CMSC 22610 Winter 2009	Implementation of Computer Languages - I	Homework 1 January 6, 2009

Homework 1

Due: January 13, 2009

1. Consider a language of *propositional formulae* formed from variables (a, b, c, ...), negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$), according to the following abstract syntax:

$$\begin{array}{rcl} \phi & ::= & a \\ & | & \neg\phi \\ & | & \phi_1 \wedge \phi_2 \\ & | & \phi_1 \vee \phi_2 \end{array}$$

We can represent propositional formulae in SML using the following datatype:

```
datatype prop = Var of string
              | Not of prop
              | And of prop * prop
              | Or of prop * prop
```

For example, the formula $a \wedge \neg(b \vee \neg c)$ is represented as the following SML value:

```
And (Var "a", Not (Or (Var "b", Not (Var "c"))))
```

The language of *disjunctive normal formulae* (DNF) is given by the following abstract syntax:

$$\begin{array}{rcl} A & ::= & a \\ & | & \neg a \\ C & ::= & A \\ & | & A \wedge C \\ D & ::= & C \\ & | & C \vee D \end{array}$$

We can represent disjunctive normal formulae in SML using the following datatypes:

```
datatype atom = Var of string | Not of string
datatype conjunct = Atom of atom | And of atom * conjunct
datatype disjunct = Conj of conjunct | Or of conjunct * disjunct
```

Because we have used the same constructor names, we must put the `prop` and `disjunct` types in separate modules:

```
structure Prop =
  struct
     datatype prop = ...
  end
structure DNF =
  struct
     datatype atom = ...
     datatype conjunct = ...
     datatype disjunct = ...
  end
```

One can convert an arbitrary formula to DNF by repeated application of the following rewrite rules:

$$\begin{array}{rcl} \neg(\neg\phi) & \Rightarrow & \phi \\ \neg(\phi_1 \wedge \phi_2) & \Rightarrow & \neg\phi_1 \vee \neg\phi_2 \\ \neg(\phi_1 \vee \phi_2) & \Rightarrow & \neg\phi_1 \wedge \neg\phi_2 \\ \phi_1 \wedge (\phi_2 \vee \phi_3) & \Rightarrow & (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3) \\ (\phi_1 \vee \phi_2) \wedge \phi_3 & \Rightarrow & (\phi_1 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3) \end{array}$$

Write an SML function `toDNF` that converts propositional formulae to their equivalent DNF. It should have the following signature:

```
val toDNF : Prop.prop -> DNF.disjunct
```

Your solution should consist of four files: `prop.sml` (holding the module `Prop`), `dnf.sml` (holding the module `DNF`), `convert.sml` (holding the module `Convert`, which contains the `toDNF` function), and `hw1.cm` (containing the CM specification). Please ensure that your name appears in a comment at the beginning of each file.

The CM specification should be as follows:

```
Library
  structure Prop
  structure DNF
  structure Convert
is
  $/basis.cm
  prop.sml
  dnf.sml
  convert.sml
```

Submission: Put your solution in a directory named *xxx*`-hw1`, where "*xxx*" is your login ID. Create a tar file from the directory (please do not include the `.cm` subdirectory) and email it to the TA (`ams@cs.uchicago.edu`) by 1:30pm, January 13.

Hint: One approach to this problem is to stage it as two steps: first you push the negations to the leaves, which results in a "simple" formula formed from disjunction, conjunction, and atoms. Then convert the simple formula into DNF.

2. In the following, you are asked to give LangF declarations. The purpose of this exercise is to familiarize yourself with the LangF language and to understand its syntax and semantics. Be sure to carefully review the LangF project overview handout.

   (a) Give a LangF declaration for a datatype `Prop` that represents propositional formulae; that is, translate the SML datatype `prop` given above into LangF.

   (b) Give a LangF declaration for a polymorphic datatype `Option` with one type parameter and two constructors — one constructor has no arguments and the other constructor has one argument having the type of the type parameter; that is, translate the SML datatype `'a option` into LangF.

   (c) Recall the list datatype from the LangF project overview:

   ```
   datatype List ['a] = Nil | Cons {'a, List ['a]}
   ```

   Give a LangF declaration for a function `exists`, with the LangF type

   ```
   ['a] -> ('a -> Bool) -> List ['a] -> Bool
   ```

such that applying `exists` to a type *ty*, a function *p*, and a list *l*, applies *p* to each element of the list *l*, from left to right, until *p* applied to an element evaluates to `True`; it returns `True` if such an element exists and `False` otherwise.

For example, the following LangF expression should evaluate to `False`:

```
exists [Integer]
       (fn (i: Integer) => i > 15)
       (Cons [Integer] {10,
        Cons [Integer] {5,
        Cons [Integer] {0, Nil [Integer]}}})
```

and the following LangF expression should evaluate to `True` (without aborting):

```
exists [Integer]
       (fn (i: Integer) =>
          if i < 5 then fail [Bool] "" else i == 5)
       (Cons [Integer] {10,
        Cons [Integer] {5,
        Cons [Integer] {0, Nil [Integer]}}})
```

(d) Give a LangF declaration for a function `map`, with the LangF type

```
['a] -> ['b] -> ('a -> 'b) -> List ['a] -> List ['b]
```

such that applying `map` to a type ty_1, a type ty_2, a function *f*, and a list *l*, applies *f* to each element of the list *l*, from left to right, returning the list of results.

For example, the LangF expression:

```
map [Integer] [Integer]
    (fn (i: Integer) => i * 2)
    (Cons [Integer] {10,
     Cons [Integer] {5,
     Cons [Integer] {0, Nil [Integer]}}})
```

should evaluate to the LangF value:

```
(Cons [Integer] {20,
 Cons [Integer] {10,
 Cons [Integer] {0, Nil [Integer]}}})
```

Submission: Hand in your solutions at the beginning of class on January 13. Since there are syntactic elements of LangF programs that could be difficult to distinguish when hand written (e.g., `{` vs. `(`, lower-case vs. upper-case), typing up and printing out your solution is recommended.

Document history

January 9, 2009 Fixed typo in **`datatype`** `disjunct Or` constructor type (should be `conjunct`, not `conjuct`) and `toDNF` type (should be `-> DNF.disjunct`, not `-> DNF.conjunct`).

January 8, 2009 Fixed typo in **`datatype`** `conjunct` (missing `Atom` variant) and **`datatype`** `conjunct` (missing `Conj` variant).

January 7, 2009 Fixed typos in **`datatype`** `prop` (`And` and `Or` variants), DNF syntax (second *D* production), and in rewrite rule (last RHS).

January 6, 2009 Original version