

Logic Programming

Abelson & Sussman sections 4.4, 4.4.1, 4.4.3

Just the language, not the implementation today.

- Introduction
- Programming with relations
- Primitives
- Means of Combination
- Means of Abstraction
- Logic Programming vs. Logic

Section 4.4 in general

Four goals for next two lectures:

1. Another language / methodology to **add to your set of tools**.
2. As **a language** has: primitives, means of combination, means of abstraction – that finally don't look like Scheme's
3. Under the hood: **Pattern matching and unification** Pattern matching – a good technique to know. If have time will look at pattern matching, unification algorithms.
4. Very different internals from previous evaluators, but the top level still looks like **mutual recursion between eval and apply**. `apply` deals with combinations. If a combination is made with an abstraction then `apply` must use `eval` to further evaluate with the abstraction.

1

2

Logic programming is declarative

Have a database of facts "a virtual world" and your then write queries "How can I show that the following statement is true about this world?".

Some "begats" in Genesis.

```
(define genesis-data-base
  '(
    (son-of adam abel)
    (son-of adam cain)
    (son-of cain enoch)
    (son-of enoch irad)
  ))
```

Who is Cain the son of?

```
;;; Query input:
(son-of ?x Cain)

;;; Query Results:
(son-of adam cain)
```

Match pattern and variable `?x` in database.

3

4

Who is the son of Cain

```
;;; Query input:  
(son-of Cain ?x)  
  
;;; Query Results:  
(son-of cain enoch)
```

Match pattern and variable **?x** in database.

Who is a son of Adam

```
;;; Query input:  
(son-of Adam ?x)  
  
;;; Query Results:  
(son-of adam cain)  
(son-of adam abel)
```

All matches. (Design choice vs. nondeterministic evaluator, which presents results one at a time)

5

6

What is the relationship (adam,cain)

```
;;; Query input:  
(?x Adam Cain)  
  
;;; Query Results:  
(son-of adam cain)
```

Most logic-programming languages are not designed to answer this.

Getting lots of relationships

Our database just contains “facts”.

Could also contain “rules”:

```
If (son-of ?x ?y) and (son-of ?y ?z) then  
(grandson-of ?x ?z)
```

Get new information by abstracting over existing relations.

7

8

Logic programming is about relations

in Scheme have

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

What is declarative content of append?

- For any Y: '() and Y **append to form** Y
- If CDR-X and Y **append to form** Z then
(CONS CAR-X CDR-X) and Y **append to form**
(CONS CAR-X Z)

Append as a relation

```
;;; Query input:
(append-to-form (1 2 3) (4 5 6) (1 2 3 4 5 6))
```

```
;;; Query results:
(append-to-form (1 2 3) (4 5 6) (1 2 3 4 5 6))
```

```
;;; Query input:
(append-to-form (1 2 3) (4 5 6) (1 2 3 4 5))
```

```
;;; Query results:
```

First append-to-form relation is correct.
Second is incorrect. Correct results printed out.

9

10

Append as a relation

```
;;; Query input:
(append-to-form (1 2 3) (4 5 6) ?x)

;;; Query results:
(append-to-form (1 2 3) (4 5 6) (1 2 3 4 5 6))

;;; Query input:
(append-to-form (1 2 3) (4 5 6) (1 2 . ?x))

;;; Query results:
(append-to-form (1 2 3) (4 5 6) (1 2 3 4 5 6))
```

Variable may represent all or part of a term.

11

Append as a relation

```
;;; Query input:
(append-to-form ?x ?y (1 2))

;;; Query results:
(append-to-form (1 2) () (1 2))
(append-to-form () (1 2) (1 2))
(append-to-form (1) (2) (1 2))
```

```
;;; Query input:
```

Results show all lists in append-to-form relationship with the third list.

12

Logic vs. functional programming

Scheme function `append` takes lists `x`, `y`, returns `z`.
Query language relation `append-to-form` determines the ways in which the relationship “`x` and `y` append to form `z`” can be satisfied.

Basics of logic programming:

- Logic is used to **express** what is true
- Logic is used to **check** what is true
- Logic is used to **determine** what is true.

13

The Query language

Start looking at primitives, combination, abstraction in the Query language in terms of a personnel database:

Microshaft a small Cambridge company employing the characters from the textbook.

```
(address (last-name first-name) (city street number))
(job (last-name first-name) (department position))
(salary (last-name first-name) number)
(supervisor (last-name-e first-name-e) (last-name-s first-name-s))
```

14

Running the Query Evaluator

Code from book at
mitpress.mit.edu/sicp/code/index.html

Modify to use www.cs.uml.edu/~dimock/courses/opl/Spring2005/sicp.ss to define book usages that are not standard Scheme.

In DrScheme: run modified `ch4-query.scm`
In interactions window:

```
> (load "microshaft-database")
(initialize-data-base microshaft-data-base)
(query-driver-loop)
```

15

Primitives

Primitive in the Query language is a query.

- `()` . match Scheme notation for pairs. (Like other evaluators, input is through the Scheme reader.)
- Any word not starting with `?` is a constant – except a few keywords – it must be matchable literally.
- Any word starting with `?` is a variable, it can match anything, up to constraints of Scheme notation.

16

Primitive queries

Get all supervisor relationships in database

```
;;; Query input:
(supervisor ?x ?y)

;;; Query results:
(supervisor (aull dewitt) (warbucks oliver))
(supervisor (cratchet robert) (scrooge eben))
(supervisor (scrooge eben) (warbucks oliver))
(supervisor (bitdiddle ben) (warbucks oliver))
(supervisor (reasoner louis) (hacker alyssa p))
(supervisor (tweakit lem e) (bitdiddle ben))
(supervisor (fect cy d) (bitdiddle ben))
(supervisor (hacker alyssa p) (bitdiddle ben))
```

17

Primitive queries

Who supervises his/her self?

```
;;; Query input:
(supervisor ?x ?x)

;;; Query results:
```

Who is a computer programmer?

```
;;; Query input:
(job ?x (computer programmer))

;;; Query results:
(job (fect cy d) (computer programmer))
(job (hacker alyssa p) (computer programmer))
```

18

Primitive queries

Who works in computer department?

```
;;; Query input:
(job ?x (computer ?type))

;;; Query results:
(job (tweakit lem e) (computer technician))
(job (fect cy d) (computer programmer))
(job (hacker alyssa p) (computer programmer))
(job (bitdiddle ben) (computer wizard))
```

19

Primitive queries

Missed one! remember Scheme pairs!

```
;;; Query input:
(job ?x (computer . ?type))

;;; Query results:
(job (reasoner louis) (computer programmer trainee))
(job (tweakit lem e) (computer technician))
(job (fect cy d) (computer programmer))
(job (hacker alyssa p) (computer programmer))
(job (bitdiddle ben) (computer wizard))
```

Pair allowing arbitrary cdr, as opposed to two-element list

20

Primitive queries

Who lives in Cambridge?

```
;;; Query input:
(address ?x (Cambridge . ?y))

;;; Query results:
(address (fect cy d) (cambridge (ames street) 3))
(address (hacker alyssa p) (cambridge (mass ave) 78))
```

21

Means of Combination

keyword and – two sub-queries must both be true.
Use same variable in different places to connect the queries. (Actually arbitrary number of subqueries)

```
;;; Query input:
(and (job ?x (computer . ?y))
      (supervisor ?x ?z))

;;; Query results:
(and (job (reasoner louis) (computer programmer trainee))
      (supervisor (reasoner louis) (hacker alyssa p)))
(and (job (tweakit lem e) (computer technician))
      (supervisor (tweakit lem e) (bitdiddle ben)))
(and (job (fect cy d) (computer programmer))
      (supervisor (fect cy d) (bitdiddle ben)))
(and (job (hacker alyssa p) (computer programmer))
      (supervisor (hacker alyssa p) (bitdiddle ben)))
(and (job (bitdiddle ben) (computer wizard))
      (supervisor (bitdiddle ben) (warbucks oliver)))
```

22

Means of Combination

keyword or – either of two subqueries may be true
(Actually arbitrary number of subqueries)

```
;;; Query input:
(or (supervisor ?x (Bitdiddle Ben))
     (supervisor ?x (Hacker Alyssa P)))

;;; Query results:
(or (supervisor (tweakit lem e) (bitdiddle ben))
     (supervisor (tweakit lem e) (hacker alyssa p)))
(or (supervisor (reasoner louis) (bitdiddle ben))
     (supervisor (reasoner louis) (hacker alyssa p)))
(or (supervisor (fect cy d) (bitdiddle ben))
     (supervisor (fect cy d) (hacker alyssa p)))
(or (supervisor (hacker alyssa p) (bitdiddle ben))
     (supervisor (hacker alyssa p) (hacker alyssa p)))
```

23

Means of Combination

```
;;; Query input:
(and (supervisor ?x (Bitdiddle Ben))
     (supervisor ?x (Hacker Alyssa P)))
```

```
;;; Query results:
```

No one supervised by both. How about:

```
;;; Query input:
(and (supervisor ?x (Bitdiddle Ben))
     (supervisor ?y (Hacker Alyssa P)))

;;; Query results:
(and (supervisor (tweakit lem e) (bitdiddle ben))
     (supervisor (reasoner louis) (hacker alyssa p)))
(and (supervisor (fect cy d) (bitdiddle ben))
     (supervisor (reasoner louis) (hacker alyssa p)))
(and (supervisor (hacker alyssa p) (bitdiddle ben))
     (supervisor (reasoner louis) (hacker alyssa p)))
```

24

Means of Combination

keyword **lisp-value** underlying system calculates.

```
(lisp-value predicate <exp1> ... <expn>)
```

Apply predicate to <exp1> ... <expn> and keep only values for which predicate is true. **Filter!**

Everyone who makes more than 30000

```
;;; Query input:
(and (salary ?person ?amount)
     (lisp-value > ?amount 30000))

;;; Query results:
(and (salary (scrooge eben) 75000) (lisp-value > 75000 30000))
(and (salary (warbucks oliver) 150000) (lisp-value > 150000 30000))
(and (salary (fect cy d) 35000) (lisp-value > 35000 30000))
(and (salary (hacker alyssa p) 40000) (lisp-value > 40000 30000))
(and (salary (bitdiddle ben) 60000) (lisp-value > 60000 30000))
```

25

Filtering vs. Logic

Can't filter unless have values for variables.

Another example of **generate and test**.

keyword **not** – a logical operation, but need to understand as filtering. **not** in the Query Evaluator is not quite the same as in logic: removes possible answers, does not generate them.

26

Means of Combination

Find everyone in computer department whose supervisor is not in computer department.

```
;;; Query input:
(and (job ?x (computer . ?y))
     (and (supervisor ?x ?z)
          (not (job ?z (computer . ?w)))))

;;; Query results:
(and (job (bitdiddle ben) (computer wizard))
     (and (supervisor (bitdiddle ben) (warbucks oliver))
          (not (job (warbucks oliver) (computer . ?w)))))
```

27

Means of Combination

This one doesn't tell us who supervisor is: variables **w** and **z** only appear inside **not**.

```
;;; Query input:
(and (job ?x (computer . ?y))
     (not (and (supervisor ?x ?z)
               (job ?z (computer . ?w)))))

;;; Query results:
(and (job (bitdiddle ben) (computer wizard))
     (not (and (supervisor (bitdiddle ben) ?z)
               (job ?z (computer . ?w)))))
```

28

Means of Combination

keyword **always-true** – sometimes useful in recursive query situations is the filter that doesn't remove anything.

29

Means of Abstraction

If someone works in a division but does not have a supervisor in the division, that means that the person is a "big-shot"

```
(rule
  (bigshot ?x ?dept)
  (and (job ?x (?dept . ?y))
        (not (and (supervisor ?x ?z)
                  (job ?z (?dept . ?w))))))
```

Usual idea of abstraction:

- pull out code for reuse.
- (optionally) name the code you are reusing.

The Query language always names its abstractions.

30

Means of Abstraction

```
(rule
  (bigshot ?x ?dept)
  (and (job ?x (?dept . ?y))
        (not (and (supervisor ?x ?z)
                  (job ?z (?dept . ?w))))))
```

Three parts:

```
(rule          ; keyword
  conclusion  ; looks like a fact but generally contains
              ; variables
  body)       ; a query -- primitive or combination,
              ; normally using variables in conclusion
```

Read: "the conclusion is true if the body is true."

31

Means of Abstraction

Rules normally kept in database. Keyword **assert!** allows you to add rules or facts to running system.

```
;;; Query input:
(assert!
  (rule
    (bigshot ?x ?dept)
    (and (job ?x (?dept . ?y))
          (not (and (supervisor ?x ?z)
                    (job ?z (?dept . ?w))))))

  (bigshot ?x ?d)

  (bigshot (scrooge eben) accounting)
  (bigshot (warbucks oliver) administration)
  (bigshot (bitdiddle ben) computer)
```

32

Means of Abstraction

Rules already in Microshaft database:

```
(rule (same ?x ?x))

(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
            (address ?person-2 (?town . ?rest-2))
            (not (same ?person-1 ?person-2))))
```

Note: rules can be used in other rules.

```
;;; Query input:
(lives-near (Hacker Alyssa P) ?who)

;;; Query results:
(lives-near (hacker alyssa p) (fect cy d))
```

33

Query Language Summary

- primitives – simple queries
- means-of-combination –
and or not lisp-value always-true
- means-of-abstraction – rules

Logic programming: Prolog is standard language,

- produces one answer at a time
- implemented using backtracking search like nondeterministic evaluator
- has lots of added features for math, strings, controlling the way that search works, controlling the way that matching works, allowing imperative programming...

34

Query Language Summary

The query language: simplification of Planner, Conniver languages at MIT in the '70s (Carl Hewitt).

- produces all answers
- uses streams internally
- only added feature is ability to use Scheme predicates

More to talk about: implementation:

- internal representation of a query.
- how combinations and abstraction work internally
- how pattern matching works.
- how unification works.

More to talk about: `not` vs. logical negation

35