symbol.

```
            new_rules[lhs] = [rhs + ' ' + new_lhs for rhs in non_recursion] if
non_recursion else ['ε']
```

The original non-recursive productions are modified by appending the new non-terminal symbol, and if there are no non-recursive productions, it's replaced with an epsilon ('ε') production.

```

            new_rules[new_lhs] = [(rhs + ' ' + new_lhs).strip() for rhs in
direct_recursion] + ['ε']
```

For the new non-terminal symbol representing direct recursion, the original direct recursive productions are modified by appending the new non-terminal symbol, and an epsilon production is added.

```
        else:
            new_rules[lhs] = rhs_list
```

If there is no direct recursion, the original productions remain unchanged.

```
      return new_rules
```

Finally, the function returns the modified grammar rules with left recursion eliminated.

The function **left_factor(rules)**, aims to perform left factoring on a given set of grammar rules.

```
    new_rules = {}
```
This initializes an empty dictionary **new_rules**, which will hold the modified grammar rules after left factoring.

```
    for lhs, alternatives in rules.items():
```
This loop iterates over each non-terminal symbol (**lhs**) and its corresponding list of alternative productions (**alternatives**) in the input grammar rules dictionary.

```
        if len(alternatives) > 1:
```
This condition checks if there is more than one alternative production for the current non-terminal symbol.

```
            common_prefix = None
```
It initializes a variable **common_prefix** to store the common prefix shared among the alternative productions. It will be used to factor out common prefixes.

```
            for alt in alternatives:
                parts = alt.split()
```
This loop iterates over each alternative production (**alt**) for the current non-terminal symbol and splits it into its constituent parts.

```
                if common_prefix is None:
                    common_prefix = parts[0]
```
If **common_prefix** is not yet initialized, it sets it to the first part of the first alternative production.

```
                elif parts[0] != common_prefix:
                    common_prefix = None
                    break
```
If **common_prefix** is already set and if the first part of any alternative production does not match the **common_prefix**, it sets **common_prefix** to **None** and breaks out of the loop. This indicates that there is no common prefix among all alternative productions.

```
            if common_prefix:
```
If there is a common prefix among all alternative productions:

```
                new_lhs = lhs + "'"
```

A new non-terminal symbol is created by appending a prime (') to the original left-hand side symbol.

```
                new_rules[lhs] = [common_prefix + ' ' + new_lhs]
```

The original non-terminal symbol is updated to produce the common prefix followed by the new non-terminal symbol.

```
                new_rules[new_lhs] = [alt[len(common_prefix):].strip() or 'ε' for
alt in alternatives]
```

The new non-terminal symbol is updated to produce the remaining parts of the alternative productions after removing the common prefix. If there is nothing remaining after removing the common prefix, it is replaced with an epsilon ('ε') production.

```
            else:
                new_rules[lhs] = alternatives
```

If there is no common prefix among all alternative productions, the original non-terminal symbol remains unchanged.

```
        else:
            new_rules[lhs] = alternatives
```

If there is only one alternative production for the current non-terminal symbol, it remains unchanged.

```
    return new_rules
```

Finally, the function returns the modified grammar rules with left factoring applied.

This function, **find_nullable_nonterminals(rules)**, is responsible for identifying nullable non-terminal symbols within a given set of grammar rules.

```
    nullable = set()
```

This line initializes an empty set called **nullable**, which will store the non-terminal symbols that are nullable, i.e., can derive the empty string ('ε').

```
    changes = True
```

A boolean variable **changes** is initialized to **True**, indicating that there might be changes in the nullable set in the upcoming iterations.

```
    while changes:
```

This initiates a loop that continues as long as there are changes detected in the **nullable** set.

```
        changes = False
```

At the beginning of each iteration, **changes** is set to **False**. If any new nullable symbols are found during the iteration, it will be set back to **True**, indicating that another iteration is required to ensure all nullable symbols are identified.

```
        for lhs, rhs_list in rules.items():
```

This loop iterates over each non-terminal symbol (**lhs**) and its corresponding list of right-hand side productions (**rhs_list**) in the input grammar rules dictionary.

```
            for rhs in rhs_list:
```

Within the loop for each non-terminal symbol, another loop iterates over each production (**rhs**) in its list of right-hand side productions.

```
                parts = rhs.split()
```

Each production is split into its constituent parts.

```
                if all(part in nullable or part == 'ε' for part in parts):
```

This condition checks if all parts of the current production (**rhs**) are either already present in the **nullable** set or if they are equal to 'ε' (the empty string). If this condition is true, it means that the entire production can derive 'ε', and thus the left-hand side non-terminal (**lhs**) is nullable.

```
                    if lhs not in nullable:
                        nullable.add(lhs)
                        changes = True
```

If the current left-hand side non-terminal is not already in the **nullable** set, it is added, and **changes** is set to **True**, indicating that a change has been made and another iteration might be needed.

```
    return nullable
```

Finally, the function returns the set of nullable non-terminal symbols after all iterations are complete.

This function, **remove_null_productions(rules, nullable)**, is designed to remove null (epsilon) productions from a given set of grammar rules, considering the nullable non-terminals that have been previously identified.

```
    new_rules = {}
```

This line initializes an empty dictionary **new_rules**, which will store the modified grammar rules after removing null productions.

```
    for lhs, rhs_list in rules.items():
```

This loop iterates over each non-terminal symbol (**lhs**) and its corresponding list of right-hand side productions (**rhs_list**) in the input grammar rules dictionary.

```
        new_set = set(rhs_list)
```

A new set **new_set** is initialized with the original right-hand side productions of the current non-terminal symbol. Using a set here helps to automatically remove duplicates.

```
        if lhs in nullable:
            new_set.add('ε')
```

If the current non-terminal symbol (**lhs**) is nullable (i.e., it can derive the empty string), then 'ε' (the empty string) is added to the set of right-hand side productions.

```
        for rhs in rhs_list:
```

This loop iterates over each production (**rhs**) in the list of right-hand side productions for the current non-terminal symbol.

```
            parts = rhs.split()
```

Each production is split into its constituent parts.

```
            for i in range(len(parts)):
                if parts[i] in nullable:
                    new_rhs = parts[:i] + parts[i+1:]
                    if new_rhs:
                        new_set.add(' '.join(new_rhs).strip())
                    else:
                        new_set.add('ε')
```

For each part in the production, if it is nullable (i.e., it can derive the empty string), a new production is generated by removing that part. If the resulting production is not empty, it is added to the set of right-hand side productions. If the resulting production is empty, 'ε' (the empty string) is added to the set.

```
        new_rules[lhs] = list(new_set)
```

After processing all productions for the current non-terminal symbol, the set of modified right-hand side productions is converted back to a list and assigned to the current non-terminal symbol in the **new_rules** dictionary.

```
    return new_rules
```

Finally, the function returns the modified grammar rules with null productions removed.

This function, **convert_to_ll1(grammar)**, is responsible for converting a given context-free grammar into an LL(1) grammar by applying several transformations.

```
    nullable = find_nullable_nonterminals(grammar)
```

This line invokes the **find_nullable_nonterminals** function on the input grammar to identify nullable non-terminal symbols, which are symbols that can derive the empty string ('ε'). The result is stored in the **nullable** set.

```
    grammar = remove_null_productions(grammar, nullable)
```

Here, the **remove_null_productions** function is called to remove null (epsilon) productions from the grammar, considering the nullable non-terminals identified earlier. The modified grammar without null productions is then stored back into the **grammar** variable.

```
    grammar = eliminate_left_recursion(grammar)
```

This line calls the **eliminate_left_recursion** function to remove left recursion from the grammar. Left recursion is a situation where a non-terminal symbol directly produces a sequence that starts with itself. The modified grammar without left recursion is then stored back into the **grammar** variable.

```
    grammar = left_factor(grammar)
```

Here, the **left_factor** function is called to perform left factoring on the grammar. Left factoring is a process to remove common prefixes from the alternative productions of a non-terminal symbol. The modified grammar after left factoring is then stored back into the **grammar** variable.

```
    return grammar
```

Finally, the function returns the modified grammar, which should now be in LL(1) form. LL(1) grammars are a subset of context-free grammars that are suitable for parsing using a predictive parsing algorithm without backtracking. The LL(1) property means that for any pair of distinct productions of the same non-terminal, there is a unique terminal symbol that can be used to predict which production to use during parsing.

```
grammar = {
    "A": ["E"],
  "E": ["E + T" , "E - T" , "T" ],
  "T": ["T * F" , "T / F" , "F"],
```

```
  "F": ["Id" , "No" , "( E )"]
}
```

This grammar defines a simple arithmetic expression language with addition, subtraction, multiplication, division, and parentheses. The non-terminal symbols are "A", "E", "T", and "F", representing the starting symbol, expressions, terms, and factors, respectively. Terminals include "Id" (identifiers), "No" (numbers), and operators.

After applying the **convert_to_ll1** function, the LL(1) grammar is obtained. Here's the output of the LL(1) grammar:

A -> E

E -> T E'

E' -> + T E' | - T E' | ε

T -> F T'

T' -> * F T' | / F T' | ε

F -> (E) | Id | No

1. **A -> E**: This rule states that the starting symbol "A" can derive an expression "E".
2. **E -> T E'**: Expressions can start with a term "T" followed by the non-terminal "E'".
3. **E' -> + T E' | - T E' | ε**: This production handles the addition and subtraction operations. It allows expressions to continue with an optional "+" or "-" symbol followed by a term "T" and another "E'". The epsilon ('ε') represents an empty string, indicating that there can be no further expansion after an expression ends.
4. **T -> F T'**: Terms can start with a factor "F" followed by the non-terminal "T'".
5. **T' -> * F T' | / F T' | ε**: This production handles the multiplication and division operations. It allows terms to continue with an optional "*" or "/" symbol followed by a factor "F" and another "T'". The epsilon ('ε') represents an empty string, indicating that there can be no further expansion after a term ends.
6. **F -> Id | No | (E)**: Factors can be identifiers "Id", numbers "No", or expressions within parentheses "(E)".

Another example:

```
grammar = {
    "E": ["T", "E + T"],
```

```
    "F": ["( E )", "id"],
    "T": ["F", "T * F"]
}
```

This grammar defines a simple language with arithmetic expressions involving parentheses, addition, multiplication, and identifiers. The non-terminal symbols are "E", "F", and "T", representing expressions, factors, and terms, respectively. Terminals include "id" (identifiers) and parentheses.

After applying the **convert_to_ll1** function, the LL(1) grammar is obtained. Here's the output of the LL(1) grammar:

E -> T E': This rule states that an expression "E" can start with a term "T" followed by the non-terminal "E'".

2. **E' -> + T E' | ε**: This production handles addition operations. It allows expressions to continue with an optional "+" symbol followed by a term "T" and another "E'". The epsilon ('ε') represents an empty string, indicating that there can be no further expansion after an expression ends.
3. **T -> F T'**: This rule states that a term "T" can start with a factor "F" followed by the non-terminal "T'".
4. **T' -> * F T' | ε**: This production handles multiplication operations. It allows terms to continue with an optional "*" symbol followed by a factor "F" and another "T'". The epsilon ('ε') represents an empty string, indicating that there can be no further expansion after a term ends.
5. **F -> (E) | id**: This rule states that a factor "F" can be either an expression within parentheses "(E)" or an identifier "id".