CLIPS Tutorial 3

Advanced Pattern Matching (cont’d)

&

Some more examples
The *OR* Conditional Element

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

(deffrule 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.

(deffrule 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.

\[
\text{(defrule use-carbon-dioxide-extinguisher} \\
\quad \text{?system <- (extinguisher-system} \\
\quad \quad \text{(type carbon-dioxide)} \\
\quad \quad \text{(status off))} \\
\quad \text{(or (emergency (type class-B-fire))} \\
\quad \quad \text{(and (emergency (type class-C-fire))} \\
\quad \quad \quad \text{(electrical-power (status off))})}) \\
\quad \Rightarrow \\
\quad \text{(modify ?system (status on))} \\
\quad \text{(printout t "Use carbon dioxide extinguisher"} \\
\quad \quad \text{crlf)}
\]
The \textit{NOT} Conditional Element

- CLIPS allows the specification of the absence of a fact in the LHS of a rule using the \textit{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 \textit{not} CE can be conveniently applied to the simple rules above as follows:

  \begin{verbatim}
  (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))
  \end{verbatim}
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.

```lisp
(deftemplate emergency (slot type))
(defrule operator-alert-for-emergency
  (exists (emergency))
  =>
  (printout t "Emergency: Operator Alert" crlf)
  (assert (operator-alert)))
```
THE FOR ALL 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 for all CE is as follows:
  
  (forall <first-CE>
   <remaining-CEs>+)

  Each fact matching the <first-CE> must also have facts that match 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))

  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))

  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))
Some more examples
The Field Constraints

- Combining the and constraint & the not constraint:

  **Example 1:**
  
  (defrule black-or-brown-hair
   (person (name ?name)
   (hair ?color & ~brown & ~black))
  =>
  (printout t ?name “ has “ ?color “ hair” crlf))

- **Example 2:**

  (defrule complex-eye-hair-match
   (person (name ?name1)
   (eyes ?eyes1 & blue | green)
   (hair ?hair1 & ~black))
   (person (name ?name2 & ~?name1)
   (eyes ?eyes2 & ~?eyes1)
   (hair ?hair2 & red | ?hair1))
  =>
  (printout t ?name1 “ has “ ?eyes1 “ eyes and “ ?hair1 “ hair” crlf)
  (printout t ?name2 “ has “ ?eyes2 “ eyes and “ ?hair2 “ hair” crlf))
The Test Conditional Element

- Using the test CE with other predicate functions
  - Example: > predicate function
    - (test (> ?size 1))
  - Example: and predicate function
  - Example: or predicate function, not predicate function:
    - (test (or (not (integrp ?choice)) (< ?choice 1) (> ?choice 3) (>= ?choice ?size)))
The Predicate Field Constraint

- Complex field by combining with the ~, & , and | connective field constraints
  - ?size&:(> ?size 1)
  - ?size&:(> ?size 1)&:(< ?size 5)
  - ?size&:(> ?size 1)|:(= ?size 5)

- Example rules:
  (defrule add-sum
   (data-item ?value&:(numberp ?value))
   ?old-total <- (total ?total)
   =>
   (retract ?old-total)
   (assert (total (+ ?total ?value)))))
Example:

(defrule find-data-type-1
  (data ?item &: (stringp ?item) | :(symbolp ?item))
=>
  (printout t ?item " is a string or symbol" crlf))

(defrule find-data-type-2
  (data ?item &~:(integerp ?item))
=>
  (printout t ?item " is not an integer" crlf))
The return value field constraints

Example:

```
(defrule rule-1
    (size ?size)
    (remainder ==(mod ?size 4))
=>
    (printout "matched" crlf))
```
Some Useful Commands & Functions

- **Math functions**
  - `(max <numeric-expression> <numeric-expression>)`
    - Returns the value of its largest arguments
  - `(min <numeric-expression> <numeric-expression>)`
    - Returns the value of its smallest arguments
  - `(+ <numeric-expression> <numeric-expression>)`
  - `(- <numeric-expression> <numeric-expression>)`
  - `(* <numeric-expression> <numeric-expression>)`
  - `(/<numeric-expression> <numeric-expression>)`
  - `(abs <numeric-expression>)`
    - Returns the absolute value of its only argument
  - `(integer <numeric-expression>)`
    - Returns its only argument converted to type integer
  - `(float <numeric-expression>)`
    - Returns its only argument converted to type float
Predicate

- (and <expression>+)
  - Returns TRUE if each of its arguments evaluates to TRUE, otherwise FALSE
- (eq <expression> <expression>+)
  - Returns TRUE if its first argument is equal in type and value to all its subsequent arguments, otherwise FALSE
- (neq <expression> <expression>+)
  - Returns TRUE if its first argument is not equal in type and value to all its subsequent arguments, otherwise FALSE
- (not <expression>)
  - Returns TRUE if its arguments evaluates to FALSE, otherwise TRUE
- (or <expression>+)
  - Returns TRUE if its arguments evaluates to TRUE, otherwise FALSE
- (= <numeric-expression> <numeric-expression>+)
  - Returns TRUE if its first argument is equal in numeric value to all its subsequent arguments, otherwise FALSE
- (<> <numeric-expression> <numeric-expression>+)
  - Returns TRUE if its first argument is not equal in numeric value to all its subsequent arguments, otherwise FALSE
Predicate

- $(> <\text{numeric-expression}> <\text{numeric-expression}>+)$
  - Returns TRUE if for all argument is, argument n-1 is greater than argument n, otherwise FALSE

- $(\geq <\text{numeric-expression}> <\text{numeric-expression}>+)$
  - Returns TRUE if for all argument is, argument n-1 is greater than or equal to argument n, otherwise FALSE

- $(< <\text{numeric-expression}> <\text{numeric-expression}>+)$
  - Returns TRUE if for all argument is, argument n-1 is less than argument n, otherwise FALSE

- $(\leq <\text{numeric-expression}> <\text{numeric-expression}>+)$
  - Returns TRUE if for all argument is, argument n-1 is less than or equal to argument n, otherwise FALSE
Predicate

- `(numberp <expression>)`
  - Returns TRUE if `<expression>` is an integer or a float, otherwise FALSE
- `(integerp <expression>)`
  - Returns TRUE if `<expression>` is an integer, otherwise FALSE
- `(floatp <expression>)`
  - Returns TRUE if `<expression>` is a float, otherwise FALSE
- `(lexemep <expression>)`
  - Returns TRUE if `<expression>` is a string or symbol, otherwise FALSE
- `(stringp <expression>)`
  - Returns TRUE if `<expression>` is a string, otherwise FALSE
- `(symbolp <expression>)`
  - Returns TRUE if `<expression>` is a symbol, otherwise FALSE
String

(str-compare <string-or-symbol-expression> <string-or-symbol-expression>)

- Returns zero if both arguments are equal, a positive integer if the first argument is lexicographically greater than the second argument, and a negative integer if the first argument is lexicographically less than the second argument.