Fall 2016 **Programming Project #5 — Databases** CMSC 420

Hanan Samet

This project involves a data base management system for the army, with data processing operators typically found in the network model. The records in the data base correspond to members in the army. For each member the following information has been gathered:

`weight` — in lbs.
`height` — in centimeters
`sex` — value `m` or `f`
`univ`(ersity) — `nil` if none
`year` — when entered service
`age` — when entered service
`salary`
`homestate`
`married` — value `y` or `n`
`kidnames` — a list: `nil` if none

The data for each member of the army is stored on the property list of the members name. There is also an atom with the name `officers` whose value is a list of the names of all of the officers. For example: `(jones smith denton)`. An example data base entry is shown in Figure **??**.

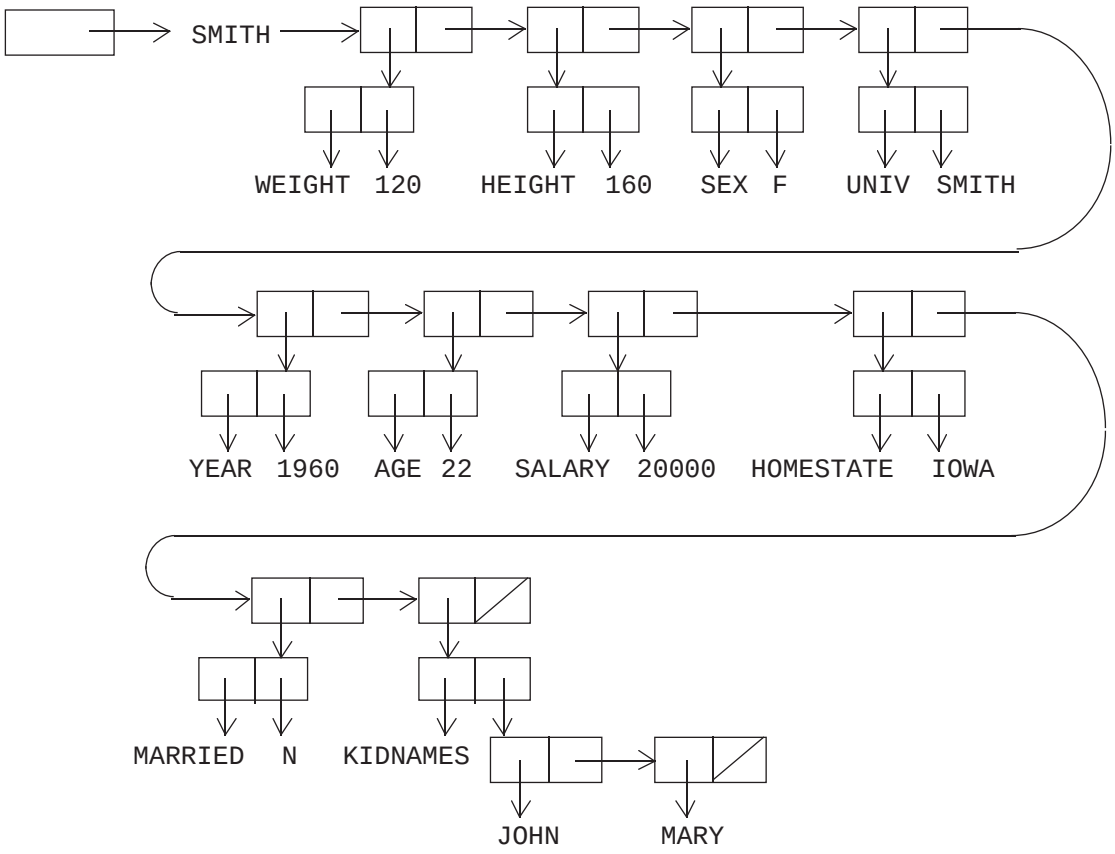

Figure 1: Example database entry

Given such a data base system there are two types of operations that you are to support.

- data base modification operations
- queries

In addition to the records, we also have entities known as sets. In our case these sets are lists whose elements are names who share certain properties. For our purposes the sets correspond to rank. We shall assume 5 ranks. They are:

```
general
colonel
major
lieutenant
private
```

Note, that we assume that the set `officers` includes all ranks, including `private` (even though technically a private is not officer in army terminology).

You are to provide routines to support the following data base modification operations:

`(store name proplist)`	An officer name `name` is entered into the data base with property values as given in `proplist`. `proplist` is a list of dotted pairs whose `car` is a property name and `cdr` is a property value. Note that `name` must also be added to the list `officers` if not already a member.
`(insert name setname)`	Inserts the officer with name `name` into set `setname` if not already a member of the set.
`(removeofficer name setname)`	Removes the officer with name `name` from set `setname` if `name` is a member of the set.
`(deleteofficer name)`	Deletes the officer with name `name` from the data base. This consists of removal from the list `officers` as well as from all sets in which `name` is a member.
`(modify name prop value)`	Modify the property `prop` of officer `name` to be `value`.

Some error checking of arguments must be implemented. For example, a name cannot be inserted into a set unless it has been entered into the database, and the setnames given in `insert` and `removeofficer` must be a name of one of the 5 ranks above. Should the arguments be illegal, the function should return `nil`; otherwise, any non-`nil` value can be returned (e.g., the set after insertion for the `insert` operation, but then you must watch for the case when the set is empty).

Two types of query operations are to be supported.

1. With respect to the entire data base — i.e., search using items in `officers`
2. With respect to a specific set.

A query operation may return a set of names satisfying a condition or merely a single atom.

Associated with each set, including `officers`, is an atom called `currentof`-setname. This atom has as its value a list corresponding to the remainder of the set after the last `findnext` or `findnextinset` operation performed on the set. This value is reset to point to the entire set whenever an `insert`, `removeofficer`, or `deleteofficer` operation affecting the set has occurred. In addition, whenever a `findnext` or `findnextinset` operation finds current of the set in question empty, then it resets it to point to the entire set prior to performing the operation. Currency is a feature common to many database management systems.

Query Operations have the form:

`(operation setname condition)`	`setname` is `officers`, `general`, `colonel`, `major`, `lieutenant` or `private`.
`(operation condition)`	`officers` is assumed to be the setname.

Query Operations:

`(findall condition)`	Returns a list of all names in `officers` satisfying `condition`. Resets `currentof-officers` to `officers`.
`(findallinset setname condition)`	Finds all names in `setname` satisfying `condition`. Resets `currentof-setname` to `setname`.
`(findnext condition)`	Returns the first name in the list pointed at by `currentof-officers` which satisfies `condition`. `currentof-officers` is set to the remaining list.
`(findnextinset setname condition)`	Returns the first name in the list pointed at by `currentof-setname` which satisfies `condition`. `currentof-setname` is reset to the remaining list.
`(findany condition)`	Any member of the set `officers` satisfying `condition` can be returned. `currentof-officers` is not affected by this operation.
`(findanyinset setname condition)`	Any member of the set `setname` satisfying `condition` can be returned. `currentof-setname` is not affected by this operation.

Note that `findnext` and `findnextinset` are cyclic, in the sense that if they reach the end of a list, then they start over from the beginning. Care must be taken that they will not get into an infinite loop should there be no officer in the list that satisfies the condition. As an example, suppose that the set `major` contained no officer of age 41. In this case the query `(findnextinset major (eqp age 41))` should return `nil`, and leave `currentof-major` unchanged. However, if `currentof-major` points at the third officer in the set `major`, and the first officer happened to be of age 41 while none of the others were, then this query should return the name of the first officer and leave `currentof-major` pointing to the second element.

The format of `condition` is:

```
(<logical operator> <logical argument1>
                    <logical argument2>
                    .
                    .
                    .
                    <logical argumentn>)
```

or: `(<relational operator> <property> <value>)`
`<logical operator>` is `and`, `or`, `not`
`<logical argumenti>` is a condition
`<relational operator>` is `eqp`, `nep`, `lep`, `gep`, `ltp`, `gtp`

The relational operators correspond to `eq` (actually, `equal` is a better choice, since `eq` always

returns `nil` on sufficiently large numbers), `ne`, `le`, `ge`, `lt`, `gt`. There are several ways to implement the evaluation of conditions. You can let the LISP interpreter do most of the work, and simply use `eval`. One method is to use `eval`, and let the functions that implement the relational operators access the property list of the atom for which the condition is being evaluated (which has to be stored in a global variable). Another is to also use `eval`, but don't access the property list each time a relational operator is evaluated. Instead, before invoking `eval`, define a variable for each possible property name, and give it the value of the corresponding property of the atom for which the condition is to be evaluated. A third method is to write your own evaluation function, which might be called `condeval`, which takes as arguments both the condition and the atom for which the condition is to be evaluated.

Some typical conditions and queries are:

```
(and (gtp salary 10000) (eqp sex m))
```

will be true for males with salary greater than 10000.

```
(findallinset colonel (eqp sex f))
```

will find all female colonels

```
(findall (or (and (eqp sex m) (gtp height 170))
             (and (eqp sex f) (gtp height 160))))
```

will find all male officers of height greater than 170 centimeters and female officers of height greater than 160 centimeters.

Some typical operations on our data base are:

1. promotions — use `removeofficer` and `insert`
2. enlistment — use `store`
3. retirement — use `deleteofficer`
4. changes in individual status — use `modify`
   e.g., change in height, weight, number of kids, etc.

A farfetched example (or not so farfetched today!) is the case when a divorced female colonel, `smith`, with two kids marries a widower male lieutenant, `jones`, with three kids. Due to considerations of rank, the lieutenant retires. Updating the data base for this case requires us to merge the `kidnames` of `jones` with those of `smith`, and `modify` `smith`s entry accordingly. Similarly, the marital status of `smith` must be updated. Due to `jones` retirement, he is `deleteofficer`'d from the data base which causes his removal from the set `lieutenant`. Despite remaining in the army `smith` is not such a liberated woman that she does not take on her husbands name. Thus one must also `store` an entry for `jones` in the data base whose property values are identical to those of `smith`. `jones` must also be `inserted` into the `colonel` set. Once this is done, `smith` is `deleteofficer`'d from the data base which causes the removal of the `smith` entry from the `colonel` set. Note that if our data base were organized in a manner such that `name` and `rank` were also properties (e.g., as is done in a conventional database management system) then the name change could be achieved by a `modify` operation.

You are to define the appropriate functions in LISP. Test your functions with the data sent to you. You should turn in a listing of your run which includes the function definitions and the results of the application of the test data.

Programming Hints

First consider the following expressions and their printed values after having performed:

```
(setq x 'z)
```

expression	printed value
`x`	`z`
`'x`	`x`
`(x)`	`x is not a function`
`(list x)`	`(z)`

The `q` of `setq` stands for quote hence `(setq x 'z)` is equivalent to `(set 'x 'z)`. Consider what happens if we now perform `(set x 'a)` immediately after `(setq x 'z)`.

expression	printed value
`x`	`z`
`(list x)`	`(z)`
`z`	`a`
`(list z)`	`(a)`

Note that you cannot `setq` to an atom which is used as a formal parameter name. However, you can `set` to a formal parameter because `set` changes the value of the atom bound to the formal parameter name and not the value of the parameter name itself.

The atoms of the form `currentof`-setname are maintained via `set` and `setq`. They can be created on the fly via `explode` and `implode` in FRANZ LISP (see below for what to do in COMMON LISP).

expression	printed value
`(implode '(a b c))`	`abc`
`(explode 'abc)`	`(a b c)`
`(implode (explode 'abc))`	`abc`
`(explode (implode '(a b c)))`	`(a b c)`

Finally, note that there will be no quotes in the test data (there will be test data!), i.e., the command `(deleteofficer john)` will be used instead of the command `(deleteofficer 'john)`. To prevent getting `unbound john` messages, it will be necessary to define the function using the `fexpr` option (this applies to FRANZ LISP). The `fexpr` option allows the function to accept any number of arguments, none of which is evaluated. When a `fexpr` is called, it can be supplied with as many arguments as desired. LISP will take all the arguments specified, put them together in a list, and bind this list to the single formal parameter of the function. For example:

```
(defun echo fexpr (var) (car var))
(defun echolist fexpr (var) var)
(defun listeval fexpr (var) (eval var))
```

expression	printed value
`(echo me)`	`me`
`(echo me you)`	`me`
`(echolist add 1 3)`	`(add 1 3)`
`(listeval add 1 3)`	`4`

In COMMON LISP, the fexpr option does not exist, and neither do implode and explode. Instead, use the functions `defun.fexpr` and `concat-symbols` defined in section 10 of the LISP reference, repeated below. The former acts like `defun` with the `fexpr` option, e.g., `(defun.fexpr echo (var) (car var))`. The latter may be used to create the `currentof-setname` symbols, e.g., `(concat-symbols 'currentof- 'general)`.

```
(defmacro defun.fexpr (funname varlist &rest expr)
  (let ((ffunname (concat-symbols funname '.fexpr)))
    `(progn
      (defun ,ffunname ,varlist ,@expr)
      (defmacro ,funname (&body args)
        (list ',ffunname (list 'quote args))))))

(defun concat-symbols (sym1 sym2)
  (intern (concatenate 'string (string sym1) (string sym2))))
```