RAPPORT PROJET

UE5 - Comptage et énumération de structures de données

November 24, 2017

Gustavo Castro

université
PARIS-SACLAY

CONTENTS

1 QUESTIONS

1.1 Question 1

Binary Trees - Sizes	0	1	2	3	4	5	6	7	8	9	10
Tree	0	1	1	2	5	14	42	132	429	1430	4862
Node	0	0	1	2	5	14	42	132	429	1430	4862
Leaf	0	1	0	0	0	0	0	0	0	0	0
Fibonnaci words - sizes	0	1	2	3	4	5	6	7	8	9	10
Fib	1	2	3	5	8	13	21	34	55	89	144
Cas1	0	2	3	5	8	13	21	34	55	89	144
Cas2	0	1	1	2	3	5	8	13	21	34	55
Vide	1	0	0	0	0	0	0	0	0	0	0
CasAu	0	1	2	3	5	8	13	21	34	55	89
AtomA	0	1	0	0	0	0	0	0	0	0	0
AtomB	0	1	0	0	0	0	0	0	0	0	0
CasBAu	0	0	1	2	3	5	8	13	21	34	55

1.2 Question 2

"ABAlphabetGram": {"Mot": UnionRule("Vide", "NonVide", inv_union_vide_nonvide, len),

"NonVide": ProductRule("Singleton", "Mot", join_lambda, split_first, len),

"Singleton": UnionRule("AtomA", "AtomB", inv_union_first_a, len),

"AtomA": SingletonRule("A"),

"AtomB": SingletonRule("B"),

"Vide": EpsilonRule("")}

1.3 Question 3

"dyckGram": {"Dyck": UnionRule("Vide", "NonVide", inv_union_vide_nonvide, len_div_2),
"NonVide": ProductRule("DyckPremier", "Dyck", join_lambda, inv_prod_dyck_1, len_div_2),
"DyckPremier": ProductRule("Atom", "Dyck", encaps, inv_encaps_dyck, len_div_2),
"Atom": SingletonRule("()"),
"Vide": EpsilonRule("")}

1.4 Question 4

"not3Gram": {"Not3": UnionRule("Vide", "NonVide", inv_union_vide_nonvide, len),
"NonVide": UnionRule("A_Max1A", "B_Max1B", inv_union_first_a, len),
"A_Max1A": ProductRule("AtomA", "Max1A", join_lambda, split_first, len),
"B_Max1B": ProductRule("AtomB", "Max1B", join_lambda, split_first, len),
"Max1A": UnionRule("A_NoA", "NoA", inv_union_first_a, len),
"Max1B": UnionRule("B_NoB", "NoB", inv_union_first_b, len),
"A_NoA": ProductRule("AtomA", "NoA", join_lambda, split_first, len),
"B_NoB": ProductRule("AtomB", "NoB", join_lambda, split_first, len),
"NoB": UnionRule("Vide", "A_Max1A", inv_union_vide_nonvide, len),
"NoA": UnionRule("Vide", "B_Max1B", inv_union_vide_nonvide, len),
"AtomA": SingletonRule("A"),
"AtomB": SingletonRule("B"),
"Vide": EpsilonRule("")}

1.5 Question 5

"palinABGram": {"PalinAB": UnionRule("Vide", "NonVide", inv_union_vide_nonvide, len),
"NonVide": UnionRule("Singleton", "NonSingleton", inv_union_sing_nonsing, len),

"NonSingleton": ProductRule("Doublet", "PalinAB", encaps, inv_encaps_palin, len),
"Doublet": UnionRule("DoubletA", "DoubletB", inv_union_first_aa, len),
"DoubletA": ProductRule("AtomA", "AtomA", join_lambda, split_first, len),
"DoubletB": ProductRule("AtomB", "AtomB", join_lambda, split_first, len),
"Singleton": UnionRule("AtomA", "AtomB", inv_union_first_a, len),
"AtomA": SingletonRule("A"),
"AtomB": SingletonRule("B"),
"Vide": EpsilonRule("")}
"palinABCGram": {"PalinABC": UnionRule("Vide", "NonVide", inv_union_vide_nonvide, len),
"NonVide": UnionRule("Singleton", "NonSingleton", inv_union_sing_nonsing, len),
"NonSingleton": ProductRule("Doublet", "PalinABC", encaps, inv_encaps_palin, len),
"Doublet": UnionRule("DoubletA", "DoubletBouC", inv_union_first_aa, len),
"DoubletBouC": UnionRule("DoubletB", "DoubletC", inv_union_first_bb, len),
"DoubletA": ProductRule("AtomA", "AtomA", join_lambda, split_first, len),
"DoubletB": ProductRule("AtomB", "AtomB", join_lambda, split_first, len),
"DoubletC": ProductRule("AtomC", "AtomC", join_lambda, split_first, len),
"Singleton": UnionRule("AtomA", "AtomBouC", inv_union_first_a, len),
"AtomBouC": UnionRule("AtomB", "AtomC", inv_union_first_b, len),
"AtomA": SingletonRule("A"),
"AtomB": SingletonRule("B"),
"AtomC": SingletonRule("C"),
"Vide": EpsilonRule("")}

1.6 Question 6

"sameABGram": {"SameAB": UnionRule("Vide", "NonVide", inv_union_vide_nonvide, len),
"NonVide": UnionRule("A_needBu", "B_needAu", inv_union_first_a, len),
"A_needBu": ProductRule("AtomA", "NeedBu", join_lambda, split_first, len),
"B_needAu": ProductRule("AtomB", "NeedAu", join_lambda, split_first, len),
"NeedAu": ProductRule("NeedA", "SameAB", join_lambda, inv_prod_sameAB_need_a, len),

"NeedBu": ProductRule("NeedB", "SameAB", join_lambda, inv_prod_sameAB_need_b, len),
"NeedA": UnionRule("AtomA", "B_Need2A", inv_union_first_a, len),
"NeedB": UnionRule("AtomB", "A_Need2B", inv_union_first_b, len),
"A_Need2B": ProductRule("AtomA", "Need2B", join_lambda, split_first, len),
"B_Need2A": ProductRule("AtomB", "Need2A", join_lambda, split_first, len),
"Need2A": ProductRule("NeedA", "NeedA", join_lambda, inv_prod_sameAB_need_a, len),
"Need2B": ProductRule("NeedB", "NeedB", join_lambda, inv_prod_sameAB_need_b, len),
"AtomA": SingletonRule("A"),
"AtomB": SingletonRule("B"),
"Vide": EpsilonRule("")}

1.7 Question 7

The *is_grammar_correct* function described in the tests.py script verifies that a grammar is correctly constructed.

1.8 Question 8

The valuations gave these values for the treeGram:

$Tree$	$Node$	$Leaf$
1	2	1

And these values for the fiboGram:

Fib	$Cas1$	$Cas2$	$CasAu$	$CasBAu$	$AtomA$	$AtomB$	$Vide$
0	1	1	1	2	1	1	0

1.9 Question 9

The function *init_grammar* is defined in the Rules.py script and the set_grammar method is also defined in the same script.

1.10 Question 10

The count methods for the objects as shown in the project description are in the Rules.py script.

1.11 Question 11

Firstly it is very important to check that the grammar to be tested was correctly built, by checking that each non-terminal used within a definition is actually defined in the grammar.

Furthermore, as said in the project description, an important thing to test is that the length of the result of the list method is the same as the result of the count method.

Finally, another important test is to check that the rank of the unrank of a certain index is equal to this same index.

1.12 Question 12

All the tests are implemented and described in the tests.py script.

1.13 Question 13

The tests are also launched by running the tests.py script.

1.14 Question 14

All these parameters are added in the Rules.py script and their the grammars and their respective inversion functions are described in the all_grammars.py script.

1.15 Question 15

This caching is done by using a Least Recently Used cache function from the *functools32* library as a python decorator.

This function stores the results of calculated function and whenever it needs to add a new value but its memory is full, it eliminates the least recently used value from its memory and adds the new one.

The only difference from this to a usual basic cache implementation, is that we add a limit to what will be stored in memory and we eliminate the least recently used values in order to add new ones and maintain the same memory.

1.16 Question 16

A condensed grammar example is shown in the all_grammars.py script (and also tested within the tests.py script) and the auxiliary classes and the convert_condensed_gram are implemented in the Rules.py script.

1.17 Question 17

The Bound class and constructor are implemented in the Rules.py script and tested within the tests.py script.