SICP section 4.4.1

February 08, 2008 at 16:06 Tags SICP

Exercise 4.55

a.

```
(supervisor ?x (Bitdiddle Ben))
=>
(SUPERVISOR (TWEAKIT LEM E) (BITDIDDLE BEN))
(SUPERVISOR (FECT CY D) (BITDIDDLE BEN))
(SUPERVISOR (HACKER ALYSSA P) (BITDIDDLE BEN))
```

b.

```
(job ?name (accounting . ?a))
=>
(JOB (CRATCHET ROBERT) (ACCOUNTING SCRIVENER))
(JOB (SCROOGE EBEN) (ACCOUNTING CHIEF ACCOUNTANT))
```

c.

```
(address ?name (Slumerville . ?a))
=>
(ADDRESS (AULL DEWITT) (SLUMERVILLE (ONION SQUARE) 5))
(ADDRESS (REASONER LOUIS) (SLUMERVILLE (PINE TREE ROAD) 80))
(ADDRESS (BITDIDDLE BEN) (SLUMERVILLE (RIDGE ROAD) 10))
```

Exercise 4.56

a.

```
(and
  (supervisor ?name (Bitdiddle Ben))
  (address ?name ?addr))
=>
(AND (SUPERVISOR (TWEAKIT LEM E) (BITDIDDLE BEN)) (ADDRESS (TWEAKIT LEM E) (BOSTON (BAY STATE ROAD)
(AND (SUPERVISOR (FECT CY D) (BITDIDDLE BEN)) (ADDRESS (FECT CY D) (CAMBRIDGE (AMES STREET) 3)))
(AND (SUPERVISOR (HACKER ALYSSA P) (BITDIDDLE BEN)) (ADDRESS (HACKER ALYSSA P) (CAMBRIDGE (MASS AVE)
```

b.

```
(and (salary (Bitdiddle Ben) ?bens)
     (salary ?who ?whos)
     (lisp-value #'> ?bens ?whos))
=>
(AND (SALARY (BITDIDDLE BEN) 60000) (SALARY (AULL DEWITT) 25000) (LISP-VALUE #'> 60000 25000))
(AND (SALARY (BITDIDDLE BEN) 60000) (SALARY (CRATCHET ROBERT) 18000) (LISP-VALUE #'> 60000 18000))
(AND (SALARY (BITDIDDLE BEN) 60000) (SALARY (REASONER LOUIS) 30000) (LISP-VALUE #'> 60000 30000))
(AND (SALARY (BITDIDDLE BEN) 60000) (SALARY (TWEAKIT LEM E) 25000) (LISP-VALUE #'> 60000 25000))
(AND (SALARY (BITDIDDLE BEN) 60000) (SALARY (FECT CY D) 35000) (LISP-VALUE #'> 60000 35000))
(AND (SALARY (BITDIDDLE BEN) 60000) (SALARY (HACKER ALYSSA P) 40000) (LISP-VALUE #'> 60000 40000))
```

c.

```
(and (supervisor ?serf ?boss)
     (not (job ?boss (computer . ?c)))
     (job ?boss ?bossjob))
=>
(AND (SUPERVISOR (AULL DEWITT) (WARBUCKS OLIVER)) (NOT (JOB (WARBUCKS OLIVER) (COMPUTER . ?C)))
 (JOB (WARBUCKS OLIVER) (ADMINISTRATION BIG WHEEL)))
(AND (SUPERVISOR (CRATCHET ROBERT) (SCROOGE EBEN)) (NOT (JOB (SCROOGE EBEN) (COMPUTER . ?C)))
 (JOB (SCROOGE EBEN) (ACCOUNTING CHIEF ACCOUNTANT)))
(AND (SUPERVISOR (SCROOGE EBEN) (WARBUCKS OLIVER)) (NOT (JOB (WARBUCKS OLIVER) (COMPUTER . ?C)))
 (JOB (WARBUCKS OLIVER) (ADMINISTRATION BIG WHEEL)))
(AND (SUPERVISOR (BITDIDDLE BEN) (WARBUCKS OLIVER)) (NOT (JOB (WARBUCKS OLIVER) (COMPUTER . ?C)))
 (JOB (WARBUCKS OLIVER) (ADMINISTRATION BIG WHEEL)))
```

Exercise 4.57

Here's the rule:

```
(assert!
  (rule (can-replace ?1 ?2)
        (and (or (and (job ?1 ?job) (job ?2 ?job))
                 (and (job ?1 ?j1)
                      (job ?2 ?j2)
                      (can-do-job ?j1 ?j2)))
             (not (same ?1 ?2)))))
```

I had a curious incident while writing it. I placed the `(not (same ?1 ?2))` query first in the `and` and wondered why nothing works. See "Problems with `not`" in section 4.4.3 for an answer.

a.

```
(can-replace ?t (Fect Cy D))
=>
(CAN-REPLACE (BITDIDDLE BEN) (FECT CY D))
(CAN-REPLACE (HACKER ALYSSA P) (FECT CY D))
```

b.

```
(and
  (can-replace ?1 ?2)
  (salary ?1 ?s1)
  (salary ?2 ?s2)
  (lisp-value #'> ?s2 ?s1))
=>
(AND (CAN-REPLACE (AULL DEWITT) (WARBUCKS OLIVER)) (SALARY (AULL DEWITT) 25000)
 (SALARY (WARBUCKS OLIVER) 150000) (LISP-VALUE #'> 150000 25000))
(AND (CAN-REPLACE (FECT CY D) (HACKER ALYSSA P)) (SALARY (FECT CY D) 35000) (SALARY (HACKER ALYSSA P
 (LISP-VALUE #'> 40000 35000))
```

Exercise 4.58

```
(assert!
  (rule (bigshot ?person ?division)
        (and
          (job ?person (?division . ?r))
          (or
            (not (supervisor ?person ?boss))
            (and
              (supervisor ?person ?boss)
              (not (job ?boss (?division . ?q))))))))
```

Lets try it:

```
(bigshot ?who ?where)
=>
(BIGSHOT (WARBUCKS OLIVER) ADMINISTRATION)
(BIGSHOT (SCROOGE EBEN) ACCOUNTING)
(BIGSHOT (BITDIDDLE BEN) COMPUTER)
```

Exercise 4.59

a.

```
(meeting ?dept (Friday . ?time))
=>
(MEETING ADMINISTRATION (FRIDAY 1PM))
```

b.

```
(assert!
  (rule (meeting-time ?person ?day-and-time)
        (or (meeting whole-company ?day-and-time)
            (and
              (job ?person (?div . ?r))
              (meeting ?div ?day-and-time)))))

(meeting-time (Hacker Alyssa P) ?time)
=>
(MEETING-TIME (HACKER ALYSSA P) (WEDNESDAY 4PM))
(MEETING-TIME (HACKER ALYSSA P) (WEDNESDAY 3PM))
```

c.

```
(and
  (meeting-time (Hacker Alyssa P) (Wednesday . ?time))
  (meeting ?div (Wednesday . ?time)))
=>
(AND (MEETING-TIME (HACKER ALYSSA P) (WEDNESDAY 4PM)) (MEETING WHOLE-COMPANY (WEDNESDAY 4PM)))
(AND (MEETING-TIME (HACKER ALYSSA P) (WEDNESDAY 3PM)) (MEETING COMPUTER (WEDNESDAY 3PM)))
```

Exercise 4.60

To understand why this happens, we must recall how rules are matched. From "Applying rules" in section 4.4.2:

In general, the query evaluator uses the following method to apply a rule when trying to establish a query pattern in a frame that specifies bindings for some of the pattern variables:

- Unify the query with the conclusion of the rule to form, if successful, an extension of the original frame.
- Relative to the extended frame, evaluate the query formed by the body of the rule.

When we execute the first step, we unify the query:

```
(lives-near ?person-1 ?person-2)
```

With the conclusion of the rule:

```
(lives-near ?p1 ?p2)
```

To form an extended frame, in which `?person-1` is mapped to `?p1` and `?person-2` is mapped to `?p2`. We next evaluate the query formed by the body of the rule relative to this extended frame. But since the extended frame doesn't really impose any restrictions on the `?p1` and `?p2` variables, it is equivalent to just evaluating the body of the rule as a query:

```
(and
  (address ?p1 (?town . ?rest1))
  (address ?p2 (?town . ?rest2))
  (not (same ?p1 ?p2)))
```

And when we do this, we indeed get all the pairs of people living near each other, twice.

We can ask for a list in which pairs appear only once by modifying the `lives-near` rule. We'll do this by imposing some order on the names instead of using `not`. The order must be an asymmetric relation – for example "greater than". But we didn't build such relations into our logic evaluator, so I'll write it as an ordinary Lisp function and invoke it using `lisp-value`:

```
(defun greater-as-string (obj1 obj2)
  (string>
    (write-to-string obj1)
    (write-to-string obj2)))

(qinterpret
  '(assert!
    (rule (lives-near ?p1 ?p2)
          (and
            (address ?p1 (?town . ?rest1))
            (address ?p2 (?town . ?rest2))
            (lisp-value #'greater-as-string ?p1 ?p2)))))

(qinterpret
  '(lives-near ?person ?p2))
=>
(LIVES-NEAR (REASONER LOUIS) (AULL DEWITT))
(LIVES-NEAR (REASONER LOUIS) (BITDIDDLE BEN))
(LIVES-NEAR (HACKER ALYSSA P) (FECT CY D))
(LIVES-NEAR (BITDIDDLE BEN) (AULL DEWITT))
```

Note that the invocation of `lisp-value` has replaced `(not (same ...`, because the relation "greater than" subsumes the relation "not equal".

Exercise 4.61

I had a strange problem here, with the rules written the way they are. My Lisp (both CLISP and SBCL) kept reporting stack overflows. What solved the problem was rewriting the rules with "next-to" coming before the variables, like this:

```
(qinterpret
  '(assert!
    (rule (next-to ?x ?y in (?x ?y . ?u)))))

(qinterpret
  '(assert!
    (rule (next-to ?x ?y in (?v . ?z))
          (next-to ?x ?y in ?z))))
```

I suspect this happens because of the rule indexing system that assumes that the first symbol in a rule conclusion is its name.

So, to answer the exercise:

```
(next-to ?x ?y in (1 (2 3) 4))
=>
(NEXT-TO (2 3) 4 IN (1 (2 3) 4))
(NEXT-TO 1 (2 3) IN (1 (2 3) 4))

(next-to ?x ?y in (2 1 3 1))
=>
(NEXT-TO 3 1 IN (2 1 3 1))
(NEXT-TO 2 1 IN (2 1 3 1))
(NEXT-TO 1 3 IN (2 1 3 1))
```

Exercise 4.62

Since we're allowed to assume that the list is not empty, the rules can be written as follows:

```
(qinterpret
  '(assert!
    (rule (last-pair (?elem) (?elem)))))

(qinterpret
  '(assert!
    (rule (last-pair (?v . ?u) (?l))
          (last-pair ?u (?l)))))
```

Tests:

```
(qinterpret
  '(last-pair (3) ?x))
=>
(LAST-PAIR (3) (3))

(qinterpret
  '(last-pair (1 2 3) ?x))
=>
(LAST-PAIR (1 2 3) (3))

(qinterpret
  '(last-pair (2 ?x) (3)))
=>
(LAST-PAIR (2 3) (3))
```

Given the query `(last-pair ?x (3))` my rule throws the interpreter into a stack overflow. I suppose the reason for this is the infinite amount of answers to such a rule.

Exercise 4.63

```
(qinterpret
  '(assert!
    (rule (grandson ?g ?s)
          (and
            (son ?g ?f)
            (son ?f ?s)))))

(qinterpret
  '(assert!
    (rule (son ?m ?s)
          (and
            (wife ?m ?w)
            (son ?w ?s)))))
```

And now we can ask:

```
(qinterpret
  '(grandson Cain ?s))
=>
(GRANDSON CAIN IRAD)

(qinterpret
  '(son Lamech ?s))
=>
(SON LAMECH JUBAL)
(SON LAMECH JABAL)

(qinterpret
  '(grandson Methushael ?s))
=>
(GRANDSON METHUSHAEL JUBAL)
(GRANDSON METHUSHAEL JABAL)
```

For comments, please send me an email.

Back to top