Advanced CLIPS: Function and Control Structures

Prof. Khaled Shaalan

Khaled.shaalan@buid.ac.ae

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 in CLIPS) (> (/ (- y2 y1) (- x2 x1)) 0)

Return Values

- Most functions (addition) 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.
- Division results are usually rounded off.
- 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.

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.
```

Example Program 1: Summing Values Using Rules (rectangle.clp)

```
(deffacts sum-areas
    (rectangle 10 6)
    (rectangle 7 5)
    (rectangle 6 8)
    (rectangle 2 5)
    (sum 0) )
```

Process:

Given:

```
    (rectangle 10 6)
    (rectangle 7 5)
    (rectangle 6 8)
    (rectangle 2 5)
```

Results:

```
  (sum 153)
```

How?

compute-area

(rectangle 10 6) → **(add-to-sum 60)**

(rectangle 7 5) → **(add-to-sum 35)**

(rectangle 6 8) → **(add-to-sum 48)**

(rectangle 2 5) → **(add-to-sum 10)**

How?

sum-area

(add-to-sum 60)

(add-to-sum 35)

(sum 153)

(add-to-sum 48)

(add-to-sum 10)

Example Program 1: Summing Values Using Rules (rectangle.clp)

```
(deffacts sum-areas
    (rectangle 10 6)
    (rectangle 7 5)
    (rectangle 6 8)
    (rectangle 2 5)
    (sum 0) )

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

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

Does this work?

Yes!

```
CLIPS> (facts)
(rectangle 10 6)
(rectangle 7 5)
(rectangle 6 8)
(rectangle 2 5)
(sum 153)
```

Example Program 2: Summing Values Using Rules

```
(deffacts sum-areas
      (rectangle 10 6)
      (rectangle 7 5)
      (rectangle 6 8)
      (rectangle 2 5)
      (sum 0) )
(defrule compute-area
      (rectangle ?height ?width)
    =>
      (assert (add-to-sum  =(* ?height ?width))) )
(defrule sum-area
      ?sum <- (sum ?total)
      (add-to-sum ?area)
    =>
      (retract ?sum)
      (assert (sum =(+ ?total ?area))) )
```

Does this work?

Infinite Execution!

Intersection of Two Sets
(set_intersection.clp)

Structure for facts:

(set *<set-name>* *<set-elements>*)
(find-intersection *<set-name-1>* *<set-name-2>*)
(element *<set-name>* *<element>*); intermediate processing
(set-intersection *<set-elements>*) ; output of set intersection

Process:

Given:

(set s1 1 2 3 4)

(set s2 2 4 5)

Results:

(set-intersection 2 4)

How?

Disassemble S1

(set s1)

(element s1   1)

(element s1   2)

(element s1   3)

(element s1   4)

How?

Disassemble S2

(set s2)

(element s2 2)

(element s2 4)

(element s2 5)

How?

Intersection

(element s1 1)

(element s2 2)

(element s1 2)

(element s2 4)

(element s1 3)

(element s2 5)

(element s1 4)

(element intersection 2)

(element intersection 4)

How?

reassemble

(element intersection 2)

(element intersection 4)

(set-intersection 2 4)

Intersection of Two Sets
(set_intersection.clp)

Structure for facts:

(set *<set-name>* *<set-elements>*)
(find-intersection *<set-name-1>* *<set-name-2>*)
(element *<set-name>* *<element>*); intermediate processing
(set-intersection *<set-elements>*) ; output of set intersection

Rules:

```
(defrule  intersection
    (find-intersection ?set-1  ?set-2)
    ?s1  <-  (element  ?set-1  ?value)
    ?s2  <-  (element  ?set-2  ?value)
  =>
        (retract  ?s1  ?s2)
        (assert  (element  intersection  ?value)) )
```

Intersection of Two Sets (cont.)

```
(defrule disassemble
   (declare (salience 1))
   ?set <- (set ?name ?value $?elements)
  =>
   (retract ?set)
   (assert (set ?name $?elements))
   (assert (element ?name ?value)) )
(defrule reassemble
   (declare (salience 1))

   ?s1 <- (element intersection ?value)
   ?s2 <- (set-intersection $?elements)
  =>
   (retract ?s1 ?s2)
   (assert (set-intersection ?value $?elements)) )
```

Is salience required?

What Happens Now When We use these facts?

```
(deffacts sets
    (set  s1   1 2 3 4)
    (set  s2   2 4 5)
    (find-intersection  s1  s2)
    (set-intersection))
```

Output:

(set-intersection 2 4)

I/O Functions

Read

- Allows input of a single field

  (read *<logical-name>*)

- Encountering end-of-file returns EOF
- Example:

```
(defrule rule-1
    (initial-fact)
  =>
    (bind ?input (read))
    (assert (data ?input)) )
```

- **Alternative:**

```
(defrule rule-1
    (initial-fact)
  =>
    (assert (data =(read t))) )
```

Read Function from Keyboard

- The *read* function can only input a single field at a time.
- Characters entered after the first field up to the ↵ are discarded.
- Data must be followed by a carriage return to be read.

Example: Prompt & read input

```
(defrule  get-name
      =>
          (printout t “what is your name? “)
          (bind  ?response  (read))
          (assert  (user_name  ?response)) )
```

I/O Functions (cont.)

Readline

- Similar to read
- Input is an entire line and returns line as a string

(readline *<logical-name>*)

Example:

```
(defrule get-name
	=>
	(printout t "What is your name? "
	(bind ?response (readline))
	(assert (user_name ?response)))
CLIPS> (run)

My name is khaled shaalan ↵

CLIPS> (facts)

f-1 (user_name "My name is khaled shaalan")
```

Open and Close

Files must be opened before used for I/O and closed when finished

(open “*<file-name>*“ *<logical-name>* “*<mode>*“)

Modes:

R	read acess only (default)
W	write access only
R+	read and write access
A	append access only

(close *<logical-name>*)

Close with no arguments closes all opened files

Open and Close (cont.)

Example:

```
(defrule  read-from-file
          (initial-fact) =>
          (open  “data.clp”  my-data)
          (assert  (data1  =(readline  my-data)))
          (close) )
```

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))

Test Construct

- Often is more convenient than having a complex pattern

```
(defrule  example
   (data  ?d1)
   (data  ?d2)
   (data  ?d3)
   (test  (>  (min  ?d1  ?d2 ?d3)  .2))
  =>  . . . )
```

Note that this example has a problem!!

Problem with Example

```
(defrule  example
    (data  ?d1)
    (data  ?d2)
    (data   ?d3)
    (test  (>  (min   ?d1   ?d2   ?d3)   .2))
      =>   . . . )
```

Fact Base:

```
(data  .4)
(data  1.1)
(data  1.0)
```

Note that *EACH* fact can be matched by *EACH* pattern!!

What you really want to do:

```
(defrule  example2
    ?data1  <-  (data  ?d1)
    ?data2  <-  (data  ?d2)
    ?data2  <-  (data  ?d3)
    (test  (neq  ?data1  ?data2 ?data3)) ; neq is not equal
    (test  (>  (min  ?d1   ?d2 ?d3) .2))
  =>   . . . )
```

Control Facts

(control.clp)

An Alternative to Salience

```
(defrule  read-first-val
      ?init  <-  (initial-fact)
    =>
      (retract  ?init)
      (printout  t  “enter first value:”)
      (assert  (data  index  1  value  =(read)) )
      (assert  (context  read-input  phase  second-val)) )

(defrule  read-second-val
      ?context  <-  (context  read-input  phase  second-val)
    =>
      (retract  ?context)
      (printout  t  “enter second value:”)
      (assert  (cata  index  2  value  =(read))
      (assert  (context  read-input  phase  third-value)) )
```

Use of Salience for Phase Shift

```
(defrule  fault-detection-x
      (context fault-detection)
      . . .
    =>
      . . . )
(defrule  fault-detection-y
      (context fault-detection)
      . . .
    =>
      . . . )
(defrule  fault-detection-z
      (context fault-detection)
      . . .
    =>
      . . . )
```

```
(defrule  fault-detection-default
      (declare  (salience  -1))
      ?context  <-  (context fault-detection)
    =>
      (retract  ?context)
      (assert  (context analysis)) )
```

Explicit Logical Operators on LHS

- Default assumption is LHS conditions are joined by a logical AND

The *OR* Conditional Element

- Consider the two rules with similar conclusion:

```
(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))
```

Explicit Logical Operators on LHS
(cont.)

NOT

```
(defrule  definite-do-not-cross
        (light  green)
        (not  (do-not-cross  $?))
    =>
        (printout  t  “walk”  crlf)  )
```

- variables bound within a NOT may not be used outside the NOT
- Allows reasoning based on the absence of information which is a powerful reasoning technique

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))
```

Another “NOT” Example

```
(deffacts  initial-events
   (event  time  25  type  assign-order  order  o-1  machine  m-1)
   (event  time  30  type  assign-order  order  o-2  machine  m-1)
   (event  time  50  type  assign-order  order  o-3  machine  m-1)
   (clock  time  0) )

;; IF there are no remaining events to execute at current time
;; THEN update clock to next event’s time

(defrule  update-clock
    ?clock  <-  (clock  time  ?cur-time)
    (event  time  ?next  type  ?  $?)
    (not  (event  time  ?time&:(<  ?time  ?next)  type  ?  $?) )
   =>
    (retract  ?clock)
    (assert  (clock  time  ?next)) )
```

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 Conditional

```
(deftemplate emergency (slot type))

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

Exists

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

```
(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

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

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

Explicit Logical Operators on LHS
(cont.)

AND

```
(defrule  shut-off-electricity
      ?power  <-  (electrical-power  on)
      (or  ?reason  <-  (emergency  flood)
            (and  (emergency  fire)
                  ?reason  <-  (fire-class  c) )
            ?reason  <-  (sprinklers  active) )
  =>
   (retract  ?power)
   (assert  (electrical-power  off))
   (printout  t  “shut off electricity”  crlf
                  “emergency type: “  ?reason)  )
```

Procedural Constructs

IF-THEN-ELSE

```
(defrule  continue-check
    ?phase  <-  (phase  check-continue)
  =>
    (retract  ?phase)
    (printout  t  "continue (y/n)?"  crlf)
    (bind  ?answer  (read))
    (if  (or  (eq  ?answer  y)
              (eq  ?answer  yes) )
     then  (assert  (phase  continue))
     else  (halt)  )  )
```

Procedural Constructs (cont.)

WHILE

```
(defrule continue-check
    ?phase <- (phase check-continue)
  =>
    (retract ?phase)
    (bind ?loop yes)
    (while (eq ?loop yes)
           (bind ?loop no)
           (printout t "continue (y/n)?" crlf)
           (bind ?answer (read))
           (if (or (eq ?answer y)
                   (eq ?answer yes) )
            then (assert (phase continue))
            else (if (or (eq ?answer n)
                         (eq ?answer no))
                  then (halt)
                  else (bind ?loop yes) ) ) ) )
```

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>*)
```

Control string

- Must be delimited with quotes
- Consists of format flags to indicate how parameters should be printed
- 1 – 1 correspondence between flags and number of parameters – constant values or expressions
- Return value of the *format* function is the formatted string – *nil* can be used to suppress printing.
- format flags begin with % and have form:

  %-*M.Nx*

  \- optional, it specifies to left justify

  M field width (minimum)

  blanks used for padding unless *M* starts with 0

Format Statement (cont.)

N Optional number of digits past decimal point (default is 6 for floating point numbers)

x Display format:

d	integers	f	floating point
e	exponential	o	octal
g	general number (use shortest form)		
x	hexadecimal	s	string
n	crlf	%	% character

Example:

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

Produces the results:

“Name: Bob green   Age: 35”

Multi-field Functions

Following functions are useful with multiple field variables

length returns the number of elements:

(length $?X)

nth returns specified field:

(nth 3 $?X)

member is value in the variable:

(member ?atm $?X)

Multi-field Functions (cont.)

subset	is first a subset of the second: (subset $?Y $?X)
mv-delete	delete field: (mv-delete 5 $?X)
mv-append	append values to create a multifield: (mv-append $?X ?Y 123 X)
mv-subseq	return sub-sequence: (mv-subseq *<start>* *<end>* *<list>*)

String Functions

str-explode	build multifield from string: (str-explode *<string>*)
str-implode	build string from multifield: (str-implode *<list>*)
str-cat	build string from items: (str-cat *<itm1> <itm2> ... <itmn>*)
sub-string	returns sub-string: (sub-string *<start> <end> <string>*)
str-index	returns starting position: (str-index *<sub-string> <string>*)

Utility Commands

system allows execution of operating system level commands inside of CLIPS:

(system <*args*>)

(system "dir " ?directory)

(system cmd)

batch

allows top-level commands to be read (and executed) from a file:

(batch <*file-name*>)

Example file:

(load "rule-1.clp")
(load "rule-2.clp")
(reset)
(run)