Theme

- In this lecture we provide a simple logic programming language (from SICP), called a query language since it can be used to retrieve information - by lookup or deduction - from a database. In addition to studying the language, we will study an implementation of the language. The key concepts in the implementation of this language are pattern-matching, binding, and a generalization of pattern matching called unification that can be used for deduction.
Outline

- **Query Language**
  - Database of assertions
  - Simple queries
  - Compound queries
  - Rules

- **System architecture**
  - Pattern matching, Unification, Stream of frames

- **Implementing query language w/o rules**
  - Pattern matcher

- **Implementing rules**
  - Unifyer
Database of Assertions

(address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))
(job (Bitdiddle Ben) (computer wizard))
(salary (Bitdiddle Ben) 60000)

(address (Hacker Alyssa P) (Cambridge (Mass Ave) 78))
(job (Hacker Alyssa P) (computer programmer))
(salary (Hacker Alyssa P) 40000)
(supervisor (Hacker Alyssa P) (Bitdiddle Ben))

(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(job (Reasoner Louis) (computer programmer trainee))
(salary (Reasoner Louis) 30000)
(supervisor (Reasoner Louis) (Hacker Alyssa P))
Simple Queries

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

;;; Query results:
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (computer programmer))

;;; More queries
(job ?x (computer ?type))
(job ?x (computer . ?type))

(supervisor ?x ?x)
Processing Simple Queries

- The system finds all assignments to variables in the query pattern that satisfy the pattern -- that is, all sets of values for the variables such that if the pattern variables are instantiated with (replaced by) the values, the result is in the data base.

- The system responds to the query by listing all instantiations of the query pattern with the variable assignments that satisfy it.
Practice Question 1

- Enter queries for the following
  - all people supervised by Ben Bitdiddle
  - the names and jobs of all people in the accounting division
  - the names and addresses of all people who live in Slumerville
Compound Queries

;;; (and <arg1> … <argn>)
(and (job ?person (computer programmer)))
   (address ?person ?where))
;;; Query results:
(and (job (Hacker Alyssa P) (computer programmer)))
   (address (Hacker Alyssa P) (Cambridge (Mass Ave) 78)))
(and (job (Fect Cy D) (computer programmer)))
   (address (Fect Cy D) (Cambridge (Ames Street) 3)))
;;; another compound query – (or <arg1> … <argn>)
(or (supervisor ?x (Bitdiddle Ben))
   (supervisor ?x (Hacker Alyssa P)))
Compound Queries

;;; (not <arg>)
(and (supervisor ?x (Bitdiddle Ben))
   (not (job ?x (computer programmer)))))

;;; look out! – not as a filter
(not (job ?name (computer programmer))))

;;; (lisp-value <pred> <arg1>…<argn>)
(and (salary ?person ?amount)
   (lisp-value > ?amount 30000))
Practice Question 2

- Enter queries for the following
  - the names of all people who are supervised by Ben Bitdiddle, together with their addresses
  - all people whose salary is less than Ben Bitdiddle's, together with their salary and Ben Bitdiddle's salary
  - all people who are supervised by someone who is not in the computer division, together with the supervisor's name and job
Rules

;;; (rule <conclusion> <body>) – <conclusion> is pattern, <body> query

(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))))

(rule (same ?x ?x))

(lives-near ?x (Bitdiddle Ben))

;;; Query results: not facts in DB, but rather derived from facts

(lives-near (Reasoner Louis) (Bitdiddle Ben))
(lives-near (Aull DeWitt) (Bitdiddle Ben))
(and (job ?x (computer programmer)))
   (lives-near ?x (Bitdiddle Ben)))

(rule (outranked-by ?staff-person ?boss)
   (or (supervisor ?staff-person ?boss)
       (and (supervisor ?staff-person ?middle-manager)
            (outranked-by ?middle-manager ?boss)))))

;;; Query input:
(outranked-by (reasoner louis) (bitdiddle ben))

;;; Query results:
(outranked-by (reasoner louis) (bitdiddle ben))
Rules

;;; Query input:
(outranked-by (reasoner louis) ?name)

;;; Query results:
(outranked-by (reasoner louis) (hacker alyssa p))
(outranked-by (reasoner louis) (bitdiddle ben))
(outranked-by (reasoner louis) (warbucks oliver))
Rules

;;; Query input:
(outranked-by ?name (bitdiddle ben))

;;; Query results:
(outranked-by (tweakit lem e) (bitdiddle ben))
(outranked-by (reasoner louis) (bitdiddle ben))
(outranked-by (fect cy d) (bitdiddle ben))
(outranked-by (hacker alyssa p) (bitdiddle ben))
Logic as Programs

;;; (append-to-form L1 L2 L3) – L1 and L2 append to form L3
(rule (append-to-form () ?y ?y))
(rule (append-to-form (?u . ?v) ?y (?u . ?z))
    (append-to-form ?v ?y ?z))

;;; adding rules/facts to DB
;;; Query input:
(assert! (rule (append-to-form () ?y ?y)))

Assertion added to data base.

;;; Query input:
(assert! (rule (append-to-form (?u . ?v) ?y (?u . ?z))
    (append-to-form ?v ?y ?z)))

Assertion added to data base.
Logic as Programs (cont)

;;; Query input:
(append-to-form (a b) (c d) ?z)

;;; Query results:
(append-to-form (a b) (c d) (a b c d))

;;; Query input:
(append-to-form (a b) ?y (a b c d))

;;; Query results:
(append-to-form (a b) (c d) (a b c d))
Logic as Programs (cont)

;;; Query input:
(append-to-form ?x ?y (a b c d))

;;; Query results:
(append-to-form () (a b c d) (a b c d))
(append-to-form (a) (b c d) (a b c d))
(append-to-form (a b) (c d) (a b c d))
(append-to-form (a b c) (d) (a b c d))
(append-to-form (a b c d) () (a b c d))
Practice Question 3

Define a rule that says that person 1 can replace person 2 if either person 1 does the same job as person 2 or someone who does person 1's job can also do person 2's job, and if person 1 and person 2 are not the same person. Using your rule, give queries that find the following:

- all people who can replace Cy D. Fect
- all people who can replace someone who is being paid more than they are, together with the two salaries
For each frame match against entire database, producing a stream of extended frames

For a single query, start with single empty frame
Compound Queries

Input stream of frames

A

(and A B)

B

Output stream of frames, extended and filtered

Stream of assertions from database
Compound Queries

Input stream of frames -> A \( \lor A \land B \) \lor B -> Merge -> Output stream of frames, extended and filtered

Stream of assertions from database
Not as a Filter

Input stream of frames

\[ (\text{not } A) \]

Remove if satisfied

Output stream of frames, extended and filtered

Stream of assertions from database
Practice Question 4

Do the following two queries produce the same results? If not, why not.

(\text{and} \ (\text{supervisor} \ ?x \ ?y) \ \ (\text{not} \ (\text{job} \ ?x \ (\text{computer programmer}))))

(\text{and} \ (\text{not} \ (\text{job} \ ?x \ (\text{computer programmer})))) \ \ (\text{supervisor} \ ?x \ ?y))
Simple Queries & Pattern Matching

- Match query against assertions in database

- \((\text{pattern-match pat dat frame})\)
  - Attempt to match pattern and datum consistent with bindings in frame
  - Return extended frame with bindings implied by match

- Examples
  - \(\text{dat} = ((a \ b) \ c \ (a \ b)), \text{pat} = ((? \ x) \ c \ (? \ x)) \Rightarrow x = (a \ b)\)
  - \(\text{dat} = ((a \ b) \ c \ (a \ b)), \text{pat} = ((? \ x) \ c \ (? \ y)) \Rightarrow x = (a \ b), y = (a \ b)\)
  - \(\text{dat} = ((a \ b) \ c \ (a \ b)), \text{pat} = ((? \ x) \ b \ (? \ x)) \Rightarrow \text{failed}\)
Pattern Matcher

(define (pattern-match pat dat frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? pat dat) frame)
        ((var? pat) (extend-if-consistent pat dat frame))
        ((and (pair? pat) (pair? dat))
         (pattern-match (cdr pat) (cdr dat) (pattern-match (car pat) (car dat) frame)))
        (else 'failed)))
Pattern Matcher (cont)

(define (extend-if-consistent var dat frame)
  (let ((binding (binding-in-frame var frame)))
    (if binding
      (pattern-match (binding-value binding) dat frame)
      (extend var dat frame))))

- binding may be a pattern if stored during unification
  - E.G. ?x bound to (f ?y)
  - Augment with ?x bound to (f b)
  - (pattern-match (f ?y) (f b)) ⇒ ?y bound to b
Streamlining Pattern Matching

- Speed up pattern matching by limiting the database entries that must be matched
  - If car of pattern is constant, match against stream of patterns starting with same constant

```
(define THE-ASSERTIONS the-empty-stream)
(define (fetch-assertions pattern frame)
  (if (use-index? pattern)
      (get-indexed-assertions pattern)
      (get-all-assertions)))
(define (get-indexed-assertions pattern)
  (get-stream (index-key-of pattern) 'assertion-stream))
```
Implementing Query Evaluation

(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
    (if qproc
      (qproc (contents query) frame-stream)
      (simple-query query frame-stream))))

(define (simple-query query-pattern frame-stream)
  (stream-flatmap
    (lambda (frame)
      (stream-append-delayed
        (find-assertions query-pattern frame)
        (delay (apply-rules query-pattern frame)))))
    frame-stream))

(define (find-assertions pattern frame)
  (stream-flatmap (lambda (datum)
    (check-an-assertion datum pattern frame))
    (fetch-assertions pattern frame)))
Implementing Compound Queries

(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (conjoin (rest-conjuncts conjuncts)
        (qeval (first-conjunct conjuncts)
          frame-stream))))

(define (negate operands frame-stream)
  (stream-flatmap
    (lambda (frame)
      (if (stream-null? (qeval (negated-query operands)
                  (singleton-stream frame)))
        (singleton-stream frame)
        the-empty-stream))
    frame-stream))

(put 'and 'qeval conjoin)  (put 'not 'qeval negate)
Applying Rules

- 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.

- (lives-near ?x (Hacker Alyssa P))
- (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)))))
Unification

- Similar to pattern matching except both the pattern and datum can have variables.
- Unifier takes two patterns and determines if it is possible to assign values to variables that makes the two patterns equal. Return most general match

  - $(?x \ a \ ?y)$ and $(?y \ ?z \ a) \Rightarrow x = y = z = a$
  - $(?x \ ?y \ a)$ and $(?x \ b \ ?y) \Rightarrow \text{fail}$
  - $(?x \ ?x)$ and $((a \ ?y \ c) \ (a \ b \ ?z)) \Rightarrow y = b, z = c, x = (a \ b \ c)$
  - $(?x \ a)$ and $((b \ ?y) \ ?z) \Rightarrow x = (b \ ?y), z = a$
  - $(?x \ ?x) \ (?y \ (a \ ?y)) \Rightarrow \text{recursive dependancy}$
Practice Question 5

What happens when the following query is matched against the following rule?

- `(lives-near ?x (Hacker Alyssa P))`

- `(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))))`
Practice Question 6

What happens when you enter the following query?

(assert! (married Minnie Mickey))
(assert! (rule (married ?x ?y)
             (married ?y ?x)))
(married Mickey ?who)
(define (unify-match p1 p2 frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? p1 p2) frame)
        ((var? p1) (extend-if-possible p1 p2 frame))
        ((var? p2) (extend-if-possible p2 p1 frame)) ; ***
        ((and (pair? p1) (pair? p2))
         (unify-match (cdr p1)
                      (cdr p2)
                      (unify-match (car p1)
                                   (car p2)
                                   frame))))
  (else 'failed)))
Unifier (cont)

(define (extend-if-possible var val frame)
  (let ((binding (binding-in-frame var frame)))
    (cond (binding
           (unify-match
            (binding-value binding) val frame))
           ((var? val) ; ***
            (let ((binding (binding-in-frame val frame)))
             (if binding
                 (unify-match
                  var (binding-value binding) frame)
                 (extend var val frame)))))
           ((depends-on? val var frame) ; ***
            'failed)
           (else (extend var val frame))))))
(define (depends-on? exp var frame)
  (define (tree-walk e)
    (cond ((var? e)
       (if (equal? var e)
           true
           (let ((b (binding-in-frame e frame)))
             (if b
                 (tree-walk (binding-value b))
                 false))))
     ((pair? e)
      (or (tree-walk (car e))
       (tree-walk (cdr e)))
     (else false)))
    (tree-walk exp))
Implementing Rule Evaluation

(define (apply-a-rule rule query-pattern query-frame)
  (let ((clean-rule (rename-variables-in rule)))
    (let ((unify-result
              (unify-match query-pattern
                (conclusion clean-rule)
                query-frame)))
      (if (eq? unify-result 'failed)
          the-empty-stream
          (qeval (rule-body clean-rule)
                (singleton-stream unify-result))))))
Practice Question 7

- Study the code for the unifier and try several examples (you might trace the examples from slide 31)