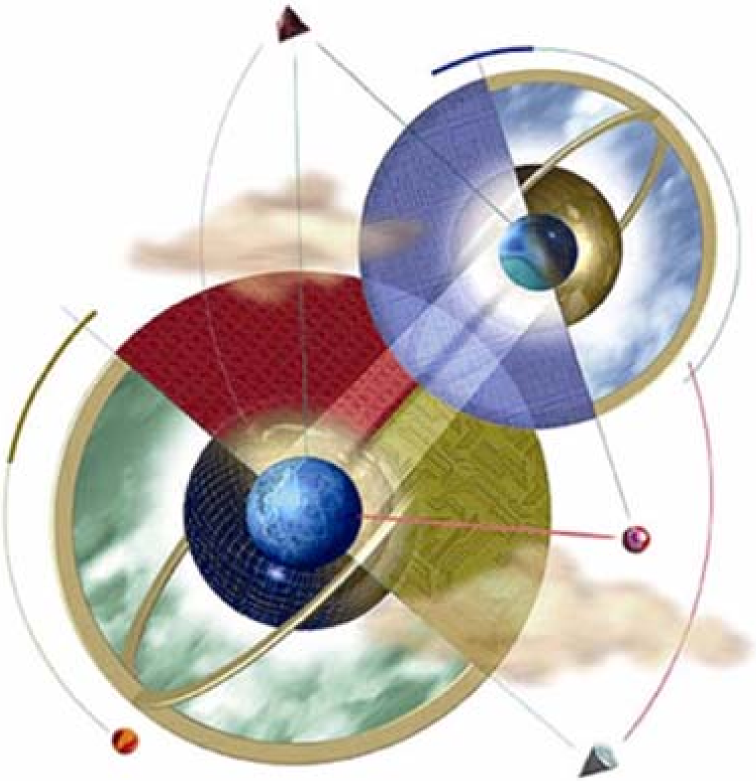


Chapter 8: Advanced Pattern Matching

Expert Systems: Principles and Programming, Fourth Edition

Field Constraints

- In addition to pattern matching capabilities and variable bindings, CLIPS has more powerful pattern matching operators – field constraints.
- Consider writing a rule for all people who do *not* have brown hair:
  - We could write a rule for every type of hair color that is not brown.
    - add another pattern CEs to test the variable ?color
    - or, place a field constraint as the value of the slot directly
    - or, attach a field constraint to the variable ?color

Connective Constraints

- Connective constraints are used to connect variables and other constraints.
  - *Not field constraint* – the ~ acts on the one constraint or variable that immediately follows it.
    - (defrule person-without-brown-hair
      (person (name ?name) (hair ?color))
      (test (neq ?color brown))
      =>
      (printout t ?name " does not have brown hair." crlf))
    - (defrule person-without-brown-hair
      (person (name ?name) (hair ~brown))
      =>
      (printout t ?name " does not have brown hair." crlf))

Connective Constraints (cont.)

- *Or field constraint* – the symbol | is used to allow one or more possible values to match a field or a pattern.
  - (defrule person-with-black-or-brown-hair

    (person (name ?name) (hair black | brown))

    =>

    (printout t ?name " has dark hair." crlf))
- *And field constraint* – the symbol & is useful with binding instances of variables and on conjunction with the *not* constraint
  - (defrule person-without-black-or-red-hair

    (person (name ?name) (hair ?color&~black&~red))

    =>

    (printout t ?name " has " ?color " hair." crlf))

Combining Field Constraints

- Field constraints can be used together with variables and other literals to provide powerful pattern matching capabilities.
- Example #1: ?eyes1&blue|green
  - This constraint binds the person's eye color to the variable, ?eyes1 if the eye color of the fact being matched is either blue or green.
- Example #2: ?hair1&~black
  - This constraint binds the variable ?hair1 if the hair color of the fact being matched is not black.

Complex Field Constraints

- For example, a rule to determine whether the two people exist:
  - The first person has either blue or green eyes and does not have black hair
  - The second person does not have the same color eyes as the first person and has either red hair or the same color hair as the first person
- (defrule complex-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 binding of the variable

- Variables will be bound only if they are the first condition in a field and only if they occur singly or are tied to the other conditions by an *and* connective constraint.
- For example
  - (defrule bad-variable-use

    (person (name ?name) (hair red |?hair))

    Will cause error!

    =>

    (printout t ?name " has " ?hair " hair " crlf))

Functions and Expressions

- CLIPS has the capability to perform calculations.
- The math functions in CLIPS are primarily used for modifying numbers that are used to make inferences by the application program.

Table 8.1 CLIPS Elementary Arithmetic Operators

Arithmetic Operators	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division

Numeric Expressions in CLIPS

- Numeric expressions are written in CLIPS in LISP-style – using prefix form – the operator symbol goes before the operands to which it pertains.
- Example #1:

  5 + 8 (infix form) → + 5 8 (prefix form)

- Example #2:

  (infix) (y2 – y1) / (x2 – x1) > 0

  (prefix) (> (/ (- y2 y1) (- x2 x1)) 0)

Return Values

- Most functions (or operators) have a return value that can be an integer, float, symbol, string, or multivalued value.
- Some functions (*facts*, *agenda* commands) have no return values – just side effects.
- Return values for +, -, and * will be integer if all arguments are integer, but if at least one value is ***floating point***, the value returned will be float.

Variable Numbers of Arguments

- Many CLIPS functions accept variable numbers of arguments.
- Example:

  CLIPS> (- 3 5 7) ↵ returns `3 - 5 =-2 - 7 = -9`

- There is no built-in arithmetic precedence in CLIPS – everything is evaluated from left to right.
- To compensate for this, precedence must be explicitly written.
  - (- (- 3 5) 7) or (- 3 (- 5 7))

Embedding Expressions

- Expressions may be freely embedded within other expressions:

```
CLIPS>  (assert (result (+ 3 6)))↵
<Fact-0>
CLIPS> (facts)↵
f-0 (result 9)
For a total of 1 fact.

CLIPS>  (assert (expression 3 + 6 * 10))↵
<Fact-0>
CLIPS> (facts)↵
f-0 (expression 3 + 6 * 10)
For a total of 1 fact.
```

Summing Values Using Rules

- Suppose you wanted to sum the areas of a group of rectangles.
  - The heights and widths of the rectangles can be specified using the deftemplate:

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

- The sum of the rectangle areas could be specified using an ordered fact such as:

```
(sum 20)
```

Summing Values

- A deffacts containing sample information is:

```
(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))
```

- to permit rectangles of same size
  – (rectangle (id 12) (height 10) (width 6))

Summing Values

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

(deffacts initial-rectangles
  (rectangle (height 10) (width 6))
  (rectangle (height 8) (width 5))
  (rectangle (height 3) (width 7))
  (sum 0))

(defrule compute-area
  (rectangle (height ?height) (width ?width))
  =>
  (assert (add-to-sum (* ?height ?width) ) ) )

(defrule sum-1
  ?sum <- (sum ?total)
  ?new-area <- (add-to-sum ?area)
  =>
  (retract ?sum ?new-area)
  (assert (sum (+ ?total ?area) ) )
  (printout t "The new sum is " (+ ?total ?area) crlf))
```

Summing Values (cont.)

- (defrule sum-2
  (rectangle (height ?height) (width ?width))
  ?sum <- (sum ?total)
  =>
  (retract ?sum)
  (assert (sum (+ ?total (* ?height ?width))))
  (printout t "The new sum is " (+ ?total (* ?height ?width)) crlf))

What happen??

- (defrule sum-3
  ?rect <- (rectangle (height ?height) (width ?width))
  ?sum <- (sum ?total)
  =>
  (retract ?rect ?sum)
  (assert (sum (+ ?total (* ?height ?width))))
  (printout t "The new sum is " (+ ?total (* ?height ?width)) crlf))

Compare with sum-1

The Bind Function

- Sometimes it is advantageous to store a value in a temporary variable to avoid recalculation.
- The *bind* function can be used to bind the value of a variable to the value of an expression using the following syntax:

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

-
```
(defrule sum-3
   ?rect <- (rectangle (height ?height) (width ?width))
   ?sum <- (sum ?total)
   =>
   (retract ?rect ?sum)
   (bind ?new (+ ?total (* ?height ?width) ) )
   (assert (sum ?new) )
   (printout t "The new sum is " ?new crlf))
```

I/O Functions

- When a CLIPS program requires input from the user of a program, a *read* function can be used to provide input from the keyboard:

```
(defrule get-name
    =>
    (printout t "What is your name? ")
    ( bind ?response (read))
    (assert (user's-name ?response)))
```

Read Function from Keyboard

- The *read* function can only input a single field at a time.
  - CLIPS> (read)

    John K. Smith
- Characters entered after the first field are discarded.
  - K. Smith are discarded
- To input, say a first and last name, they must be delimited with quotes, "John K. Smith".
- Data must be followed by a carriage return (↵) to be read.

I/O from/to Files

- Input can also come from external files.
- Output can be directed to external files.
- Before a file can be accessed, it must be opened using the *open* function:

  Example:

  ```
  (open “c:\\datafile.dat” mydata “r”)
  ```

- The *open* function acts as a predicate function
  - Returns true if the file was successfully opened
  - Returns false otherwise

I/O from/to Files (cont.)

```
(open "c:\\datafile.dat" mydata "r")
```

- datafile.dat – is the name of the file (path can also be provided)
- mydata – is the logical name that CLIPS associates with the file
- "r" – represents the mode – how the file will be used – here read access

Table 8.2 File Access Modes

Mode	Action
"r"	Read access only
"w"	Write access only (overwrite all)
"r+"	Read and write access
"a"	Append access only (append to EOF)

Close Function

- Once access to the file is no longer needed, it should be closed.
- Failure to close a file may result in loss of information.
- General format of the *close* function:

  (close [<logical-name>])

  – ex. (close mydata)

Reading / Writing to a File

- Which logical name used, depends on where information will be written – logical name *t* refers to the terminal (standard output device).

Writing to a File	Reading from a File
(open "myresult.dat" stuff "w") ↵	(open "mydata.dat" stuff "r") ↵
(printout stuff "apple" crlf) ↵	(read stuff) ↵
(printout stuff 8 crlf) ↵	(read stuff) ↵
(close stuff) ↵	(read stuff) ↵
	(close stuff) ↵

Formatting

- Output sent to a terminal or file may need to be formatted – enhanced in appearance.
- To do this, we use the *format* function which provides a variety of formatting styles.
- General format:

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

  – output the result to the logical name

  – return the result

Formatting (cont.)

- Logical name :
  - *t* indicates standard output device
  - logical name associated with a file
  - *nil* indicates that no output is printed, but the formatted string is still returned
- Control string:
  - Must be delimited with quotes (“)
  - Consists of *format flags* (%) to indicate how parameters should be printed
  - one-to-one correspondence between *flags* and number of *parameters* – constant values or expressions

Formatting

- Example:

  (format nil "Name: %-15s Age: %3d" "Bob Green" 35) ↵

Produces the results and returns:

  "Name: Bob green   Age: 35"

  – %-15s   %3d
    - - indicates left-justified
    - s indicates string
    - 15 , 3 indicates the width
    - d indicates integer

Specifications of Format Flag

%-m.Nx

- The "-" means to left justify (right is the default)
- m – total field width
- N – number of digits of precision (default = 6)
- x – display format specification

Table 8.3 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

Readline Function

- To read an entire line of input, the *readline* function can be used:

  (readline [<logical-name>])

- Example:

```
(defrule get-name
   =>
   (printout t “What is your name? “)
   (bind ?response (readline))
   (assert (user’s-name ?response)))
```

  – What is your name? John K. Smith ↵
    - ?response = “John K. Smith”
    - (user’s-name “John K. Smith”)

explode$ Function

- (readline)
  - read a line (as a string)
- (explode$ <string>)
  - convert a string into a multi-field value

```
(defrule get-name
  =>
  (printout t “What is your name? “
  (bind ?response (explode$ (readline)))
  (assert (user’s-name ?response)))
```

  - What is your name? John K. Smith ↵
    - ?response = John K. Smith
    - (user’s-name John K. Smith)

Predicate Functions

- A predicate function is defined to be any function that returns:
  - TRUE
  - FALSE
- Any value other than FALSE is considered TRUE.
- We say the predicate function returns a Boolean value.

The *Test Conditional* Element

- Processing of information often requires a loop.
- Sometimes a loop needs to terminate automatically as the result of an arbitrary expression.
- The *test condition* provides a powerful way to evaluate expressions on the LHS of a rule.
- Rather than pattern matching against a fact in a fact list, the *test CE* evaluates an expression – outermost function must be a predicate function.

Test Condition

- A rule will be triggered only if all its test CEs are satisfied along with other patterns.

(test <predicate-function>)

Example:

(test (> ?value 1))

Examples for *Test Condition*

- (defrule find-height-larger-than-170
```
    (person (name ?name) (age ?) (height ?height) (weight ?))
    (test (> ?height 170))
    =>
    (printout t ?name "'s height is larger than 170 cm." crlf))
```
- (defrule find-person
```
    (person (name ?name) (age ?age) (height ?height) (weight ?weight))
    (test (and (>= ?height 170)
               (or (> ?weight 60)
                   (< ?age 30))))
    =>
    (printout t ?name " is the person we seek." crlf))
```

```
f-5 (person (name Peter) (age 35) (height 175) (weight 60))
f-6 (person (name David) (age 25) (height 170) (weight 55))
```

Predicate Field Constraint

- The predicate field constraint **:** allows for performing predicate tests *directly within patterns*.
  - The predicate field constraint is *more efficient* than using the test CE.

```
(defrule find-height-larger-than-170
(person (name ?name) (age ?) (height ?height&:(> ?height 170)) (weight ?))
=>
(printout t ?name "'s height is larger than 170 cm." crlf))
```

```
f-5 (person (name Peter) (age 35) (height 175) (weight 60))
f-6 (person (name David) (age 25) (height 170) (weight 55))
f-7 (person (name John) (age 12) (height 145) (weight 40))
f-8 (person (name Kevin) (age 31) (height 200) (weight 98))
```

Return Value Constraint

- The return value constraint = allows the return value of a function to be used for comparison inside LHS patterns
  - The function must have a single-field return value.
- (defrule find-one-is-30cm-taller-than-another

```
(person (name ?name1) (age ?) (height ?height) (weight ?))
(person (name ?name2) (age ?) (height =(+ ?height 30)) (weight ?))
 =>
(printout t ?name2 " is 30 cm taller than " ?name1 crlf))
```

```
f-5 (person (name Peter) (age 35) (height 175) (weight 60))
f-6 (person (name David) (age 25) (height 170) (weight 55))
f-7 (person (name John) (age 12) (height 145) (weight 40))
f-8 (person (name Kevin) (age 31) (height 200) (weight 98))
```

The *OR* Conditional Element

Consider the two rules:

```
(defrule shut-off-electricity-1
    (emergency (type flood))
    =>
    (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-2
    (extinguisher-system (type water-sprinkler
                         (status on))
    =>
    (printout t "Shut off the electricity" crlf))
```

OR Conditional Element

These two rules can be combined into one rule – *or* CE requires only one CE be satisfied:

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

The *And* Conditional Element

The *and* CE is opposite in concept to the *or* CE – requiring all the CEs be satisfied:

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

Not Conditional Element

When it is advantageous to activate a rule based on the absence of a particular fact in the list, CLIPS allows the specification of the absence of the 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 handed
THEN report that no emergency is being handled
```

Not Conditional

We can implement this 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))
```

Not Conditional

Consider the following rule, which looks for the largest number out of a group of facts representing numbers

```
(defrule largest-number
  (number ?x)
  (not (number ?y&: (> ?y ?x)))
=>
  (printout t “Largest number is “ ?x crlf))
```

Scope of *Not* Conditional

Consider the following rule

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

Scope of *Not* Conditional

Consider the following rules

```
(defrule no-birthday-on-specific-date
  (check-for-no-birthday (date ?date))
  (not (person (birthday ?date)))
=>
  (printout t “No birthday on “ ?date crlf))
```

```
(defrule no-birthday-on-specific-date
    (not (person (birthday ?date)))
   (check-for-no-birthday (date ?date))
=>
  (printout t “No birthday on “ ?date crlf))
```

Not Conditional

No two people who have birthdays on the same date

```
(defrule no-identical-birthdays
  (not (and (person (name ?name)
                    (birthday ?date))
            (person (name ~?name)
                    (birthday ?date))))
=>
  (printout t “No two people have the same birthday” crlf))
```

A Complex Example

- (defrule can-not-drive-in-Taiwan

```
(defrule can-not-drive-in-Taiwan
 (query ?person)
 (not (or (driving-license (id ?) (country Taiwan) (name ?person))
          (and (driving-license (id ?) (country ?else) (name ?person))
               (DL-accepted-in-Taiwan ?else) ) ) ) )
 =>
(printout t ?person " can not drive a car in Taiwan." crlf))
```

f-3 (query John)
f-4 (query Kevin)
f-5 (query Joe)
f-6 (driving-license (id 85346) (country Taiwan) (name Kevin))
f-7 (driving-license (id 53861) (country America) (name John))
f-8 (DL-accepted-in-Taiwan America)

The *Exists* Conditional Element

- The *exists* CE allows one 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.

Exists Condition

```
(deftemplate emergency (slot type))

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

Exists Condition (cont.)

- When more than one emergency fact is asserted, the message to the operators is printed more than once. The following modification prevents this:

```
(defrule operator-alert-for-emergency
    (emergency)
    (not (operator-alert))
    =>
    (printout t "Emergency: Operator Alert" crlf)
    (assert (operator-alert)))
```

Exists

- This assumes there was already an alert – triggered by an operator-emergency rule. What if the operators were merely placed on alert to a drill:

```
(defrule operator-alert-for-emergency
      (emergency)
      (not (operator-alert))
      =>
      (printout t "Drill: Operator Alert" crlf)
      (assert (operator-alert)))
```

Exists Condition (cont.)

- Now consider how the rule has been modified using the *exists* CE:

```
(defrule operator-alert-for-emergency
  (exists (emergency))
   =>
  (printout t “Emergency: Operator Alert” crlf)
  (assert (operator-alert))))
```

Forall Conditional Elements

- The *forall* CE allows one to pattern match based on a set of CEs that are satisfied for <u>every</u> occurrence of another CE.
- (forall <first-CE> <remaining-CEs>+)
  - each fact matching the <first-CE> must also have facts that match all of the <remaining-CEs>
  - (not (and <first-CE>

    (not (and <remaining-CEs>+))))

An Example for *Forall* Condition

- (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))
```

- Once an emergency fact is asserted, the rule is de-activated until the appropriate fire-squad and evacuated facts are asserted

Logical Conditional Elements

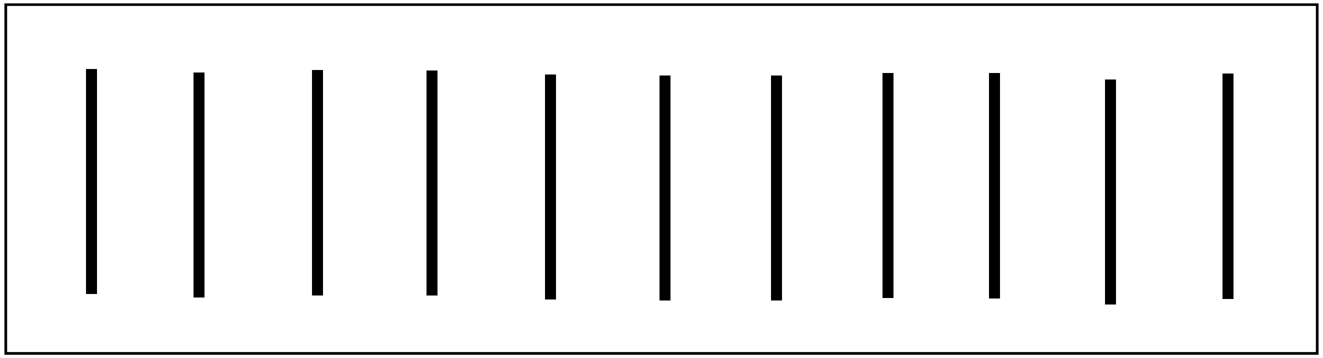
- The *logical* CE allows one to specify that the existence of a fact depends on the existence of another fact or group of facts.
- (defrule general-form-of-logical-CE

```
    (logical <dominate-fact>+)
    <remaining-fact>+
  =>
    (assert <dependent-fact>))
```

  - Once any dominate-fact is retracted, the dependent-fact is also retracted automatically
  - The retraction of any remaining-fact doesn’t cause the retraction of the dependent-fact

The Game of Sticks

- two player
- a pile of sticks to be taken
- must take 1, 2, or 3
- take the last stick => lose



The CLIPS Program

```
(deftemplate choose (slot get) (slot remainder))

(deffacts initial
   (choose (get 1) (remainder 1))
   (choose (get 1) (remainder 2))
   (choose (get 2) (remainder 3))
   (choose (get 3) (remainder 0)))

(defrule choose-total-number
   (declare (salience 1000))
  =>
   (printout t "How many sticks in the pile? ")
   (assert (total  (read))))

(defrule player-select
   (declare (salience 900))
  =>
   (printout t "Who moves first (Computer:c   Human: h)? ")
   (assert (player-get  (read))))

(defrule computer-get
   (declare (salience 800))
   ?player <- (player-get c)
   ?total <- (total ?num)
   (test (>= ?num 1))
   (choose (get ?get) (remainder =(mod ?num 4)))
=>
   (retract ?player ?total)
   (printout t  ?num " sticks in the pile." crlf)
   (assert (total (- ?num ?get)))
   (printout t "Computer take " ?get " sticks." crlf)
   (assert (player-get h)))
```

The CLIPS Program (cont.)

```
(defrule human-get
  (declare (salience 800))
  ?player <- (player-get h)
  ?total <- (total ?num)
  (test (>= ?num 1))
 =>
  (retract ?player ?total)
  (printout t “There are “ ?num “ sticks in the pile.” crlf)
  (printout t “How many sticks do you wish to take? (1~3) “)
  (assert (total (- ?num (read))))
  (assert (player-get c)))
```

```
(defrule computer-win
  (declare (salience 700))
  (player-get c)
  (total ?num)
  (test (< ?num 1))
 =>
  (printout t “You lose!” crlf))
```

```
(defrule Human-win
  (declare (salience 700))
  (player-get h)
  (total ?num)
  (test (< ?num 1))
 =>
  (printout t “You win!” crlf))
```

The Game of Sticks Revisited

```
(deftemplate choose (slot get) (slot remainder))

(deffacts initial
  (phase choose-total)
  (choose (get 1) (remainder 1))
  (choose (get 1) (remainder 2))
  (choose (get 2) (remainder 3))
  (choose (get 3) (remainder 0)))
```

```
(defrule choose-total-right
  ?phase <- (phase choose-total)
  (total ?total)
  (test (integerp ?total))
  (test (> ?total 0))
=>
  (retract ?phase)
  (assert (phase choose-player)))
```

```
(defrule choose-total
  (phase choose-total)
 =>
  (printout t “How many sticks in the pile? “)
  (assert (total  (read))))
```

```
(defrule choose-total-wrong
  ?phase <- (phase choose-total)
  ?total <- (total ?num&:(or (not (integerp ?num))
                             (<= ?num 0)))
=>
  (retract ?phase ?total)
  (assert (phase choose-total))
  (printout t “Please input a positive integer!!” crlf))
```

The Game of Sticks Revisited (cont.)

```
(defrule player-select
   (phase choose-player)
  =>
   (printout t “Who moves first  (Computer:c   Human: h)? “)
   (assert (player-get  (read))))
```

```
(defrule player-select-right
   ?phase <- (phase choose-player)
   (player-get ?player& c | h)
=>
   (retract ?phase)
   (assert (phase play-game)))
```

```
 (defrule player-select-wrong
   ?phase <- (phase choose-player)
   ?p-get <- (player-get ?player& ~c & ~h)
=>
   (retract ?phase ?p-get)
   (assert (phase choose-player))
   (printout t “Please input c or h!!” crlf))
```

The Game of Sticks Revisited (cont.)

```
(defrule human-get
  (phase play-game)
  (player-get h)
  (total ?num)
  (test (>= ?num 1))
 =>
  (printout t "There are " ?num " sticks in the pile." crlf)
  (printout t "How many sticks do you wish to take? (1~3) ")
  (assert (human-take (read))))
```

```
(defrule human-get-wrong
  (phase play-game)
  ?player <- (player-get h)
  ?h-take <- (human-take ?take)
  (total ?num)
  (test (or (not (integerp ?take)) (< ?take 1)  (> ?take 3)  (> ?take ?num)))
 =>
  (retract ?player ?h-take)
  (assert (player-get h))
  (printout t "Please input a number (1~3)!!" crlf))
```

```
(defrule human-get-right
  (phase play-game)
  ?player <- (player-get h)
  ?h-take <- (human-take ?take)
  ?total <- (total ?num)
  (test (and (integerp ?take)
             (>= ?take 1)
             (<= ?take 3)
             (<= ?take ?num)))
 =>
  (retract ?player ?h-take ?total)
  (assert (total (- ?num ?take)))
  (assert (player-get c)))
```

The Game of Sticks Revisited (cont.)

```
(defrule computer-get
   (phase play-game)
   ?player <- (player-get c)
   ?total <- (total ?num)
   (test (>= ?num 1))
   (choose (get ?get) (remainder =(mod ?num 4)))
=>
   (retract ?player ?total)
   (printout t  ?num " sticks in the pile." crlf)
   (assert (total (- ?num ?get)))
   (printout t "Computer take " ?get " sticks." crlf)
   (assert (player-get h)))
```

```
(defrule computer-win
   (player-get c)
   (total ?num)
   (test (< ?num 1))
  =>
   (printout t "You lose!" crlf))
```

```
(defrule Human-win
   (player-get h)
   (total ?num)
   (test (< ?num 1))
  =>
   (printout t "You win!" crlf))
```