Check algorithm of a pattern generation through a phrased structure grammar - an implementation in the Elixir programming language

Luciana da Costa Marques
Polytechnic School of the University of So Paulo
e-mail: luciana.marques@usp.br

1. Introduction

This paper presents an implementation of the algorithm that checks whether a certain pattern can be generated from a recursive Phrase Structure Grammar.

2. Formal Definitions

This section contains the formal definitions of Logic and Mathematics used to develop this work.

2.1. Phrase Structure Grammar

The main definition needed to this project was of what is a Phrase Structure Grammar. There are formal definition in literature, but here there was the need to also take recursion into account.

A simple definition could be as follows:

1) Let the alphabet be = a,b;
2) Let the list of non-terminals be: S,A,B;
3) Let the initial rule to be followed be: S = AB;
4) Let the rule A be A = aA or a;
5) Let the rule B be B = bB or b;

Out of implementation simplicity, there was the choice to only define two rules apart from the main one, and each of this rules is recursive on itself, that is, either of them do not require the application of a different rule. This could be easily done with only a few code changes, though.

3. Algorithm Description

The Algorithm developed consists in taking the grammar defined above as input and check whether an input string can be generated accordingly to that grammar.

It follows these steps:

1) Take the first rule (string), the rules (map), the list of terminals (map set) and the maximum pattern size (integer) permitted as input;
2) For each rule in the First Rule, generate all the possible sub-chains accordingly to that rule and the maximum permitted size, and combine them;
3) Put all the possible combinations in a map set;
4) Check if the input pattern is present in that map set.

4. Implementation

The algorithm implementation consisted mainly in the steps from the previous section and the ones indicated on the project's terms.

The following functions were defined to accomplish this:

4.1. Parse a list recursively

This function was built in order to understand how it is possible to parse a list recursively. It was indicated on the project's program and indeed its logic was very useful.

```
def recursive_parse(list) do
  if (List.first(list) == nil) do
    IO.puts("End of List")
  else
    IO.puts(List.first(list))
    list2 = List.delete_at(list,0)
    recursive_parse(list2)
  end
end
```

4.2. Get possible sub chains given a rule

For this step, there were two defined functions. The function to get the sub chains given a rule is get-subchain(). Out of simplicity, it was decided that it was better to provide the rule's prefix instead of the rule itself and keep checking the rule map. This could be useful, though, if the defined rules referenced different rules out of themselves, and to accomplish this implementation, the code would only need to be modified by adding a few lines to get the new rule prefix.

The prefix added recursively until the longest sub chain size reaches the maximum value is computed on the next function, and it is inputed to get-subchain() in the 'add' field.

```
def get_subchain(max, list, add, rules_map) do
  # if list is empty, add the first subchain as the simple prefix
  if (List.first(list) == nil) do
    list2 = List.flatten([add])
    get_subchain(max,list2,add,rules_map)
  else
    # get last computed subchain
    current = List.last(list)
    # if not empty, get the size of the last computed subchain
    # if size is equal to max, nothing to add anymore
    if (String.length(current) == max) do
      List.flatten(list)
    else
      # if not equal to max, then add the prefix
      element = current<>add
      # if the result is a string of size bigger than max,
      # just return the previous list
      if (String.length(element) > max) do
        List.flatten(list)
      else
        # if not, append the result to the list
        list2 = List.flatten(list++[element])
        # and keep searching
        get_subchain(max,list2,add,rules_map)
      end
    end
  end
end
```

And here the function to get the rule’s prefix. In case for ”A” = ”aA”, for example, the prefix is ”a” and the function keeps appending to list all the possible combinations until a combination’s size reaches the value in max (which is the maximum sub chain size).

```
def get_add_elements(rule, terminals, add) do
  {current,rule2} = String.split_at(rule,1)
  if (MapSet.member?(terminals, current)) do
    add
  else
    add2 = add<>current
    get_add_elements(rule2,terminals,add2)
  end
end
```

4.3. Combine all possible rule subchains

Now, since it is possible to compute all the sub chains accordingly to the starting rule, there is only the need to combine them in the appropriate order.

The main combination function is combine-subchains(), which returns the final combination list given the grammar’s definitions.

```
def combine_subchains(list, first_rule, rules_map, max, terminals) do
  # if end of the first rule, just return the list created
  if (String.first(first_rule) == nil) do
    List.flatten(list)
  else
    # gets the current rule and updates firs_rule
    {terminal, new_first_rule} = String.split_at(first_rule,1)
    rule = rules_map[terminal]

    # gets the rule's prefix
    add = get_add_elements(rule, terminals, "")

    # gets the possible subchains
    rule_chains = get_subchain(max, [], add, rules_map)

    # if no previous computed subchain, just add the rule's subchains
    if (List.first(list) == nil) do
      list2 = List.flatten(list++[rule_chains])
      combine_subchains(list2,new_first_rule,rules_map,max,terminals)
    else
      # combine the rule's subchains with the already computed ones
      list2 = combine_lists(list,rule_chains,max,[])
      combine_subchains(list2,new_first_rule,rules_map,max,terminals)
    end
  end
end
```

This functions also needs an auxiliary function, which combines two given lists in order (that is, if list1 comes first from list2, than the resulting list only has elements starting with list1’s elements and finishing with list2’s elements).

```

def combine_lists(list,rule_chains,max,result) do
  # When list is empty, just return the computed combinations
  if (List.first(list) == nil) do
    List.flatten(result)
  else
    # Get the current element being added
    element = List.first(list)
    # Add the element to all in rule_chains, checking their size is (<= max)
    list2 = List.flatten(Enum.map(rule_chains, fn x -> if(String.length(element<>x) <= max) do element<>x end end))
    list4 = Enum.reject(list2, &is_nil/1)
    # Updates the list that holds the combining elements
    list3 = List.delete_at(list,0)
    # Keep going until 'list' is empty
    combine_lists(list3,rule_chains,max,List.flatten(result++list4))
  end
end
```

5. Tests

Testing is an important part of developing in Elixir. There were 7 tests made.

5.1. Test 1 - Parse a List recursively

This test was mainly done to test the recursive parsing test.

```
 test ”recursive parse” do
  CHAINREC.recursive_parse([1])
 end
```

5.2. Create a MapSet

Some structures are not obvious to use in Elixir, because functional programming has it’s own rules. There were plenty of wasted time with MapSet, because it is not possible to create uppercase named variables, so this test was used to correct the problems regarding this.

```
 test ”crate mapset” do
   terminals = MapSet.new |> MapSet.put(”A”)
   assert MapSet.member?(terminals, ”A”) == true
 end
```

5.3. Test of the rule subchains generation

This test was to check whether the sub chains were being created appropriately.

```
 test ”rule subchains” do
   max = 3
   rules_map = Map.new |> Map.put(”A”,”aA”) |>
   Map.put(”B”,”bB”)
   terminals = MapSet.new |> MapSet.put(”A”) |>
   MapSet.put(”B”)
   terminal = ”A”
   rule = rules_map[terminal]
   add = CHAINREC.get_add_elements(rule,
   terminals, ””)
  CHAINREC.get_subchain(max, [], add, rules_map)
   == [”a”,”aa”,”aaa”]
 end
```

5.4. Check if subchains are being correctly combined

This test was done to check whether the subchains are being correctly combined.

```
 test ”get all subchains” do
   first_rule = ”AB”
   rules_map = Map.new |> Map.put(”A”,”aA”) |>
   Map.put(”B”,”bB”)
   terminals = MapSet.new |> MapSet.put(”A”) |>
   MapSet.put(”B”)
   max = 3
   assert CHAINREC.combine_subchains([],
   first_rule, rules_map, max, terminals) == [”ab
   ”,”abb”,”aab”]
 end
```

5.5. Check if a given pattern can be generated by a certain grammar

This was the main and final test. It checks if an input pattern can be generated by the grammar previously defined.

```
 test ”check chain” do
   first_rule = ”AB”
   rules_map = Map.new |> Map.put(”A”,”aA”) |>
   Map.put(”B”,”bB”)
   terminals = MapSet.new |> MapSet.put(”A”) |>
   MapSet.put(”B”)
   max = 3
   possible_chains_list = CHAINREC.
   combine_subchains([], first_rule, rules_map,
   max, terminals)
   test_chain = ”aab”
   possible_chains_set = possible_chains_list |>
   Enum.uniq |> List.to_tuple
   possible_chains_set = MapSet.new(
   possible_chains_list)
   assert MapSet.member?(possible_chains_set,
   test_chain) == true
 end
```

6. Conclusion

Basically all tests passed.

A possible improvement would be to create an API that takes rules that need to apply other rules apart from themself, like ”A” = ”aB”.