Parser

Goian Tudor – 933/2

https://github.com/VaruTudor/Formal-Languages-and-Compiler-Design

The Grammar has 5 fields:
N – non-terminals
E – terminals
P – productions
S – start symbol
isCFG – flag which checks is the grammar is context-free
For the production we use a dictionary, where for each key (symbol) there is a list of tuples (symbol, destination). Most methods are for file parsing except checkCFG(rules, N) which receives the rules and the set of non-terminals and getProductionsFor(nonTerminal) which receives a non-terminal symbol. Bellow there are some test examples.

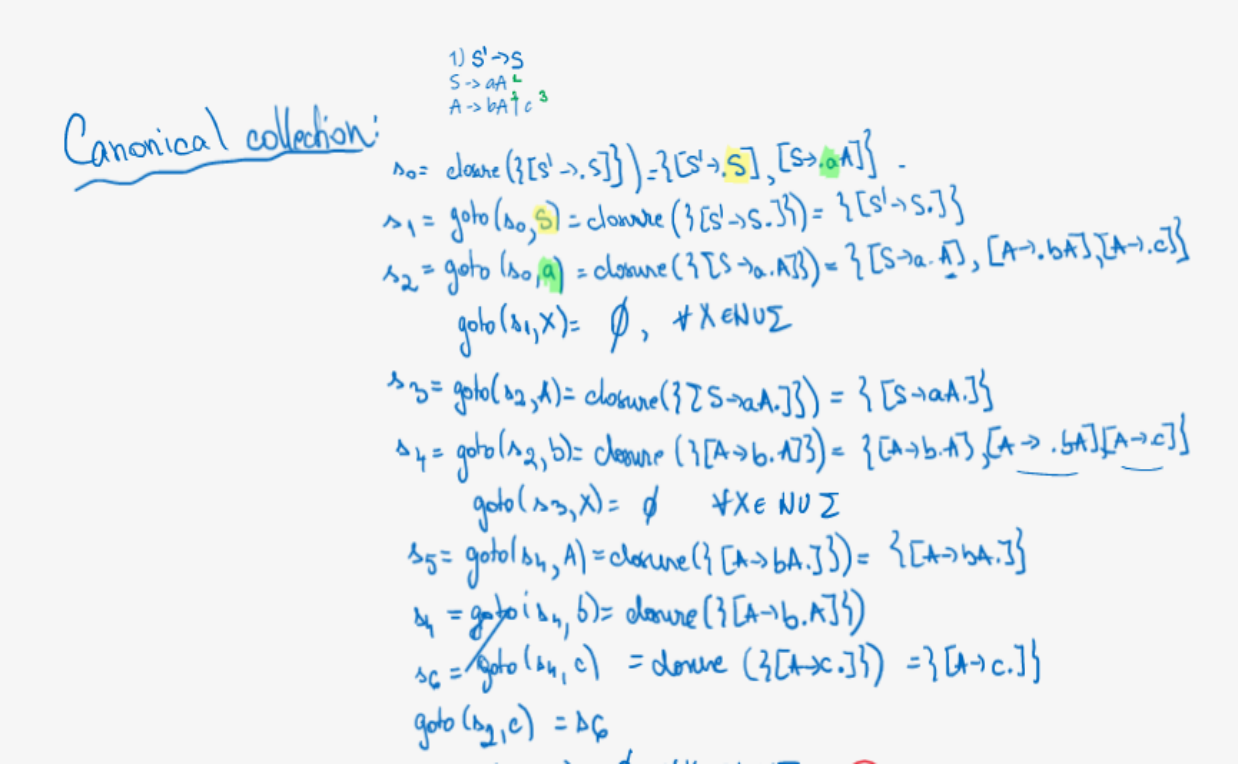
```
N = { S, A, B, C, D }
E = { (, ), +, *, int }
S = S
P = {
    S -> A B C,
    A -> ( S ),
    B -> * S | E,
    C -> + A | C
}
```

```
N = { S, A, B, C, D }
E = { (, ), +, *, int }
productions
[[('A B C', 1)], [('( S )', 2)], [('* S', 3), ('E', 4)], [('+ A', 5), ('C', 6)], None]
productions for A
[('( S )', 2)]
CFG
```

```
closure(itemList):
    For each item in the item list, if after dot there is a non-terminal, add
it to the list of items and repeat the process.
    pseudocode:
    repeat
        for any [A -> α.Bβ] in C do
            for any B -> γ in P do
                if [B -> .γ] 2/ C then
                    C = C U [B -> .γ]
                end if
            end for
        end for
    until C stops changing
    input:
        itemList: a list of LR(0) items
    :return:
        the resulting state



goto(state, symbol):
    For each LR0 item in the state move the dot if the symbol follows it.
Perform closure on the modified item.
    pseudocode:
        goto(s, X) = closure({[A → αX.β]|[A → α.Xβ] ∈ s})
    input:
        state: the state to be checked
        symbol: the symbol to be checked
    output:
        closure of the updated items



computeCanonicalCollection():
    For each state s in the canonical collection, for each symbol X (in both
terminals and non-terminals), check if goto(s,X) result is not an empty list
nor exists already in the canonical collection and if so add it to the
canonical collection
    pseudocode:
    repeat
        for any s in C do
            for any X in N U ß do
                if goto(s,X) != Ø and goto(s,X) not in C then
                    C = C U goto(s,X)
                end if
            end for
        end for
    until C stops changing



computeTableActions():
    For each state in the canonical collection add in the LR0 Table it's
appropriate action
```

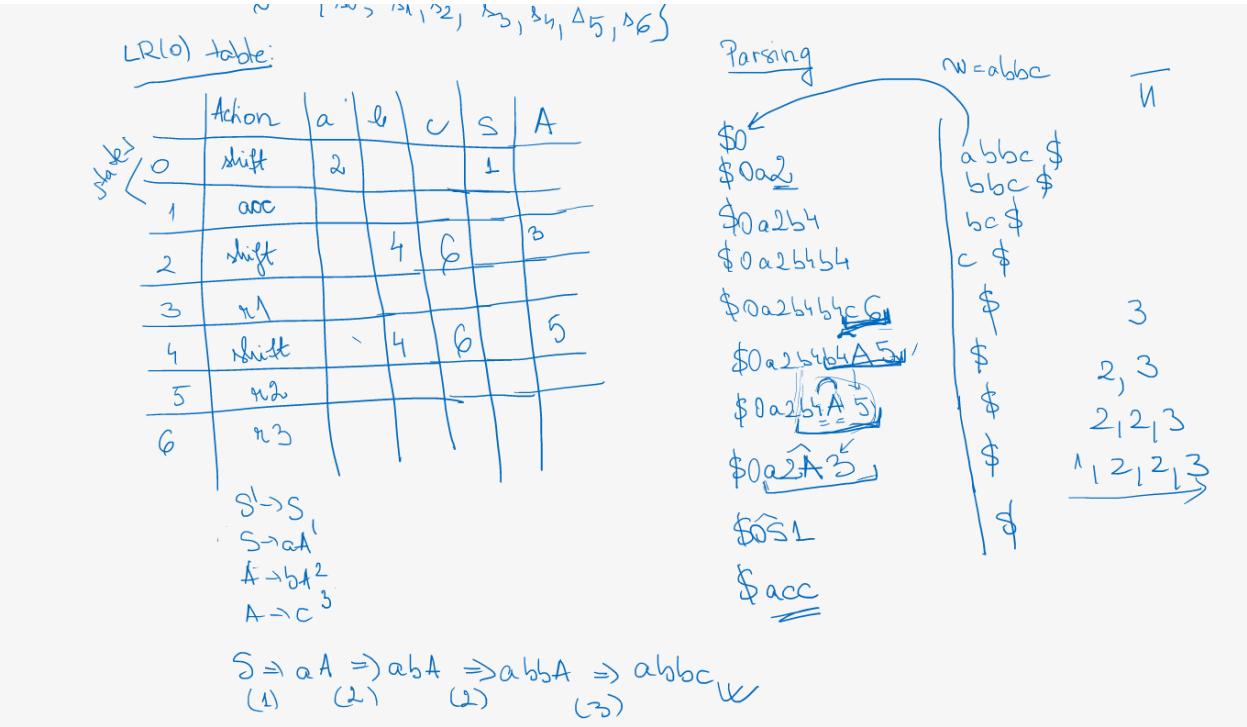
```
def buildInputStack(sequence):
    Iterate and put each symbol found in the input list.
    input:
        sequence: a string containing symbols among it
    output:
        a list of symbols



def parseSequence(sequence):
    Performs the parsing algorithm using 3 stacks (input, working and output) handling each type of
action for a state: shift, reduce or accept. The workingStack is a list considered a stack -> (meaning the
top of the stack is the right most element), and the inputStack is also a list considered a stack <-
(meaning the first element is the top of the stack)
    input:
        a string of symbols to be parsed
    output:
        outputStack
```

Bellow there is a test example computing the Canonical Collection for a grammar.




```
------------------------------------------------------------
Non-terminals and Terminals:
N = { S, A }
E = { a, b, c }
------------------------------------------------------------
Productions:
{'S': [('aA', 1)], 'A': [('bA', 2), ('c', 3)]}
------------------------------------------------------------
Productions for A:
[('bA', 2), ('c', 3)]
------------------------------------------------------------
The grammar is CFG
------------------------------------------------------------
Canonical Collection:
S0( [S` -> .S, S -> .aA] )
S1( [S` -> S.] )
S2( [S -> a.A, A -> .bA, A -> .c] )
S3( [S -> aA.] )
S4( [A -> b.A, A -> .bA, A -> .c] )
S5( [A -> c.] )
S6( [A -> bA.] )
------------------------------------------------------------
```

Bellow there is a test example for parsing a sequence




```
Parsing sequence: abbc
[0]                                   | ['a', 'b', 'b', 'c']          | []
[0, 'a', 2]                           | ['b', 'b', 'c']               | []
[0, 'a', 2, 'b', 4]                   | ['b', 'c']                    | []
[0, 'a', 2, 'b', 4, 'b', 4]           | ['c']                         | []
[0, 'a', 2, 'b', 4, 'b', 4, 'c', 5]   | []                            | []
[0, 'a', 2, 'b', 4, 'b', 4, 'A', 6]   | []                            | ['3']
[0, 'a', 2, 'b', 4, 'A', 6]           | []                            | ['2', '3']
[0, 'a', 2, 'A', 3]                   | []                            | ['2', '2', '3']
[0, 'S', 1]                           | []                            | ['1', '2', '2', '3']

Final output: ['1', '2', '2', '3']
```