Documentation Recursive Descendant Parser

Configuration Class:

```
public class Configuration {
    private String state;
    private Integer index;
    private final Stack<Production> workStack;
    private final Stack<String> inputStack;
}
```

- with getters setters and constructor;

We have all the functions that change the state in different methods:

- EXPAND :
  - WHEN: head of input stack is a nonterminal

```
private void EXPAND(Configuration configuration, MyGrammar grammar) {
    String head = configuration.getInputStack().get(0);
    List<String> firstProduction = grammar.getProductionsByKey(head).get(0);

    configuration.getWorkStack().push(new Production(head, firstProduction));
    configuration.getInputStack().remove( index: 0);
    configuration.getInputStack().addAll( index: 0, firstProduction);
}
```

- ADVANCE :
  - WHEN: head of input stack is a terminal = current symbol from input

```
private void ADVANCE(Configuration configuration) {
    configuration.increaseIndex();
    String head = configuration.getInputStack().remove( index: 0);
    configuration.getWorkStack().push(new Production(head, List.of(head)));
}
```

- MOMENTARY INSUCCESS :
  - WHEN: head of input stack is a terminal ≠ current symbol from input

```
private void MOMENTARY_INSUCCESS(Configuration configuration) {
    configuration.setState(BACK);
}
```

- BACK :
  - WHEN: head of working stack is a terminal

```
private void BACK(Configuration configuration) {
    configuration.decreaseIndex();
    Production lastProduction = configuration.getWorkStack().pop();
    configuration.getInputStack().add( index: 0, lastProduction.getKey());
}
```

- ANOTHER TRY :
  - WHEN: head of working stack is a nonterminal

```
private void ANOTHER_TRY(Configuration configuration, MyGrammar grammar) {
    Production lastProduction = configuration.getWorkStack().peek(); // Aj
    List<List<String>> productions = grammar.getProductionsByKey(lastProduction.getKey());
    List<String> nextProductionValue = getNextProduction(lastProduction, productions);

    if (nextProductionValue != null) {
        configuration.setState(NORMAL);
        configuration.getWorkStack().pop();
        configuration.getWorkStack().push(new Production(lastProduction.getKey(), nextProductionValue));

        lastProduction.getValue().forEach(value -> configuration.getInputStack().remove( index: 0) );
        configuration.getInputStack().addAll( index: 0, nextProductionValue);

    } else {
        if (configuration.getIndex() == 0 && Objects.equals(lastProduction.getKey(), grammar.getStartSymbol())) {
            configuration.setState(ERROR);
            configuration.getWorkStack().pop();
            lastProduction.getValue().forEach(value -> configuration.getInputStack().remove( index: 0) );
        } else {
            configuration.getWorkStack().pop();
            lastProduction.getValue().forEach(value -> configuration.getInputStack().remove( index: 0) );
            configuration.getInputStack().add( index: 0, lastProduction.getKey());
        }
    }
}
```

- SUCCESS :
    - WHEN: input stack is empty and the index is equal with the number of words in the input sequence

```
private void SUCCESS(Configuration configuration) {
    configuration.setState(FINAL);
}
```

Here is the logic of the recursive descendant:

```
public void recursiveDescendant(MyGrammar grammar, String input) {

    try {
        FileWriter fileWriter = new FileWriter(Constant.output + "debug");
        BufferedWriter writer = new BufferedWriter(fileWriter);

        Configuration configuration = new Configuration(grammar.getStartSymbol());

        while (!Objects.equals(configuration.getState(), FINAL) &&
                !Objects.equals(configuration.getState(), ERROR)) {
            writer.write("##################\n");
            writer.write("INPUT: " + configuration.getInputStack() + "\n");
            writer.write("WORK: " + configuration.getWorkStack() + "\n");
            writer.write("STATE: " + configuration.getState() + "\n");
            writer.write("INDEX: " + configuration.getIndex() + "\n");

            if (Objects.equals(configuration.getState(), NORMAL)) {
```

```
                if (configuration.getIndex() == input.split(" ").length &&
configuration.getInputStack().isEmpty()) {
                    SUCCESS(configuration);
                } else {
                    // IMPORTANT
                    if (configuration.getInputStack().isEmpty()) {
                        MOMENTARY_INSUCCESS(configuration);
                    }
                    else {
                        String head = configuration.getInputStack().get(0);
                        if (grammar.getNonTerminalSymbols().contains(head)) {
                            EXPAND(configuration, grammar);

                        } else {
                            if (configuration.getIndex() < input.length() && Objects.equals(head,
input.split(" ")[configuration.getIndex()])) {
                                ADVANCE(configuration);

                            } else {
                                MOMENTARY_INSUCCESS(configuration);
                            }
                        }
                    }
                }

            } else if (Objects.equals(configuration.getState(), BACK)) {
                String lastProductionKey = configuration.getWorkStack().peek().getKey();

                if (grammar.getTerminalSymbols().contains(lastProductionKey)) {
                    // BACK
                    BACK(configuration);
                } else {
                    // ANOTHER TRY
                    ANOTHER_TRY(configuration, grammar);
                }
            }
        }
        // printing logic
        writer.close();
    } catch (IOException exception) {
        System.out.println(exception.getMessage());
    }
}
```

Here is the printing logic:

- If the state is error we just print an “Error” message;

- Else the sequence is accepted and we compute, starting with the startingSymbol, the result using the working stack, keeping only the productions in the grammar rules

```
if (Objects.equals(configuration.getState(), ERROR))
  System.out.println("Error");
else {
  System.out.println("Sequence accepted");

  String result = grammar.getStartSymbol();
  System.out.println(result);

  List<Production> finalProductions = configuration.getWorkStack()
    .stream()
    .filter(production -> grammar.getRules().contains(production))
    .collect(Collectors.toList());

  for (var production : finalProductions) {
    result = result.replaceFirst(production.getKey(), String.join(" ", production.getValue()));
    System.out.println(result);
}
```

EXAMPLE:

CODE:
a a a c b c b a a a a c b c

PARSER:
S
a S
a a S b S
a a a S b S b S
a a a c b S b S
a a a c b c b S
a a a c b c b a S
a a a c b c b a a S
a a a c b c b a a a S
a a a c b c b a a a a S b S
a a a c b c b a a a a c b S
a a a c b c b a a a a c b c