Homework 6
SNU 4910.210 Fall 2014

Chung-Kil Hur

due: 11/23(Sun) 24:00

Exercise 1 "The Number of Ways"

On the chessboard of size $N \times N$, a function `ways` that counts the number of minimum-length ways from $(0, 0)$ to (n, m) can be recursively defined as follows:

```
(define (ways n m)
  (cond ((= n 0) 1)
        ((= m 0) 1)
        (else (+ (ways (- n 1) m)
                 (ways n (- m 1))))))
```

However, this recursive function is very inefficient so that it cannot calculate `(ways 50 50)` immediately. Consult with the Section 3.3.3 and Exercise 3.27 of the textbook(SICP) in order to define `memo-ways`, which memorizes the results and then uses it in later executions.

$$\texttt{memo-ways} : nat \times nat \to nat$$

□

Exercise 2 "Tournament Strings"

Usually, matches of a tournament is a complete binary tree. We defined the teams of 2013 Woorld Cup and the tournament matches like this:

```
type team = Korea | France | Usa | Brazil | Japan | Nigeria | Cameroon
          | Poland | Portugal | Italy | Germany | Norway | Sweden | England
          | Argentina
type tourna = LEAF of team
            | NODE of tourna * tourna
```

Write a function `parenize` that takes an input of `tourna` and then transform it into a string:

```
parenize:  tourna -> string
```

For example,

```
parenize(NODE(NODE(LEAF Korea, LEAF Portugal), LEAF Brazil))
= "((Korea Portugal) Brazil)"
```

□

Exercise 3 "Drop"

Now, write a function `drop`:

```
drop:  tourna * team -> string
```

`drop(t, Brazil)` prints a new tournament matches after the `Brazil` team is eliminated (lost) from the tournament. Use `parenize` in Exercise 2. □

Exercise 4 "True or False"

We defined the formulas of Propositional logic as follows:

```
type formula = TRUE
             | FALSE
             | NOT of formula
             | ANDALSO of formula * formula
             | ORELSE of formula * formula
             | IMPLY of formula * formula
             | LESS of expr * expr
and  expr = NUM of int
          | PLUS of expr * expr
          | MINUS of expr * expr
```

Define a function `eval`

```
eval:  formula -> bool
```

which calculates the boolean result of the given input formula. □

Exercise 5 (10pts) "Mathemadiga"

Let's make a automatic-differentiation tool, like Maple or Mathematica.

```
diff:  ae * string -> ae
```

`diff` takes an algebraic expression and a variable, and it returns the differentiated expression with respect to the variable.

For example, the differentiation of $ax^2 + bx + c$ with respect to x is $2ax + b$.

You have freedom to apply further operations such as simplifying the result. The expression is given with a type `ae`:

```
type ae = CONST of int
        | VAR of string
        | POWER of string * int
        | TIMES of ae list
        | SUM of ae list
```

□

Exercise 6 "Crazy-k"

Numbers in base-k ($k > 1$) are usually represented as follows:

$$d_0 \cdots d_n$$

where

$$\forall d_i \in \{0, \cdots, k-1\}.$$

and "$d_0 \cdots d_n$" denotes the integer

$$d_0 \times k^0 + \cdots + d_n \times k^n \ .$$

Let us define "crazy-k" as follows by slightly extending "base-k". Numbers in crazy-k ($k > 1$) are represented as follows:

$$d_0 \cdots d_n$$

where

$$\forall d_i \in \{1-k, \cdots, 0, \cdots, k-1\}.$$

and "$d_0 \cdots d_n$" denotes the integer

$$d_0 \times k^0 + \cdots + d_n \times k^n \ .$$

For example, consider crazy-2 with $\{-1, 0, 1\}$ as digits. Suppose that `0`, `+` and `-` represent 0, 1 and -1 respectively. Then, `+`, `+0+`, `+-` and `+-0-` denote 1, 5, -1 and -9 respectively.

We defined a type `crazy2` in OCaml as follows:

```
type crazy2 = NIL | ZERO of crazy2 | ONE of crazy2 | MONE of crazy2
```

For example, `0+-`is represented as:

```
ZERO(ONE(MONE NIL))
```

Define a function `crazy2val` that takes a `crazy2` value and then calculates the corresponding integer value.

```
crazy2val:  crazy2 -> int.
```

□

Exercise 7 "Addition in Crazy-2"

Define a function `crazy2add` that takes two numbers in crazy-2 and evaluates to their sum in crazy-2.

`crazy2add` : Crazy-2 $*$ Crazy-2 $\rightarrow$ Crazy-2.

`crazy2add` should satisfy the following properties:

- For any z and z' in crazy-2,

  `(crazy2val (crazy2add` z z'`))` $=$ `(crazy2val` z`)` $+$ `(crazy2val` z'`)`.

- `crazy2add` should be defined recursively. Note that it is not allowed to convert numbers in crazy-2 into integers, add them as integers, and revert the sum back into crazy-2.

□

Challenge 2 "Pleasant Worry"

Younghee's worry is to buy gifts for her nephews with the least cost while satisfying all of them. Her nephews are so jealous. Every year, she gives a gift to each of them, but they always compare the gifts with each other and then become jealous, fighting, crying...

This year, Younghee decided to solve this problem like this way. Before shopping, she gives the whole list of goods to the nephews, and let them tell the condition in which they are satisfied.

The nephews' conditions are like this: "I must have a fountain pen and, at least all things that C receives" "I must have things that both A and B receive in common, and all things that C receives without a CD." and so on. These are some examples of presents that the nephews will receive, according to their conditions:

- Jealous nephews get nothing. A: "At least all things B gets ", B: "At least all things A gets ", C: "At least all things B gets "
- Picky nephews get nothing. A: "At least all things B gets, without fountain pen", B: "At least all things A gets, without CD ", C: "At least all things B gets, without USB memory stick"
- Greedy nephews get nothing. A: "At least all things B and C get", B: "At least all things A and C get ", C: "At least all things A and B get "
- Unselfish nephews get only things they want. A: "At least a fountain pen", B: "At least CD", C: "At least USB memory stick."

The conditions of nephews are expressed like this:

$$\text{"I must have at least } (cond_1 \text{ and } \cdots \text{ and } cond_k)\text{."}$$

Now write a function `shoppingList` that takes conditions of nephews as an input, and then makes a list of gifts to buy.

$$\texttt{shoppingList} : (id \times cond)\ list \to (id \times gift\ list)\ list$$

The result is a list of gifts for each nephew. For example, if the conditions are like this,

$$\begin{array}{lll} A: & \text{at least} & (\{1,2\} \text{ and } \text{common}(B, C)) \\ B: & \text{at least} & \text{common}(C, \{2,3\}) \\ C: & \text{at least} & (\{1\} \text{ and } (A \text{ except } \{3\})) \end{array}$$

Since the minimal gifts are $\{1, 2\}$ for A, $\{2\}$ for B, and $\{1, 2\}$ for C, the result of `shoppingList` is:

```
((A . (1 2)) (B . (2)) (C . (1 2)))
```

There are no two identical gifts for one nephew.

For implementation, the functions that build/use the conditions are given as follows.

- For building:

`mustItems` : *giftlist* $\rightarrow$ *cond*	must have those gifts
`mustBeTheSame` : *id* $\rightarrow$ *cond*	must have gifts that someone has
`mustHaveExceptFor` : *cond* $*$ *giftlist* $\rightarrow$ *cond*	must satisfy condition without some gifts
`mustHaveCommon` : *cond* $*$ *cond* $\rightarrow$ *cond*	must have common gifts in two conditions
`mustAnd` : *cond* $*$ *cond* $\rightarrow$ *cond*	must satisfy both conditions

- For using:

`isItems` : *cond* $\rightarrow$ *bool*	`isSame` : *cond* $\rightarrow$ *bool*
`isExcept` : *cond* $\rightarrow$ *bool*	`isCommon` : *cond* $\rightarrow$ *bool*
`isAnd` : *cond* $\rightarrow$ *bool*	`whichItems` : *cond* $\rightarrow$ *giftlist*
`whoTheSame` : *cond* $\rightarrow$ *id*	`condExcept` : *cond* $\rightarrow$ *cond*
`itemsExcept` : *cond* $\rightarrow$ *giftlist*	`condCommon` : *cond* $\rightarrow$ *cond* $\times$ *cond*
`condAnd` : *cond* $\rightarrow$ *cond* $\times$ *cond*	

In the above, *gift* is implemented as integers, and a nephew's name *id* as symbols in Racket. For example, the condition of the nephew A in the above is like this: `(mustAnd (mustItems '(1 2)) (mustHaveCommon (mustBeTheSame 'B) (mustBeTheSame 'C)))` □