CLIPS Programming 2

Advanced Pattern Matching

Variables

- Variables in CLIPS are always written in a question mark followed by a symbolic field name.
- Some examples of variables:

```
?speed
?sensor
?value
?noun
?color
```

- There should be no space between the question mark and the symbolic field name

```
CLIPS>
(defrule find-blue-eyes
    (person (name ?name)(eyes blue))
    =>
    (printout t ?name " has blue eyes." crlf)
```

Fact Address

- A variable can be bound to the fact address of the fact matching a pattern on the LHS of a rule by using the pattern binding operator, "<-".
  - Once the variable is bound it can be used with the retract, modify, or duplicate commands in place of a fact index.

Fact Address

```
CLIPS> (clear)
CLIPS>
   (deftemplate person
        (slot name)
        (slot address))
CLIPS>
   (deftemplate moved
        (slot name)
        (slot address))
CLIPS>
   (defrule process-moved-information
        ?f1 <- (moved (name ?name)
                (address ?address))
        ?f2 <- (person (name ?name))
=>
   (retract ?f1)
   (modify ?f2 (address ?address)))
```

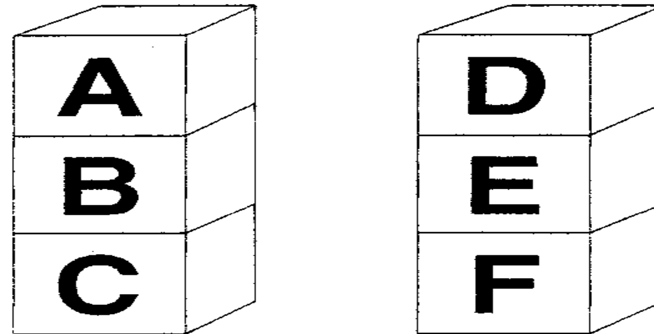
Fact Address

```
CLIPS>
   (deffacts example
        (person (name "John Hill")
                (address "25 Mulberry Lane"))
        (moved (name "John Hill")
                (address "37 Cherry Lane")))
CLIPS>(reset)
CLIPS>(watch rules)
CLIPS>(watch facts)
CLIPS>(run)
FIRE   1 process-moved-information: f-2,f-1
<== f-2       (moved (name "John Hill")
                     (address "37 Cherry Lane"))
<== f-1       (person(name "John Hill")
                     (address "25 Mulberry Lane"))
==> f-3       (person(name "John Hill")
                     (address "37 Cherry Lane"))
CLIPS>
```

Single-Field Wildcards

- A **single-field wildcard** can be used when a field is required, but the value is not important.
- A single-field wildcard is represented by a question mark.
- E.g.
  - (person (name ? ? ?last-name))
- When a single-field slot is left unspecified in a pattern, CLIPS treats it as if there is a single-field wildcard check for that slot.
  - E.g. (person (name ?first ?last))
  - Is equivalent to:
    - (person (name ?first ?last) (social-security-number ?))

Blocks World




```
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

```
RULE CLEAR-UPPER-BLOCK
IF   The goal is to move block ?x and
     block ?x is 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

RULE CLEAR-LOWER-BLOCK
IF   The goal is to move another block on top of
       block ?x and
     block ?x is 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 MOVE-TO-FLOOR
  IF The goal is to move block ?upper on top of the floor and
  block ?upper is the top block in its stack,
  THEN Move block ?upper on top of the floor.

Blocks World

- Types of facts needed:
  - The information about which blocks are on top of other blocks is crucial. This information could be described with the following deftemplate:

```
(deftemplate on-top-of
 (slot upper)
 (slot lower))
```

  - The facts identifying the blocks from the special words nothing and floor.
    - E.g. (block A)
  - A fact is needed to describe the block-moving goals.

```
(deftemplate goal (slot move) (slot on-top-of))
```

Blocks World

- The initial configuration of the blocks world:

```
(deffacts 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))
(on-top-of (upper B) (lower C))
(on-top-of (upper C) (lower floor))
(on-top-of (upper nothing) (lower D))
(on-top-of (upper D) (lower E))
(on-top-of (upper E) (lower F))
(on-top-of (upper F) (lower floor))
(goal (move C) (on-top-of E)))
```

Blocks World

- The move-directly rule:

```
(defrule move-directly
  ?goal <- (goal (move ?block1)
                 (on-top-of ?block2))
  (block ?block1)
  (block ?block2)
  (on-top-of (upper nothing) (lower ?block1))
  ?stack-l <- (on-top-of (upper ?block1)
  (lower ?block3))
  ?stack-2 <- (on-top-of (upper nothing)
   (lower ?block2))
=>
  (retract ?goal ?stack-l ?stack-2)
  (assert (on-top-of (upper ?blockl) (lower ?block2))
          (on-top-of (upper nothing) (lower ?block3)))
  (printout t ?block1 " moved on top of" ?block2
                ". " crlf))
```

Blocks World

- The move-to-floor rule:

```
(defrule move-to-floor
  ?goal <- (goal (move ?blockl)
                 (on-top-of floor))
  (block ?blockl)
  (on-top-of (upper nothing) (lower ?blockl))
  ?stack <- (on-top-of(upper ?blockl)
                      (lower ?block2))
=>
  (retract ?goal ?stack)
  (assert (on-top-of (upper ?blockl)
                      (lower floor))
          (on-top-of (upper nothing)
                      (lower ?block2)))
  (printout t ?blockl " moved on top of floor.” crlf) )
```

Blocks World

- The clear-upper-block rule

```
(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))))
```

Blocks World

- The clear-lower-block rule

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

Blocks World

- A sample run of the program

```
CLIPS> (unwatch all)
CLIPS> (reset)
CLIPS> (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
CLIPS>
```

Multifield Wildcards and Variables

- Multifield wildcards and variables can be used to match against zero or more fields of a pattern.
- The multitield wildcard is indicated by a dollar sign followed by a question mark, “$?”, and represents zero or more occurrences of a field.
  - The *person* pattern will match any name slot that contains at least one field and has as its last field the specified name:

```
(defrule print-social-security-numbers
 (print-ss-numbers-for ?last-name)
 (person (name $? ?last-name)
 (social-security-number ?ss-number))
 =>
 (printout t ?ss-number crlf))
```

Multifield Wildcards and Variables

- Multifield variables are preceded by a "$?"

```
(deftemplate person
  (multislot name)
  (multislot children))

(deffacts some-people
  (person (name John Q. Public)
          (children Jane Paul Mary))
  (person (name Jack R. Public)
          (children Rick) ) )

(defrule print-children
  (print-children $?name)
  (person (name $?name)
          (children $?children))
=>
(printout t ?name " has children" ?children
          crlf) )
```

Multifield Wildcards and Variables

– The following dialog shows how the *print-children* rule works:

```
CLIPS> (reset)
CLIPS> (assert (print-children John Q. Public)
<Fact-3>
CLIPS> (run)
 (John Q. Public) has children (Jane Paul Mary)
CLIPS>
```

- More than one multifield variable can be used in a single slot:

```
(defrule find-child
 (find-child ?child)
 (person (name $?name)
          (children $?before ?child $?after))
=>
 (printout t ?name " has child " ?child crlf)
 (printout t "Other children are " $?before " " $?after
crlf))
```

Blocks World Revisited

- Reimplementation with multifield wildcards and variables
- Stacks are represented as single facts.

```
(deffacts initial-state
(stack A B C)
(stack D E F)
(goal (move C) (on-top-of E))
(stack) )
```

- The empty *stack* fact is included to prevent this fact from being added later.

Blocks World Revisited

```
(defrule move-directly
   ?goal <- (goal (move ?blockl)
                  (on-top-of ?block2))
   ?stack-l <- (stack ?block1 $?rest1)
   ?stack-2 <- (stack ?block2 $?rest2)
   =>
   (retract ?goal ?stack-1 ?stack-2)
   (assert (stack $?rest1))
   (assert (stack ?block1 ?block2 $?rest2))
   (printout t ?block1 " moved on top of "
            ?block2 "." crlf))
```

Blocks World Revisited

```
(defrule move-to-floor
  ?goal <- (goal (move ?blockl)
                 (on-top-of floor))
  ?stack-l <-(stack ?block1 $?rest)
  =>
  (retract ?goal ?stack-l)
  (assert (stack ?blockl))
  (assert (stack $?rest))
  (printout t ?blockl " moved on top of floor."
            crlf))
```

Blocks World Revisited

```
(defrule clear-upper-block
  (goal (move ?block1))
  (stack ?top $? ?block1 $?)
  =>
  (assert (goal (move ?top)
                (on-top-of floor))))

(defrule clear-lower-block
  (goal (on-top-of ?blockl))
  (stack ?top $? ?blockl $?)
  =>
  (assert (goal (move ?top)
                (on-top-of floor))))
```

Field Constraints

- The *Not* Field Constraint
  - Its symbol is the tilde, “~.”
  - The *not* constraint acts on the one constraint or variable that immediately follows it.

```
(defrule person-without-brown-hair
   (person (name ?name) (hair ~brown))
   =>
   (printout t ?name " does not have brown hair"
        crlf) )
```

The Or Field Constraint

- The ***or* constraint**
  - represented by the bar, “|“
  - is used to allow one or more possible values to match a field of a pattern

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

The *And* Field Constraint

- The symbol of the *and* constraint is the ampersand, “&”

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

Functions and Expressions

- CLIPS Elementary Arithmetic Operators

Arithmetic Operators	Meaning
+	Addition
−	Subtraction
*	Multiplication
/	Division

- A numeric expression must be written in prefix form

```
Infix form: (y2 – y1) / (x2 – x1)>0
Prefix form: ( > (/ (- y2 y1)(- x2 x1)) 0)
```

Functions and Expressions

```
CLIPS> (+ 2 3.0)
5.0
CLIPS> (+ 2.0 3)
5.0
CLIPS> (+ 2 3)
5
CLIPS>
CLIPS> (+ 2 3 4) <- Evaluation proceeds from left to right
9
CLIPS> (- 2 3 4)
-5
CLIPS> (* 2 3 4)
24
CLIPS> (/ 2 3 4)
0.166667
CLIPS>
The answer for division may vary slightly depending on the
   machine being used.
```

Functions and Expressions

```
2 + 3 * 4
1.    2 + (3 * 4)
      CLIPS> (+ 2 (* 3 4))
      14
      CLIPS>
2.    (2 + 3) * 4
      CLIPS> (* (+ 2 3) 4)
      20
      CLIPS>
```

```
CLIPS> (assert (answer (+ 2 2)))
<Fact-0>
CLIPS> (facts)
f-0  (answer 4)
For a total of 1 fact.
CLIPS> (clear)
```

Summing Values using Rules

- As a simple example of using functions to perform calculations
  - consider the problem of summing up the area of a group of rectangles.
  - The heights and widths are specified with the following deftemplate.

    ```
    (deftemplate rectangle (slot height) (slot width) )
    ```

  - A deffacts containing sample information

    ```
    (deffacts initial-information
            (rectangle (height 10) (width 6))
            (rectangle (height 7) (width 5))
            (rectangle (height 6) (width 8))
            (rectangle (height 2) (width 5))
            (sum 0))
    ```

Summing Values using Rules

- An attempt to produce a rule to sum the rectangle:

```
(defrule sum-rectangles
   (rectangle (height ?height) (width ?width))
   ?sum <- (sum ?total)
   =>
   (retract ?sum)
   (assert (sum (+ ?total (* ?height ?width)))))
```

  – This rule, however, will loop endlessly.
    - One solution to solve the problem would be to retract the *rectangle* fact after its area was added to the *sum* fact.
    - If the *rectangle* fact needs to be preserved, a different approach is required.

Summing Values using Rules

```
(defrule sum-rectangles
   (rectangle (height ?height) (width ?width))
   =>
   (assert (add-to-sum (* ?height ?width))))

(defrule sum-areas
   ?sum <- (sum ?total)
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (assert (sum (+ ?total ?area))))
```

The *BIND* Function

- The **bind** function can be used to bind the value of a variable to the value of an expression.

```
(bind <variable> <value>)
```

- The bound variable，<variable>，uses the syntax of a single-field variable.
- The new value，<value>，should be an expression that evaluates to either a single-field or a multifield value.

```
(defrule sum-areas
    ?sum <- (sum ?total)
    ?new-area <- (add-to-sum ?area)
    =>
    (retract ?sum ?new-area)
    (printout t "Adding " ?area " to " ?total crlf)
    (bind ?new-total (+ ?total ?area))
    (printout t "New sum is " ?new-total crlf)
    (assert (sum ?new-total)))
```

I/O Functions

- The *Read* Function
  - CLIPS allows information to be read from the keyboard using the read function.

```
CLIPS> (clear)
CLIPS>
(defrule get-first-name
  =>
  (printout t "What is your first name? ")
  (bind ?response (read))
  (assert (user's-name ?response)))
CLIPS> (reset)
CLIPS> (run)
What is your first name? Gary
CLIPS> (facts)
f-0    (initial-fact)
f-1     (user's-name Gary)
For a total of 2 facts.
CLIPS>
```

I/O Functions

- The *Open* Function
  - Before a file can be accessed for reading or writing, it must be opened using the open function.

```
(open <file-name> <file-ID> [<file-access>])
```

  - As an example,
    - `(open "input.dat" data "r")`
  - File Access Modes

Mode	Action
"r"	Read access only
"w"	Write access only
"r+"	Read and write access
"a"	Append access only

I/O Functions

- If <file-access> is not included as an argument, the default value of "r" will be used.
- The *open* function acts as a predicate function.
  - It returns the symbol *TRUE* if a file was successfully opened; otherwise the symbol *FALSE* is returned .

I/O Functions

- The Close Function
  - Once access is no longer needed to a file, it should be closed.

```
(close [<file-ID>])
```

  - Where the optional argument <file-ID> specifies the logical name of the file to be closed. If <file-ID> is not specified, all open files will be closed.
  - E.g.

```
(close data)
```

  - Remember each opened file should eventually be closed with the close function.
  - If a command is not issued to close a file, the data written to it may be lost.

I/O Functions

- Reading and Writing to a File
  - The use of logical names allows input and output to and from other sources.

```
CLIPS> (open "example.dat" example "w")
TRUE
CLIPS> (printout example "green" crlf)
CLIPS> (printout example 7 crlf)
CLIPS> (close example)
TRUE
CLIPS>
```

I/O Functions

- The general format of the *read* function is:

```
(read [<logical-name>])
```

  – The *read* function defaults to reading from the standard input device, *t,* if it is given no arguments.

```
CLIPS> (open "example.dat" example "r")
TRUE
CLIPS>(read example)
green
CLIPS>(read example)
7
CLIPS>(read example)
EOF
CLIPS>(read example)
TRUE
CLIPS>
```

I/O Functions

- The *Format* Function

```
(format <logical-name> <control-string>
        <parameters>*)
```

  – control string, which must be contained within double quotes.
    - The control string consists of format flags
  – The return value of the *format* function is the formatted string.
  – If the logical name *nil* is used with the format command, then no output is printed (either to the terminal or a file), but the formatted string is still returned.

```
CLIPS> (format nil "Name: %-15s Age: %3d "
                    "Bob Green" 35)
"Name: Bob Green          Age: 35"
CLIPS> (format nil "Name: %-15s Age: %3d "
                   "Ralph Heiden" 32)
"Name: Ralph Heiden      Age: 32"
CLIPS>
```

I/O Functions

- The format flag “%-15s” is used to print the name in a column that is 15 characters wide.
- The - sign indicates that the output is to be left justified and the character s indicates that a string or symbol is to be printed.
- The general specification of a format flag is:

  %-M.Nx

  – where “-” is optional and means to left justify. The default is to right justify.
  – The letter M is a number specifying the field width in columns.
  – The letter N is an optional number specifying the number of digits past the decimal point that will be printed.
  – The letter x is a character specifying the display format specification.

I/O Functions

- Display format Specifications

Character	Meaning
d	Integer
f	Floating-point
e	Exponential (in power-of-ten format)
g	General (numeric); display in whatever format is shorter
o	Octal; unsigned number (N specifier not applicable)
x	Hexadecimal; unsigned number (N specifier not applicable)
s	String; quoted strings will be stripped of quotes
n	Carriage return/line feed
%	The "%" character itself

I/O Functions

- The Readline Function
  - The readline function can be used to read an entire line of input.

```
(readline [<logical-name>])
```

```
CLIPS> (clear)
CLIPS> (defrule get-name
   =>
   (printout t "What is your name? ")
   (bind ?response (readline))
   (assert (user's-name ?response)))
CLIPS> (defrule print-singlefield-values
        (user's-name ?name)
        =>
        (printout t ?name crlf))
CLIPS> (reset)
CLIPS> (run)
What is your name? Swan Lai Tsz-wai
Swan Lai Tsz-wai
CLIPS> (facts)
f-0     (initial-fact)
f-1     (user's-name "Swan Lai Tsz-wai")
For a total of 2 facts.
CLIPS>
```

Predicate Functions

- A predicate function is defined to be any function that returns either the symbol *TRUE* or the symbol *FALSE.*
- Predicate functions may be either predefined or user-defined functions.
  - Predefined functions are those functions already provided by CLIPS.
  - User-defined or external functions are functions other than predefined functions that are written in C or another language and linked with CLIPS.

```
CLIPS> (and (> 4 3) (> 4 5))
FALSE
CLIPS> (or (> 4 3) (> 4 5))
TRUE
CLIPS> (> 4 3)
TRUE
CLIPS> (< 6 2)
FALSE
CLIPS> (integerp 3)
TRUE
CLIPS> (integerp 3.5)
FALSE
CLIPS>
```

The Test Conditional Element

- The test conditional element provides a powerful way to evaluate expressions on the LHS of the rule.
- Instead of pattern matching against a fact in the fact list, the test CE evaluates an expression.
- If the expression evaluates to any value other than the symbol *FALSE,* the test CE is satisfied. If the expression evaluates to the symbol *FALSE,* the test CE is not satisfied.

```
(test <predicate-function>)
e.g. (test (> ?size 1))
```

The Predicate Field Constraint

- The **predicate field constraint,** :, is useful for performing predicate tests directly within patterns.
- It can stand by itself in a field or be used as one part of a more complex field using the -, &, and | connective field constraints.
- The predicate field constraint is always followed by a function for evaluation

```
(pile-size ?size)
(test (> ?size 1))
```

- These two patterns could be replaced with the following single pattern:

```
(pile-size ?size&:(> ?size 1))
```

The Return Value Field Constraint

- The **return value field constraint, =,** allows the return value of a function to be used for comparison inside a pattern.
- Like the predicate field constraint, the return value field constraint must be followed by a function.
- However, the function does not have to be a predicate function.
- The only restriction is that the function must have a single-field return value.
- For example, the field constraint:

```
=(mod ?size 4)
```

  - could be read as “The field is equal to ?size modulus 4.”

The *OR* Conditional Element

- Two separate rules can be combined using an or CE.

```
(defrule shut-off-electricity
   (or (emergency (type flood))
       (extinguisher-system (type water-sprinkler)
                            (status on)))
   =>
   (printout t "Shut off the electricity" crlf))
```

- A more appropriate rule is to update the fact list to indicate the electricity has been shut off.

```
(defrule shut-off-electricity
    ?power <- (electrical-power (status on))
    (or (emergency (type flood))
        (extinguisher-system (type water-sprinkler)
                             (status on)))
    =>
    (modify ?power (status off)))
    (printout t "Shut off the electricity" crlf))
```

The *AND* Conditional Element

- The *and* CE is provided so it can be used with other CEs to make more complex patterns.

```
(defrule use-carbon-dioxide-extinguisher
  ?system <- (extinguisher-system
               (type carbon-dioxide)
               (status off))
  (or (emergency (type class-B-fire))
      (and (emergency (type class-C-fire))
           (electrical-power (status off))))
  =>
  (modify ?system (status on))
  (printout t "Use carbon dioxide extinguisher"
               crlf) )
```

The *NOT* Conditional Element

- CLIPS allows the specification of the absence of a fact in the LHS of a rule using the *not* conditional element.

```
IF the monitoring status is to be reported and
  there is an emergency being handled
THEN report the type of the emergency
IF the monitoring status is to be reported and
  there is no emergency being handled
THEN report that no emergency is being handled
```

- The *not* CE can be conveniently applied to the simple rules above as follows:

```
(defrule report-emergency
   (report-status)
   (emergency (type ?type))
   =>
   (printout t "Handling" ?type " emergency" crlf) )
(defrule no-emergency
   (report-status)
   (not (emergency))
   =>
   (printout t "No emergency being handled" crlf))
```

The *EXISTS* Conditional Element

- The *exists* conditional element allows you to pattern match based on the existence of at least one fact that matches a pattern without regard to the total number of facts that actually match the pattern.
- This allows a single partial match or activation for a rule to be generated based on the existence of one fact out of a class of facts.
- E.g.

```
(deftemplate emergency (slot type))
(defrule operator-alert-for-emergency
  (exists (emergency))
  =>
  (printout t "Emergency: Operator Alert" crlf)
  (assert (operator-alert)))
```

THE *FORALL* CONDITIONAL ELEMENT

- It allows you to pattern match based on a set of CEs that are satisfied for every occurrence of another CE.
- The general format of the *forall* CE is as follows:

```
(forall <first-CE>
    <remaining-CEs>+)
```

  - Each fact matching the <first-CE> must also have facts that mach all of the <remaining-CEs>.

```
(deftemplate emergency (slot type) (slot location))
(deftemplate fire-squad (slot name) (slot location))
(deftemplate evacuated (slot building))
(defrule all-fires-being-handled
  (forall (emergency (type fire) (location ?where))
          (fire-squad (location ?where))
          (evacuated (building ?where)))
          =>
          (printout t "All buildings that are on fire " crlf
                  "have been evacuated and" crlf
                  "have firefighters on location" crlf))
```