Pattern Matching

Wilensky Chapter 21

Pattern Matching

- ◊ A ubiquitous function in intelligence is pattern matching
  - » **IQ tests, for example, contain pattern matching problems because they are recognized as an important class of problem that people deal with.**
- ◊ Pattern matching means to compare one object with another object and recognize if they are similar
  - » **Basic case is comparing constants**
  - » **More interesting is to compare parameterized patterns**
    - > **A is like B except for ....**
    - > **A is like B where ...**
      - – **a statement that subobjects, while not identical, correspond to each other**

What is a pattern?

◊ In Lisp, a pattern is a form (S-expression) that contains

- » constants – called literals
- » pattern matching variables

◊ We need a syntax to differentiate the two

- » Can prefix pattern matching variables with ?
  - > for example ?x ?abc

◊ An abstract pattern could look like

- » (a b ?x c ?y)

◊ A more meaningful pattern could be

- » (causes (hit ?x ?y) (hurt ?y))
  - > Interpreted as – x hitting y, causes y to be hurt

Pattern variable representation

- How will we represent pattern matching variables in Lisp? – the rest is simply a list with symbols for the constants

  - **Use the construct (*VAR* X)**

    - **where *VAR* is a special symbol we recognize within the matcher program**

When do two patterns match?

- Two patterns can be matched when it is possible to **unify** them
- **Unification** means an assignment can be made to the variables in each pattern such that the patterns become identical.
  - **We usually mean the most general possible assignment.**
- An assignment is shown by the pair (variable value)
  - **((*VAR* X) abc)**
  - **((*VAR* X) (*VAR* Y))**

© Gunnar Gotshalks

Unification Examples – 1

- (a ?x b)
  (a y b)
  match if ?x <-- y
  we say that ?x is bound to y

- (a ?x b)
  (a ?y b)
  match if ?x <-- ?y

- (a ?x (b ?z))
  (a (((e))) ?y)
  match if ?x <-- (((e)))
  ?y <-- (b ?z)

Unification Examples – 2

◊ More complex examples

» (a ?x ?x) match if ?x = ?y
(a ?y c) and ?y = c

> **Cannot naively bind ?x to ?y and then ?x to c as then we are trying to assign two different values to ?x need to substitute ?y for ?x and then see that ?y binds to c**

» (a ?x ?x ?x)
(a ?y ?y ?y)

> **Cannot naively try to bind ?x to ?y , as on the second attempt, we end up binding ?y to ?y , then on the third attempt, we have an infinite loop**

Unification Examples – 3

- More complex examples

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

    There is no consistent binding to make a match

    - **Again need to prevent an infinite loop**

Pattern variable input

- How do we represent input?
  - We would like to keep the notation ?x
  - Instruct the read program to recognize the construct ?symbol and create the list (*VAR* symbol)

```
(set-macro-character  #\?       ;See page 245
  #’( lambda  ( stream  char )
       ( list  '*var*  ( read  stream  t  nil  t) )))
```

  - Test with (read), enter ?x and see (*VAR* x) as the result

Pattern matcher output

- Need to distinguish three cases (see p369 for a discussion)
  - **No match is possible**
    - **output is nil**
  - **Match is possible but no variable bindings are required**
    - **output is T ; nil – two values returned**
  - **Match is possible with variable bindings**
    - **output is T ; (list of bindings)**
    - **a binding is a pair ((*VAR* variable) value)**
- Example with a binding required
  - **(match '(a ?x c ?y e) '(a b ?z d e))**
    - **T ; (((*VAR* Y) D) ((*VAR* Z) C) ((*VAR* X) B))**

Matcher

- Reminder that we need to define the macro characer ?

```
(set-macro-character  #\?
          #’( lambda  ( stream  char )
               ( list  '*var*  ( read  stream  t  nil  t) )))
```

- The entry function creates the initial empty binding

```
(defun  match  ( pattern1  pattern2 )
 (match-with-bindings  pattern1  pattern2  nil ) )
```

Matching cases – 1

- ◊ Matching two patterns requires a recursive descent into the patterns to match sub-patterns the following cases can occur
  - » **Pattern1 – a variable, an atom, a list**
  - » **Pattern2 – a variable, an atom, a list**

Matching cases - 2

◊ The matching program has to examine the possible combinations

Pattern1	Pattern2	Result
atom	atom	match if equal, else no match
atom	variable	try to bind atom to variable
atom	list	no match
variable	atom	try to bind atom to variable
variable	variable	try to bind variable to variable
variable	list	try to bind list to variable
list	atom	no match
list	variable	try to bind list to variable
list	list	recursive descent on first and rest

Match with bindings – 1

◊ Organize when bindings need to be done

```
(defun match-with-bindings (pattern1 pattern2 bindings)
 (cond

 > Pattern 1 is a variable?

       ( ( pattern-var-p pattern1 )
           ( variable-match  pattern1  pattern2  bindings) )

 > Pattern 2 is a variable?

        ( ( pattern-var-p  pattern2 )
           ( variable-match  pattern2  pattern1  bindings ) )

 > Pattern 1 is an atom? Note use of values

       ( ( atom  pattern1 )
          ( if  ( eq  pattern1  pattern2 )  ( values  t  bindings)))

 > Pattern 2 is an atom?

       ( ( atom  pattern2 ) nil )
```

Match with bindings – 2

> Pattern1 and Pattern2 are both lists – use recursion and multiple values

```
     ( t
       (multiple-value-bind  ( flag  carbindings )
         (match-with-bindings       ( car  pattern1 )
                                    ( car  pattern2 )
                                    bindings )
         (and flag
             (match-with-bindings       ( cdr  pattern1 )
                                        ( cdr  pattern2 )
                                       carbindings )
)))))
```

Variable match

◊ Find a binding for **pattern-var** within **item** using the current **bindings**

```
(defun variable-match (pattern-var item bindings)
```

> **Check for equality – no additional bindings are necessary**

```
(if (equal pattern-var item) (values t bindings)
```

> **Otherwise ...**

© Gunnar Gotshalks

Variable match – 2

◊ Need a binding

```
(let ((var-binding   ;;; determine if a binding already exits
         (get-binding pattern-var bindings)))
```

- Handle the case where a binding exists

```
      (cond (var-binding
              (match-with-bindings var-binding item bindings))
```

- No binding for the variable – check for circularity – need to see if the pattern-var occurs in item or is bound to a variable in item.

```
              ((not (contained-in pattern-var item bindings))
                 (values t
                         (add-binding pattern-var item bindings)))
))))
```

Contained in – 1

◊ Check for circularity by – seeing if **pattern-var** occurs in item or is defined as the value of a binding of a variable in **item**

```
(defun contained-in (pattern-var item bindings)
  > Cannot be contained in an atom
  (cond ((atom item) nil)
  > Check if item is a variable
        ((pattern-var-p item)
  > Does pattern-var occur in item
         (or (equal pattern-var item)
  > Does pattern-var occur as the value of a binding?
              (contained-in pattern-var
                            (get-binding item bindings)
                            bindings)))
```

Contained in – 2

- The item is a list so recursively check for contained in

```
(t
   (or (contained-in  pattern-var  (car item)
                                    bindings)
         (contained-in  pattern-var  (cdr item)
                        bindings)
))))
```

Matcher – Housekeeping functions

- Add the binding to the current **bindings** (a list of 2 element lists)

```
(defun  add-binding  ( pattern-var  item  bindings )
  ( cons  ( list  pattern-var  item )  bindings ))
```

- If **item** is a pattern variable return true, else return false

```
(defun  pattern-var-p  ( item )
  ( and  ( listp  item )  ( eq  '*var*  ( car  item ))))
```

- Get the binding, if any, for **pattern-var** in the binding list **bindings**

```
(defun  get-binding  ( pattern-var  bindings )
  ( cadr  ( assoc pattern-var bindings :test #'equal)))
```