FUNDAMENTALS OF ARTIFICIAL INTELLIGENCE - CS161

Assignment 9 - Due 11:59 p.m., Tuesday, March 12

DO NOT CONSULT ANY OUTSIDE REFERENCES FOR THIS PROBLEM SET

Your assignment is to write a simple production system interpreter in Lisp. Your top-level function, called INTERPRET, will take as its first argument an arbitrary Lisp expression representing working memory, called EXP, and as its second argument a list of productions, called PRODS, and return the final working memory expression. Each production will be a two-element list consisting of a left-hand side and a right-hand side. Your productions will in general contain variables on both sides. Variables are identified by appearing in a separate global list called VARIABLES, defined using defconstant. For example, if VARIABLES equals (X Y Z), then whenever any one of these letters appears in a production, it is treated as a variable instead of a constant. A variable can match any well-formed list structure or atom. You may assume that no variable appears more than once in the left-hand side of the same production rule. The left-hand sides of productions can match any subexpression in working memory, including the whole thing. To fire a production, the left-hand side must match some subexpression of working memory, and then that subexpression is replaced with the right-hand side with the variables replaced by what they were bound to by the matching. Conflict resolution may be handled by the order of productions in the production list. In other words, the first production in the list that matches is fired. Your production interpreter should terminate when no further productions can fire. Your production system should be completely general, and not rely on features of particular problems.

Test your interpreter by writing a complete list of productions to perform reduction to clause form for propositional logic. Define these using defconstant, and call them CLAUSEPRODS. In this problem, the input will be a propositional expression in prefix normal form, abbreviating "equal" by "E", "implies" by "I", "and" by "A", "or" by "O", and "not" by "N". For example, the proposition "(s or t)->q" would be represented as (I (O s t) q). The output will be in clause form, with the Ands, Ors, and Nots remaining explicitly in the expression. For example, the output for the above expression might be (A (O (N s) q) (O (N t) q)). Be sure to include productions required to simplify your clause form expressions. Your rules should be able to support "and," "or," and "equal" with either two or three propositions. That is, (O p q), (A p q), (E p q), (O p q r), (A p q r), and (E p q r) are all valid expressions. "not" will always take one proposition, e.g. (N p), and "implies" should always take two propositions, e.g. (I p q). Note that your rules do NOT need to support more than three propositions for any connective. So, your rules do not need to support the expression (O p q r s). Note also that your output need not have ternary connectives in it. For example, if the initial working memory is (O p q r), then any of the following is a valid output: (O p q r), (O p (O q r)), and (O (O p q) r). Rigorously test your program on a broad range of example problems.