Report

Earley parser implementation

Jean Cardoso , Mazouz Abderahim

`init(g,w):`

```python
# Creation and initialisation of the table T for the word w and the grammar gr
def init(g, w):
    # g: Grammar
    # w: word

    T = {}

    for i in range(len(w) + 1):
        T[i] = TableCell() # key: indice, value: TableCell

    for rule in g.rules:
        if (rule.lhs.name == "S"):
            item = Item(0, rule.lhs, [], rule.rhs) # rule lhs has to be the axiom
            T[0].cAppend(item, "init")

    return T
```

1. **Dictionary Initialization (`T = {}`):**
   - `T` is a dictionary that will represent the parsing table. Each key in the dictionary corresponds to a position in the input word `w`, and the value associated with each key is a `TableCell`.
2. **Table Initialization (`for i in range(len(w) + 1):`):**
   - The code uses a loop to initialize each position in the table with an empty `TableCell`. The loop runs from 0 to the length of the input word `w`, ensuring that there is a table entry for each position.
3. **Rule Initialization (`for rule in g.rules:`):**
   - The code then iterates over the rules in the grammar `g`. For each rule, it checks if the left-hand side (`lhs`) of the rule is the start symbol "S" (assumed to be the axiom).
   - If the left-hand side is "S," it creates an `Item` representing the rule. The `Item` includes information such as the rule's position (0 in this case), the left-hand side symbol, an empty list for the right-hand side, and the full right-hand side of the rule.
   - It then appends this `Item` to the `TableCell` at position 0 using a method named `cAppend`.

`pred(g,it,T,j)`

```python
# Insert in the table any new items resulting from the pred operation for the item it
def pred(g, it, T, j):
    # g: Grammar
    # it: Item
    # T: table
    # j : index

    # fs_ad: Symbol (first symbol after the dot, it is a non-terminal symbol)
    fs_ad = it.ad[0]
    # iterate over rules
    for r in g.rules:
        # if the first symbol after the dot is on the left hand side of the rule r
        if r.lhs.name == fs_ad.name:
            # add an item with the rule r to the cell
            T[j].cAppend( Item (j, r.lhs, [], r.rhs), "pred" )
```

1. **Parameters:**
    - `g`: The grammar.
    - `it`: The current item for which the "pred" operation is performed.
    - `T`: The parsing table.
    - `j`: The index representing the current position in the parsing table.
2. **Extracting the First Symbol after the Dot (`fs_ad = it.ad[0]`):**
    - `it.ad` represents the symbols after the dot in the current item `it`. The code extracts the first symbol after the dot and assigns it to the variable `fs_ad`.
3. **Rule Iteration (`for r in g.rules:`):**
    - The code then iterates over the rules in the grammar `g`.
4. **Checking the First Symbol Condition (`if r.lhs.name == fs_ad.name:`):**
    - For each rule `r`, it checks if the left-hand side (`lhs`) of the rule has the same name as the first symbol after the dot (`fs_ad`). This essentially checks if the next symbol expected after the dot of the current item matches the non-terminal symbol on the left-hand side of the rule.
5. **Adding New Items (`T[j].cAppend(Item(j, r.lhs, [], r.rhs), "pred")`):**
    - If the condition is met, a new item is created using the rule `r`. The new item is created with the same position `j`, the left-hand side symbol `r.lhs`, an empty list for the symbols before the dot (represented by `[]`), and the right-hand side symbols `r.rhs`.
    - This new item is then appended to the parsing table `T` at position `j` using a method named `cAppend`. The operation is labeled as "pred" to indicate that it resulted from the "pred" operation.

`scan(it,T,j)`

```python
# Insert in the table any new items resulting from the scan operation for the item it
def scan(it,T,j):
    # it: Item
    # T: table
    # j: index

    # copy the item to avoid modifying the original item
    # it2add: Item
    it2add = it

    # fs_ad: Symbol (store the first symbol after the dot)
    fs_ad = it2add.ad[0]
    it2add.bd.append(fs_ad)
    it2add.ad = it2add.ad[1:]

    # add the new item to the chart
    T[j + 1].cAppend(it2add, "scan")
```

1. `it`: This is an object of type `Item` that is passed as a parameter to the `scan` function. It seems to represent an item in a parsing chart.
2. `T`: This parameter is assumed to be a table, and the function modifies this table.
3. `j`: This is an index, and it's used to determine the position in the table where the new item should be added.
4. `it2add`: A copy of the original item (`it`) is created to avoid modifying the original.
5. `fs_ad`: This variable stores the first symbol after the dot in the item (`it`).
6. `it2add.bd.append(fs_ad)`: The first symbol after the dot (`fs_ad`) is appended to the "bd" attribute of the copied item (`it2add`).
7. `it2add.ad = it2add.ad[1:]`: The first symbol after the dot is removed from the "ad" attribute of the copied item (`it2add`).
8. `T[j + 1].cAppend(it2add, "scan")`: The modified item (`it2add`) is appended to the chart (`T`) at the next position (`j + 1`). The "scan" operation is recorded.

`comp(it,T,j)`:

```python
# Insert in the table any possible new items resulting from the comp operation for the item it
def comp(it,T,j):
    # it: Item
    # T: table
    # j: index

    k = 0
    # iterate over items in the cell of the indice of the current item it in T
    # T[it.i].c: list[Item]
    while k < len(T[it.i].c):
        # compItem: Item (item that is being analysed in the comp operation)
        compItem = T[it.i].c[k]

        # if the item expects the current non-terminal symbol (it.lhs) after the dot
        if ( (len(compItem.ad) > 0) and (compItem.ad[0].name == it.lhs.name) ):
            # create a new item by moving the dot to the right
            newItem = Item(compItem.i, compItem.lhs, compItem.bd + [it.lhs], compItem.ad[1:])
            # add the new item to the chart
            T[j].cAppend(newItem, "comp")

        k += 1
```

1. **`it`**: This is an object of type `Item` representing an item in the parsing chart. The completion operation is performed on this item.
2. **`T`**: This parameter is assumed to be a table, and the function modifies this table.
3. **`j`**: This is an index, and it's used to determine the position in the table where the new items should be added.
4. **`k`**: This is a loop counter variable used to iterate over items in the cell of the index of the current item `it` in `T`.
5. **`while k < len(T[it.i].c):`** This loop iterates over items in the cell of the index `it.i` in the table `T`. The items in this cell are part of the parsing chart.
6. **`compItem`**: This is an item being analyzed in the completion operation.
7. **`if (len(compItem.ad) > 0) and (compItem.ad[0].name == it.lhs.name):`** This condition checks if the `compItem` expects the current non-terminal symbol (`it.lhs`) after the dot.
8. **`newItem = Item(compItem.i, compItem.lhs, compItem.bd + [it.lhs], compItem.ad[1:])`**: If the condition is true, a new item (`newItem`) is created by moving the dot to the right. This new item is formed by taking the existing `compItem.bd` and adding `it.lhs` to it, while also updating the remaining symbols after the dot.
9. **`T[j].cAppend(newItem, "comp")`**: The new item (`newItem`) is added to the chart at the position `j`. The "comp" operation is recorded.

`table_complete(g,w,T):`

```python
# Return True if the analysis is successful, otherwise False
def table_complete(g, w, T):
    # g: Grammar
    # w: word
    # T: table

    # final_cell: TableCell
    final_cell = T[len(w)]  # Get the final cell in the parsing table

    # Check if there is an item in the final cell indicating successful parsing
    for item in final_cell.c:
        if item.lhs == g.axiom and item.ad == [] and item.i == 0:
            return True

    return False
```

1. **Parameters:**
    a. `g`: The grammar.
    b. `w`: The word that is being parsed by the program
    c. `T`: The parsing table.
2. `final_cell`: this is a TableCell object and it is the final cell of the parsing table
3. `for item in final_cell.c:` iterate over items in the final cell
4. `if item.lhs == g.axiom and item.ad == [] and item.i == 0:` check if there is an item in the last cell in the form S -> α . with the indice 0, which indicates a sucessful parsing
5. `return False:` return False if the method does not return True at any point

`parse_earley(g,w):`

```
# Parse the word w for the grammar g return the parsing table at the end of the algorithm
def parse_earley(g, w):
    # g: Grammar
    # w: word

    # Initialisation
    T = init(g, w)

    # Top-down analysis
    # iterate over cells in the chart T (j: index of the cell)
    for j in range(len(w) + 1):
        k = 0
        # iterate over items in the j-th cell in the chart T
        # T[i_c].c: list[item]
        while k < len(T[j].c):
            # currentItem: Item (item that is being analysed in the main loop)
            currentItem = T[j].c[k]

            # COMP
            if len(currentItem.ad) == 0:
                comp(currentItem, T, j)

            # PRED
            # check if first symbol after the dot is a non-terminal symbol
            # currentItem.ad[0]: Symbol
            elif g.isNonTerminal(currentItem.ad[0]):
                pred(g, currentItem, T, j)

            # SCAN
            elif j < len(w):  # to make sure to not trigger the scan operation in the last
                              cell
                # We know the first symbol after the dot is a terminal symbol since it is
                    the last option left
                # (either nothing after dot, or a non-terminal or a terminal symbol)
                # Check if it corresponds to the character of the word to parse w at index j
                if (w[j] == currentItem.ad[0].name):
                    scan(currentItem, T, j)

            k += 1

    if table_complete(g, w, T):
        print("Success")
    else:
        print("Failed parsing\n")

    return T
```

This part of the code is the main method of the program. It corresponds to the Earley algorithm seen in class implemented in Python.

1. **`T = init(g,w):`** Initialise the table
2. **`for j in range(len(w) + 1):`** The method iterates over the cells in T created in advance, according to the lenght over the word w, in the init function, with the index j

3. `while k < len(T[j].c):` The method then iterates over the items in the cell. The number of items is not known in advance since items are likely going to be added to the cell, hence the while loop
4. block `if len(currentItem.ad) == 0:` if there is no symbol after the dot, do the comp operation
5. block `elif g.isNonTerminal(currentItem.ad[0]):` if the first symbol after the dot is a non-terminal symbol, do the pred operation
6. block `elif j < len(w):` if none of the other operations were made, we do the scan operation, and we only do it if the method is not currently analysing items in the last cell of T
7. print Sucess if the table complete method returns True. Else, print Failed Parsing