Computational Systems COMP2002

Coursework 2

Individual Coursework *(cf. collaboration policy)*

- *Aims of this coursework:*
  1. *Programming recursive functions on all forms of trees.*
  2. *Assignments and side-effects are* **not** *allowed.*
- *Deadline: Week 11, Thursday 15th December 2011, 4.00pm.*
- *Reading: chapters 1, 2, 3, 4, 5, 6, 7 of lecture notes.*
- *DrRacket Language: Standard (R5RS)*
- *Electronic submission: Four files:*
  1. `formula.scm`*: answer to question 1.*
  2. `formula-string.scm`*: answer to question 2.*
  3. `evaluate-formula.scm`*: answer to question 3.*
  4. `formula-nnf.scm`*: answer to question 4.*
- *Paper submission: Please note that there is no paper submission for this coursework.*
- *Marking scheme: Each question is worth 25%.*

You are required to implement a number of functions that will form part of a system for manipulating and evaluating propositional logic formulae. The system uses a list-based prefix representation for formulae; well-formed formulae will conform to the following grammar:

$$\begin{array}{lll} \langle wff \rangle & ::= & \langle symbol \rangle \\ & & \texttt{(not } \langle wff \rangle \texttt{)} \\ & & \texttt{(and } \langle wffs \rangle \texttt{)} \\ & & \texttt{(or } \langle wffs \rangle \texttt{)} \\ \\ \langle wffs \rangle & ::= & \langle wff \rangle \\ & & \langle wff \rangle \ \langle wffs \rangle \end{array}$$

Note that the symbols `not`, `and` and `or` are reserved, and may not be used as atomic propositions.

1. Define a function `formula?` that takes an expression and returns `#t` if it is a well-formed formula according to the grammar above, and `#f` otherwise.

   The function `formula?` produces the following results:

```
(formula?  '())                      ⇒  #f
(formula?  'a)                       ⇒  #t
(formula?  '(not a))                 ⇒  #t
(formula?  '(not a b))               ⇒  #f
(formula?  '(and a b))               ⇒  #t
(formula?  '(or a b))                ⇒  #t
(formula?  '(or (not a) (and b c)))  ⇒  #t
```

2. Define a function `formula->string` that takes a well-formed formula and returns a string containing an infix representation of the formula suitable for pretty-printing. The `not` operator should be replaced with `~`, the `or` operator with `v` and the `and` operator with `^`, and all symbols should be rendered in upper case. Parentheses should be added where necessary to avoid ambiguities.

   The function `formula->string` produces the following results:

```
(formula->string 'a)                        ⇒  "A"
(formula->string '(not a))                  ⇒  "~A"
(formula->string '(not (not a)))            ⇒  "~~A"
(formula->string '(or a b c))               ⇒  "A v B v C"
(formula->string '(or (not a) (and b c))    ⇒  "~A v (B ^ C)"
```

3. Define a function `evaluate-formula` that takes a well-formed formula and an interpretation (an alist of atomic propositions and truth values) and returns the value of the formula under that interpretation. You may assume that the interpretation contains truth values for all of the atomic propositions in the formula.

   The function `evaluate-formula` produces the following results:

```
(evaluate-formula 'a '((a . #t))                    ⇒  #t
(evaluate-formula 'a '((a . #f))                    ⇒  #f
(evaluate-formula '(not a) '((a . #f))              ⇒  #t
(evaluate-formula '(and a b) '((a . #t) (b . #f))   ⇒  #f
(evaluate-formula '(or a b) '((a . #t) (b . #f))    ⇒  #t
```

4. Define a function `formula->nnf` that takes a well-formed formula and returns the formula rewritten into negation normal form. In negation normal form, negation only occurs immediately before atomic propositions. For example, the following formula is in NNF:

$$(\neg a \wedge b) \vee \neg c \vee d$$

   whereas this logically equivalent formula is not:

$$\neg((a \vee \neg b) \wedge c \wedge \neg d)$$

   In order to carry out this transformation, you will need to make use of the following logical equivalences:

$$\begin{aligned} \neg\neg a &\equiv a \\ \neg(a \wedge b) &\equiv \neg a \vee \neg b \\ \neg(a \vee b) &\equiv \neg a \wedge \neg b \end{aligned}$$

   The function `formula->nnf` produces the following results:

```
(formula->nnf 'a)                       ⇒  a
(formula->nnf '(not a))                 ⇒  (not a)
(formula->nnf '(not (not a))            ⇒  a
(formula->nnf '(not (and a b))          ⇒  (or (not a) (not b))
(formula->nnf '(not (or a b))           ⇒  (and (not a) (not b))
(formula->nnf '(not (and a (not b)))    ⇒  (or (not a) b)
```