

Matching patter under constraints: Pattern matcher takes frame containing some (or no) constraints and produces an output frame with possibly more constraints, or fails.

Frame to pattern matcher containing:
Input constraint on variables: `x = a`

```
(? x ?y ?y ?x) match (a b b a)
```

Frame from pattern matcher contains what
Output constraints on variables?

Frame to pattern matcher containing:
Input constraint on variables: `y = a`

```
(? x ?y ?y ?x) match (a b b a)
```

Frame from pattern matcher contains what
Output constraints on variables?

3

- Do some pattern matching
- Look at pattern matcher code
- Overview: Stream of streams implementation
- Do some unification
- Eval / Apply
- Look at unifier code

1

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)))

(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))))

; ADT for binding: variable . value
(define (binding-variable binding)
  (car binding))
(define (binding-value binding)
  (cdr binding))

; Frame is list of bindings
(define (binding-in-frame variable frame)
  (assoc variable frame))
(define (extend variable value frame)
  (cons (make-binding variable value) frame))
```

4

Pattern Matching

Somewhere in evaluator there is a pattern matcher.

```
(a ?x c)

(job ?x (computer ?y))

(job ?x (computer . ?y))

(a ?x ?x)

(?x ?y ?x ?y)

(a ?x)
```

2

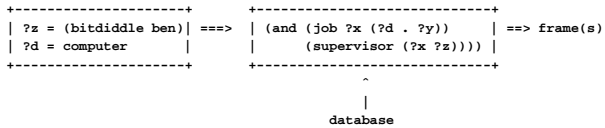
Abstractions

```
(rule
  (boss ?z ?d)
  (and (job ?x (?d . ?y))
        (supervisor (?x ?z))))
```

To process query

```
(boss (bitdiddle ben) computer)
```

1. Create frame(s) from matching conclusion of rule
2. Use them as stream of frames to query which is rule body



technical points

- (boss ?who computer) has ?z = ?who Now have patterns in frame.
No longer pattern matching: unification.
- (boss ?y computer) the ?y is not the same as ?y in body of rule!
Whenever processing a query rename variables to unique names.

eval and apply

Query language:

To apply a rule

Evaluate the **rule** body relative to the environment formed by **unifying the rule conclusion with the given query**.

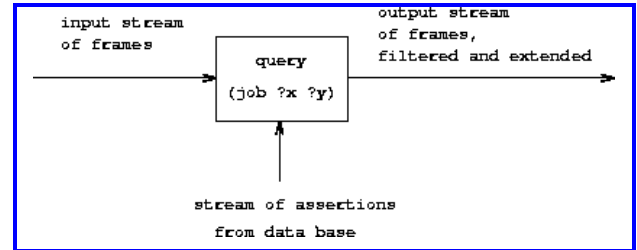
Scheme:

To apply a procedure

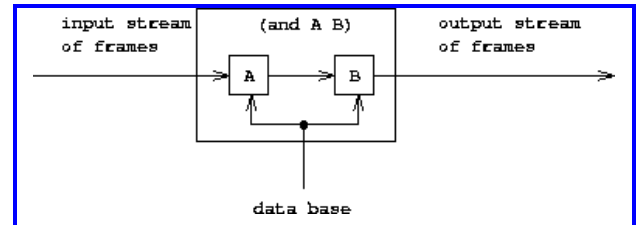
Evaluate the **procedure** body relative to the environment formed by **binding the procedure parameters to the arguments**.

High level design

Any query takes a stream of database elements and a stream of frames and produces a stream of frames.



Compound queries use same structure with fancier contents in the query box:



More compound queries (combinations)

Query language conclusion

Logic programming: useful for rule based systems.
Include in your set of tools.

Remember logic programming is not exactly logic:
need to understand how language works, not just how
logic works.

Implementation: still uses eval / apply mutual recursion
to support using abstractions.

Unification / pattern matching: know them.

Unification

Match portions of term with variables in both
directions:

```
unify  (?x ?x)

with  ((a ?y c) (a b ?z))

?x = ...
?y = ...
?z = ...
```

How about

```
unify  (?x ?x)

with  ((?y a ?w) (b ?v ?z))

?y = ...
?v = ...
?w = ...
?x = ...
```

created frame contains a variable!

11

9

Unification algorithm

```
(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)))

(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) ; {\em ; ***}
           (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) ; {\em ; ***}
           'failed)
          (else (extend var val frame)))))
```

10