Middle East Technical University
Department of Computer Engineering

CENG 140

Spring 2010

Take Home Exam 2

REGULATIONS

Due date: 24 May, 2010 Monday *(Not subject to postpone)*

Submission: Electronically. You will be submitting your program source code through a text file which you will name as `the2.c` by means of the COW system.

Team: There is **no** teaming up. The homework has to be done and turned in individually.

Cheating: Source(s) and Receiver(s) will receive zero and be subject to disciplinary action.

INTRODUCTION

Wang's Algorithm is used to prove theorems in propositional calculus. So, for example, it is possible to start with three promises

$$P \rightarrow Q, Q \rightarrow R, \neg R$$

and prove the proposition $\neg P$.

To begin proving a theorem with Wang's algorithm, all premises are written on the left-hand side of an arrow that we may call the *sequent arrow* ($\Rightarrow$). Note that the sequent arrow is different then then *implication* arrow. The desired conclusion is written to the right of the sequent arrow. Thus we have:

$$P \rightarrow Q, Q \rightarrow R, \neg R \Rightarrow \neg P$$

This string of symbols is is called a sequent. This particular sequent contains four *top-level* formulas; there are three on the left and on on the right (it contains more than four formulas if we count embedded ones such as the formula P in $P \rightarrow Q$.)

Successively, we apply transformations to the sequent that break it down into simpler ones. The general form of a sequent is:

$$F_1, \ldots, F_m \Rightarrow F_{m+1}, \ldots, F_{m+n}$$

where each F_i is a formula. Intuitively, this sequent may be thought of as representing a larger formula,

$$F_1 \wedge \cdots \wedge F_m \rightarrow F_{m+1} \vee \cdots \vee F_{m+n}$$

Here are the transformation (R1 through R5) and termination (R6 and R7) rules:

R1 : If one of the top-level formulas of a sequent has the form $\neg\square$, we may drop the negation and move $\square$ to the other side of the sequent arrow. Here $\square$ is any formula, e.g., $(P \vee \neg Q)$. If the negation is to the left of the sequent arrow, we call the transformation *NOT on the left;* otherwise it is *NOT on the right.*

R2 : If a top-level formula on the left of the arrow has the form $\square \wedge \triangle$, or on the right of the arrow has the form $\square \vee \triangle$, the connective may be replaced by a comma. The two forms of this rule are called *AND on the left* and *OR on the right*, respectively.

R3 : If a top-level formula on the left has the form $\square \vee \triangle$, we may replace the sequent with two new sequents, one having $\square$ substituted for the occurrence of $\square \vee \triangle$, and the other having $\triangle$ substituted. This is called *splitting on the left* or *OR on the left.*

R4 : If the form $\square \wedge \triangle$ occurs on the right, we may also split the sequent as in Rule R3. This is *splitting on the right* or *AND on the right.*

R5 : A formula (at any level) of the form $\square \rightarrow \triangle$ may be replaced by $\neg\square \vee \triangle$, thus eliminating the implication connective.

R6 : A sequent is considered proved if some top-level formula $\Box$ occurs on both the left and right sides of the sequent arrow. Such a sequent is called an *axiom*. No further transformations are needed on this sequent, although there may remain other sequents to be proved. (The original sequent is not proved until all the sequents obtained from it have been proved.)

R7 : A sequent is proved invalid if all formulas in it are individual proposition symbols (i.e., no connectives), and no symbol occurs on both sides of the sequent arrow. If such a sequent is found, the algorithm terminates; the original *conclusion* does not follow logically from the premises.

Wang's algorithm always converges on a solution to the given problem. Every application of a transformation makes some progress either by eliminating a connective and thus shortening a sequent (even though this may create a new sequent as in the case of R3), or by eliminating the connective "$\rightarrow$". The order in which rules are applied has some bearing on the length of a proof or refutation, but not on the outcome itself.

We may now proceed with the proof for our example. We label the sequents generated, starting with S1 for the initial one.

Label	Sequent	Comment
S1:	$P \rightarrow Q,\ Q \rightarrow R,\ \neg R \Rightarrow \neg P$	Initial sequent.
S2:	$\neg P \vee Q,\ \neg Q \vee R,\ \neg R \Rightarrow \neg P$	Two applications of R5.
S3:	$\neg P \vee Q,\ \neg Q \vee R \Rightarrow \neg P,\ R$	Rl.
S4:	$\neg P,\ \neg Q \vee R \Rightarrow \neg P,\ R$	S4 and S5 are obtained from S3 with R3. Note that S4 is an axiom since P appears on both sides of the sequent arrow at the top level.
S5:	$Q,\ \neg Q \vee R \Rightarrow \neg P,\ R$	The other sequent generated by the application of R3.
S6:	$Q,\ \neg Q \Rightarrow \neg P,\ R$	S6 and S7 are obtained from S5 using R3.
S7:	$Q,\ R \Rightarrow \neg P,\ R$	This is an axiom.
S8:	$Q \Rightarrow \neg P,\ R,\ Q$	Obtained from S6 using R1. S8 is an axiom. The original sequent is now proved, since it has successfully been transformed into a set of three axioms with no unproved sequents left over.

PROBLEM

You are expected to write a program that reads a sequent from the input, in a single line. Then applying the Wang's algorithms determines whether it is proved (valid) or disproved (proved invalid), and prints a `T` or `F`, respectively.

Any binary operation, namely $\rightarrow$, $\vee$, $\wedge$ will always be given enclosed in parenthesis. This is not so for the unary prefix operator of negation '$\neg$'.

Here is the BNF representation for the input.

⟨sequent⟩ ::= ⟨lhs⟩ `#` ⟨rhs⟩
⟨lhs⟩ ::= ⟨formula list⟩ | ε
⟨rhs⟩ ::= ⟨formula list⟩ | ε
⟨formula list⟩ ::= ⟨formula⟩ | ⟨formula⟩ `,` ⟨formula list⟩
⟨formula⟩ ::= ⟨letter⟩ | `-` ⟨formula⟩ | `(` ⟨formula⟩ ⟨infixop⟩ ⟨formula⟩ `)`
⟨infixop⟩ ::= `&` | `|` | `>`
⟨letter⟩ ::= `A` | `B` | ... | `Z`

The semantic is:

Symbol	Meaning
-	$\neg$
|	$\vee$
&	$\wedge$
>	$\rightarrow$

Though the input will not contain whitespaces, you are free to make your implementation insensitive to blanks.

The above given example of:
$P \rightarrow Q,\ Q \rightarrow R,\ \neg R \Rightarrow \neg P$
would be input as:

```
(P>Q),(Q>R),-R#-P
```

The expected output for this input would be:

```
T
```

SPECIFICATIONS

- This is a string manipulation task. So, you are expected to do dynamic allocations for the modified strings (of course free the unused as well).
- The maximum length of an input is 200 characters.
- There will be no erroneous input.
- `struct` usage is **not allowed**. This is not a dynamic data structure task (wait till the3).