Chapter 8

CLIPS Pattern Matching
Course Outline

• Introduction to Artificial Intelligence
• Introduction to Expert Systems
• Representation of Knowledge
• Methods of Inference
• CLIPS
• Reasoning under Uncertainty
• Design of Expert Systems
• Knowledge Acquisition
Content

- Variables
- Multiple use of variables
- Fact address
- Single-field wild cards
- Blocks world
- Multifield wildcards and variables
- Field constraints
- Functions and expressions
Variables

Variable: ?speed

No blank between? And speed

To bind values to slot name

e.g. IF x is a grandfather THEN x is man

(defrule grandfather
  ( is-a-grandfather ?x )
  =>
  (assert (is-a-man ?x ))
)

> (watch facts)
> (assert (is-a-grandfather John))
> f-1 (is-a-grandfather John)
<Fact-1>
> (run)
> f-2 (is-a-man John)
>
Binding slot value John to slot name is-a-grandfather and is-a-man
(defrule grandfather
  (is-a-grandfather ?name)
  =>
  (assert ( is-a-father ?name)
           (is-a-man    ?name)
  )
  (printout t ?name "is grandfather" crlf)
)

terminal                              next line
Multiple use of variables

• Facts match LHS condition in order
  (defrule find-eyes
    (find (eyes ?eyes))
    (person (name ?name) (eyes ?eyes))
  =>
    (printout t ?name " has " ?eyes " eyes. " crlf))

• (assert (person (name Jane)(eyes blue)(hair red)))
• (assert (find (eyes blue)));

• (run) ;?eyes is bound to blue; ?name is bound to Jane
  – Jane has blue eyes.
Fact Addresses

• A fact is bound to a fact address for retract, modification, duplicates of facts on the LHS of rule

• Pattern binding operator <-

  • (defrule modify-grandfather-fac
  •   ?old <- ( is-a-grandfather ?name) ;Keep fact in fact address ?old
  •   =>$
  •   (retract ?old) ;delete content at this address
  •   (assert ( is-a-father ?name)
  •   (is-a-man ?name) )
  • )
(defrule simple-loop
  ?old-fact <- (loop-fact)
=>
  (printout t "Looping" crlf)
  (retract ?old-fact) (assert (loop-fact))
)

Let the rule be put into AGENDA again and be fired repeatedly

AGENDA

Infinite loop
Single-field wildcards “?”

- Used when a field is required, but the value is not important
  - Notation: “?”

(defrule print-ssn
  (print-ssn-for ?last-name)
  (person (name ? ? ?last-name) (ssn ?ss-number))
  =>
  (printout t ?ss-number crlf) )

>(assert (person (name Joe Q. Public)(ssn 123-456-7890))
>(assert (print-ssn-for Public)
>(run)
> 123-456-7890
Building a program

• Analysis
  – algorithm
  – Pseudo rules

• Design
  – Facts, define templates
  – Initial states, goal states

• Implementation
  – Coding in CLIPS
Building a program

1. Fact format is user-defined, which should be as easy to read as possible.
   
   John is a employee of IBM.
   He is 24 years old.
   
   (employee-of-IBM John 24 male)
   Or
   (John 24 male IBM)

2. Rule format should match fact format.
   
   (defrule find-employee-of-IBM
       (employee-of-IBM ?name ?age ?sex)
     =>
       (printout t ?name ?age ?sex)
   )
Doctor: The usual features of catching cold are
Sneeze, cough, headache, and feel tired.

Knowledge Acquisition

Rule:
If sneeze & cough & headache & feel-tired
THEN
Catch cold = TRUE

(1) Deductive error
(2) Ambiguity
(3) Variable binding?
• **Goal**: Move block C on top of block E

• **Analysis**: (four cases)
  – C is top block and E is top block
    • Rule MOVE-DIRECTLY
  – C is not top block
    • Rule CLEAR-UPPER-BLOCK
  – E is not top block
    • Rule CLEAR-LOWER-BLOCK
  – Rule: MOVE-TO-FLOOR
Blocks World

• C is top block and E is top block
  – Rule MOVE-DIRECTLY
  – If the goal is to move block ?upper on top of block ?lower and
  – block ?upper is the top block in its stack and
  – block ?lower is the top block in its stack
  – Then Move block ?upper on top of block ?lower
Blocks World

- C is not top block
  - Rule CLEAR-UPPER-BLOCK
    - If the goal is to move block ?x and
    - block ?x is the not the top block in its stack and
    - block ?above is on top of block ?x
    - Then the goal is to move block ?above to the floor
Blocks World

• E is not top block
  – Rule CLEAR-LOWER-BLOCK
  – If the goal is to move another block on top of block ?x and
  – block ?x is the not the top block in its stack and
  – block ?above is on top of block ?x
  – Then the goal is to move block ?above to the floor
Blocks World

• Rule Rule: MOVE-TO-FLOOR
  – If the goal is to move block ?upper on the top of floor and
  – block ?upper is the top block in its stack
  – Then Move block ?upper on top of the floor
**Blocks World**

- Design
- Fact templates:
  1. (block <name>) ; (factname value)
     
     e.g. (block A), (block B)…
  2. (deftemplate on-top-of (slot upper)(slot lower))
     
     e.g. (on-top-of (upper A)(lower B))
     (on-top-of (upper C)(lower floor))
     (on-top-of (upper nothing)(lower D))
Blocks World

• Design
• 3. Goal template:
  – Initial goal: (goal (move C) (on-top-of E))
    (deftemplate goal (slot move)(slot on-top-of))
• 4. Initial state
  (defacts initial-state
  (block A)(block B) (block C)(block D) (block E)(block F)
  (on-top-of (upper nothing)(lower A))
  (on-top-of (upper A)(lower B))…
  (goal (move C) (on-top-of E)))
Blocks World

- Implementation
  - Define rules in CLIPS
1. Rule Move-Directly

(Defrule move-directly
  ? goal <- (goal (move ?block1)(on-top-of ?block 2))
    (block ?block1)
    (block ?block2)
    (on-top-of (upper nothing)(lower ?block1))
  ?stack-1<- (on-top-of (upper ?block1) (lower ?block3))
  ?stack-2<- (on-top-of (upper nothing)(lower ?block2))

  =>

  (retract ?goal ?stack-1 ?stack-2) ;delete 3 facts above
  (assert (on-top-of (upper ?block1)(lower ?block2))
        (on-top-of (upper nothing)(lower ?block3)))
  (printout t ?block1 " moved on top of" ?block2 "." crlf))
2. Move-to-floor

3. Clear-upper-block
   (defrule clear-upper-block
     (goal (move ?block1))
     (block ?block1)
     (on-top-of (upper ?block2) (lower ?block1))
     (block ?block2)
   =>
     (assert (goal (move ?block2) (on-top-of floor))))

4. Clear-lower-block

>(Reset)
>(Run)

   A moved on top of floor.
   B moved on top of floor.
   D moved on top of floor.
   C moved on top of E.
Multiple Field Wildcards and Variables

• To match against zero or more fields of a pattern; “$?”

(defrule find-list
  (list $?) ; $? multifield wildcard
  =>
  (printout t “found list” crlf)
)

(list a) matched!
(list b) matched!
(list a b) matched!
(list a b c d) matched!
Multifield variable

(defrule find-list
  (list $?item) ; multifield variable
  =>
  (printout t "Found a list:" ?item crlf)
)

(defrule find-item-c
  (list $?before c $?after)
  =>
  (printout t "item before c: " ?before crlf)
  (printout t "item after c: " ?after crlf)
)
(list a b c)
- before: (a b)
- after: ( )

(list a c c e)
- (1) before: (a c) after: (e)
- (2) before: (a) after: (c e)

(list c d b)
- before: ?
- after: ?
(defrule find-item-c
  (list $?before c $?after)
  =>
  (printout t "before:" ?before crlf)
  (printout t "after:" ?after crlf))

(assert (list a c c b)) ;fact f-1

(agenda)

  0:find-item-C:f-1
  0:find-item-C:f-1

(RUN)

  before : (a)
  after : (c b)
  before : (a c)
  after : (b)
(defrule combin-match
  (list ? ? $? C ?)
  =>
  (printout t "found!!" crlf)
)

(list c) unmatched!
(list c d) unmatched!
(list a c e) unmatched!
(list a c b d) unmatched!
(list a d c b) unmatched!

? $? at least one
$? zero or more
? exactly one
Implementing a Stack

```
(defrule push-value
  ?push-value <- (push ?value) ;pushing a value
  ?stack <- (stack $?rest) ;a stack is an order fact list
  =>
  (retract ?push-value ?stack) ;retracts
  (assert (stack ?value ?rest)) ;push a value onto the stack fact
)
```

(assert (stack)) ←
(assert (push a)) ←
(run) ←
(facts) ←
(stack a)
(assert (push b)) ←
(run) ←
(facts) ←
(stack b a)

match (stack)
(push a)
match (stack a)
(push b)
Implementing a Stack_2

(defrule pop-value-valid ;rule for popping from non-empty stack
  ?pop-value <- (pop-value) ;popping a value
  ?stack <- (stack ?value $?rest)
=>
  (retract ?pop-value ?stack) ;retracts
  (assert (stack $?rest))
  (printout t "Popping value " ?value crlf))

(defrule pop-value-invalid ;rule for popping from empty stack
  ?pop-value <- (pop-value)
  (stack)
=>
  (retract ?pop-value) ;retracts
  (assert (stack $?rest))
  (printout t "Popping from empty stack\n" crlf))
Field constraints

The ‘NOT’ field constraint
(defrule person-not-brown-hair
  (person ?name ?~brown)
  => Not brown
  (printout t ?name crlf) don’t care)

eyes  hair
  ↓    ↓
(person John black black) matched!
(person Mary black brown) unmatched!

The ‘OR’ field constraint
(defrule black-or-brown-hair
  (person ?name ?brown | black)
  =>
  (print t ?name crlf) OR
The ‘**AND**’ field constraint

(defrule black-or-brown-hair
    (person ?name ? ?color & brown | black)
    =>
    (printout t ?name "has" ?color "hair" crlf))

(defrule black-and-brown-hair
    (person ?name ? black&brown)
    =>
    (printout t ?name crlf)

The condition will never be matched.

a useless rule
(defrule black-or-brown hair
  (person ?name ?
    ?color&~brown&~black)
  =>
  (printout t ?name "has"
    ?color "hair" crlf)
)

only the color which is neither brown nor black
    can be matched
Functions and Expressions

Math function

+ , -

*, /

**

numeric expression in prefix form

no built-in precedence of arithmetic operations

CLIPS>(+ 2 2)

4

CLIPS>(+ 2 (* 3 5))

17

CLIPS>(assert (answer (+ 2 2 )))

(answer 4)  ‘+’ is treated as an operator
Summing values using rules

Infinite loop!! Single rect fact with different sum fact
One way to resolve this is:

```
(defrule sum-rect
    ?x <- (rect ?h ?w)
    ?sum <- (sum ?total)
    =>
    (retract ?x ?sum)
    (assert (sum (+ ?total (* ?h ?w))))
)
```

(rect 10 6)
(rect 7 5)
(rect 6 8)
(rect 2 5)
(sum 0)

(RUN)

(sum 153)

Not a good idea
Since the facts (rect ? ?)
Should remain the same
An alternative approach:

(defrule sum-rect
  (rect ?h ?w)
  =>
  (assert (area (* ?h ?w))))

(defrule sum-area
  ?sum <- (sum ?total)
  ?new-area <- (area ?area)
  =>
  (retract ?sum ?new-area)
  (assert (sum (+ ?total ?area))))
Example: 3 scores

(defrule move-directly
  ?good <- (move-good ?block1
     on top-of ?block2)
  (block ?block1)
  (block ?block2)
  (on-top-of nothing ?block1)
  ?stack-1 <- (on-top-of ?block1 ?block3)
  ?stack-2 <- (on-top-of nothing ?block2)
=> ?)
8.11 The Bind Function

(bind <variable> <value>)

binding <value> to <variable>

8.12 Using Function on The RHS

Commands at RHS of a rule

the top-level prompt

eg.

(defrule print-facts
 =>
 (facts))