

# Chapter 10: Procedural Programming

Expert Systems: Principles and  
Programming, Fourth Edition

# Procedural Function

---

Flow Control:

<i>if</i> function	<i>while</i> function
<i>switch</i> function	<i>loop-for-count</i> function
<i>progn\$</i> function	<i>break</i> function
<i>halt</i> function	

# *if* Function

---

General format:

```
(if <predicate-expression>  
  then <expression>+  
  [else <expression>+])
```

The predicate expression is first evaluated. If the condition is anything other than FALSE, the THEN clause actions are executed; otherwise, the ELSE actions are executed (unless omitted).

# *if* Function

---

- Example of using *if* function

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

# *while* Function

---

General format:

```
(while <predicate-expression> [do]  
  <expression>* )
```

The predicate expression is evaluated before actions of the body are executed. If the evaluation is anything other than FALSE, expressions in the body are executed; if FALSE, control passes to the next statement after the while.

# *while* Function

---

- Example of using *while* function

```
Defrule continue-check
  ?phase <- (phase check-continue)
=>
  (retract ?phase)
  (printout t "Continue? ")
  (bind ?answer (read))
  (while (and (neq ?answer yes) (neq ?answer no))
    do
      (printout t "Continue? ")
      (bind ?answer (read)))
  (if (eq ?answer yes)
    then (assert (phase continue))
    else (assert (phase halt))))
```

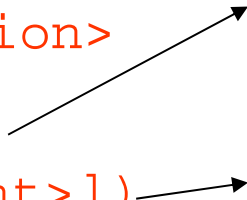
# *switch* Function

General format:

```
(switch <test-expression>  
  <case-statement>*  
  [ <default-statement> ] )
```

(case <comparison-expression>  
then <expression>\*)

(default <expression>\*)



Test-expression is evaluated first. Each comparison expression is then evaluated in order defined and if it matches the value of comparison expression, actions after *then* are executed, then termination. If no match, default statement actions are executed (if present).

# *switch* Function

---

- Example of using *switch* function

```
(defrule perform-operation
  (operation ?type ?arg1 ?arg2)
=>
  (switch ?type
    (case times then
      (printout t ?arg1 " times " ?arg2 " is " (* ?arg1 ?arg2) crlf))
    (case plus then
      (printout t ?arg1 " plus " ?arg2 " is " (+ ?arg1 ?arg2) crlf))
    (case minus then
      (printout t ?arg1 " minus " ?arg2 " is " (- ?arg1 ?arg2) crlf))
    (case divided-by then
      (printout t ?arg1 " divided by " ?arg2 " is " (/ ?arg1 ?arg2) crlf))
```

*Note: How to rewrite this rule without using the switch function?*



# *loop-for-count* Function

---

General format:

```
(loop-for-count <range-spec> [do] <expression*>)
```



```
<end-index> |  
(<loop-variable> <end-index>) |  
(<loop-variable> <start-index> <end-index>)
```

The body of the function is executed a given number of times, depending on <range-spec>. This is similar to the “*for*” loop structure in most high-level languages.

# *loop-for-count* Function

- Example of using *loop-for-count* function

```
(loop-for-count (?cnt1 2 4) do
  (loop-for-count (?cnt2 3) do
    (printout t ?cnt1 " ")
    (loop-for-count 3 do
      (printout t "."))
    (printout t " " ?cnt2 crlf)))
```

```
2 ... 1
2 ... 2
2 ... 3
3 ... 1
3 ... 2
3 ... 3
4 ... 1
4 ... 2
4 ... 3
```

```
(defrule masking-example
```

```
=>
```

```
(bind ?x 4)
(loop-for-count (?x 2) do
  (printout t "inside ?x is " ?x crlf))
(printout t "outside ?x is " ?x crlf))
```

```
Inside ?x is 1
Inside ?x is 2
Outside ?x is 4
```

# *progn\$ Function*

---

General format:

(progn\$ <list-spec> (expression)\* )

<list-spec> can be :

<multifield-expression> - body is executed once for each field

<list-variable> - field of current iteration is retrieved by referencing variable. Special variable is created by appending *-index* to <list-variable> - index of current iteration

# *progn\$ Function*

---

- Example of using *progn\$* function

```
(prog$ (create$ 1 2 3)  
  (print t . Crlf))
```

```
.  
.   
.
```

```
(progn$ (?v (create$ a b c))  
  (printout t ?v-index “ “ ?v crlf))
```

```
1 a  
2 b  
3 c
```

# *break* Function

---

General format:

( *break* )

The *break* function terminates the execution of the *while*, *loop-for-count*, or *progn*\$ function in which it is immediately contained. It causes early termination of a loop when a specified condition has been met.

# *break* Function

---

Example of using *break* function:

```
(defrule print-list
  (print-list $?list)
=>
  (progn$ (?v ?list)
    (if (<= ?v-index 5)
      then (printout t ?v " ")
      else (printout t "...")
      (break)))
  (printout t crlf))

(assert (print-list a b c d e))
Result will be: a b c d e
(assert (print-list a b c d e f g h))
Result will be: a b c d e ...
```

# *halt* Function

---

General format:

(halt)

The *halt* function can be used on the RHS of a rule to stop execution of rules on the agenda. When called, no further actions will be executed from the RHS of the rule being fired and control returns to the top-level prompt.

# *halt* Function

---

Example of using *halt* function:

```
(defrule continue-check
  ?phase <- (phase check-continue)
=>
  (retract ?phase)
  (printout t "Continue? ")
  (bind ?answer (read))
  (while (and (neq ?answer yes) (neq ?answer no))
    do
      (printout t "Continue? ")
      (bind ?answer (read)))
  (assert (phase continue))
  (if neq ?answer yes
    then (halt)))
```



# The Deffunction Construct

- CLIPS allows one to define functions similar to the way it is done in procedural languages.
- New functions are defined using the *deffunction* construct:

```
(deffunction <deffunction-name> [<optional-comment>]  
  (<regular-parameter>* [<wildcard-parameter>])  
  <expression>*)
```

↑

the body of the deffunction

Single-field variable

Multifield variable

# Deffunction

---

- The body of the deffunction is a series of expressions similar to the RHS of a rule that are executed in order when the deffunction is called.
- Deffunctions can be deleted and the *watch* command can be used to trace their execution.

# Deffunction

---

- The <regular-parameter> and <wildcard-parameter> declarations allow you to specify the arguments that will be passed to the deffunction when it is called.
- A deffunction can return values – value returned is the value of the last expression evaluated within the body of the deffunction.

# Deffunction

---

- Function for computing the length of the hypotenuse:

```
(deffunction hypotenuse-length (?a ?b)
  (** (+ (* ?a ?a) (* ?b ?b)) 0.5))
```

Rewrite in a more readable form with local variable:

```
(deffunction hypotenuse-length (?a ?b)
  (bind ?temp (+ (* ?a ?a) (* ?b ?b)))
  (** ?temp 0.5))
```

# *return* Function

---

General format:

```
(return [<expression>])
```

In addition to terminating the execution of the RHS of a rule, the *return* function also allows the currently executing deffunction to be terminated.

# *return* Function

---

Function with return:

(deffunction hypotenuse-length (?a ?b)  
 (bind ?temp (+ (\* ?a ?a) (\* ?b ?b)))  
 (return (\*\* ?temp 0.5)))

or

(deffunction hypotenuse-length (?a ?b)  
 (bind ?temp (+ (\* ?a ?a) (\* ?b ?b)))  
 (bind ?c (\*\* ?temp 0.5))  
 (return ?c))

Return is useful when a condition is satisfied that should terminate the function

```
(deffunction primep (?num)
  (loop-for-count (?i 2 (- ?num 1))
    (if (= ?num (* (div ?num ?i) ?i))
      then
        (return FALSE)))
  (return TRUE))
```

# Deffunction

- How to use deffunction to replace the validation rules in stick problem?

```
(deffacts initial-phase
  (phase choose-player))
```

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

```
(defrule good-player-choice
  ?phase <- (phase choose-player)
  ?choice <- (player-select ?player&c | h)
=>
  (retract ?phase ?choice)
  (assert (player-move ?player))
  (assert (phase select-pile-size)))
```

```
(defrule bad-player-choice
  ?phase <- (phase choose-player)
  ?choice <- (player-select ?player&~c&~h)
=>
  (retract ?phase ?choice)
  (assert (phase choose-player))
  (printout t "Choose c or h." crlf))
```

# Deffunction

---

```
(deffunction check-input (?question ?values)
  (printout t ?question " " ?values " ")
  (bind ?answer (read))
  (while (not (member$ ?answer ?values))
    (printout t ?question " " ?values " ")
    (bind ?answer (read)))
  (return ?answer))
```

## **Using this function:**

```
(check-input "Who moves first, Computer or Human?" (create$ c h))
```

## **The modified version:**

```
(defrule player-select
  ?f <- (phase choose-player)
```

=>

```
  (retract ?f
    (bind ?player (check-input "Who moves first, Computer or human?"
      (create$ c h)))
    (assert (player-move ?player)))
```



# Recursions and Forward Declarations

---

- Deffunctions can call other functions within their body as well as themselves – a recursive call.
- Sometimes functions make circular references to one another. To accomplish this:
  - A forward declaration is done
  - The name and argument list of the function are declared
  - The function body is left empty
  - The forward declaration can be replaced later with a version that includes the body

# Watching Deffunctions

---

- When deffunctions are watched using the *watch* command, an informational message is printed whenever a deffunction begins or ends execution.
- The >> symbol means that a deffunction is being entered, and the << symbol means it is being exited.

# Wildcard Parameters

---

- If all arguments in the parameter list of a deffunction are single-field variables:
  - 1-1 correspondence between these parameters and the number of arguments that must be passed when the deffunction is called.
- If the last parameter declared in a deffunction is a multifield variable (wildcard parameter)
  - Deffunction can be called with more arguments than are specified in the parameter list.

# Deffunction Commands

---

1. *ppdeffunction* – displays the text representation of a deffunction
2. *pndeffunction* – used to delete a deffunction
3. *list-deffunction* – displays the list of deffunctions defined.
4. *get-deffunction-list* – returns a multifold value containing the list of deffunctions

# User-Defined Function

---

User-defined functions are functions written in the C language and called from CLIPS.

# The Defglobal Construct

---

- CLIPS allows one to define *global variables* that retain their values outside the scope of a construct.
- *Local variables* are local to the construct that references them.

```
(defrule example-1  
  (data-1 ?x)  
=>  
  (printout t "?x=" ? crlf))
```

```
(defrule example-2  
  (data-2 ?x)  
=>  
  (printout t "?x=" ? crlf))
```

?x in example-1 does not contain the value of ?x in example-2

# Defining Global Variables

---

The *defglobal construct* has the following format:

```
(defglobal [<defmodule-name>] <global-assignment>*)
```

Defmodule-name = module in which globals will be defined (current if omitted)

Global-assignment = <global-variable> = <expression>

Expression = provides the initial value for the global variable

Global-variable = ?\*<symbol>

```
(defglobal ?*x* = 3
          ?*y* = (+ ?*x* 1))
```

# Global Variables

- By entering the global variable's name at the command line, one can see its value. `CLIPS>?*x*`  
3
- References to such variables can be made anywhere it is legal to use an `<expression>`.
- They cannot be used as a parameter of a deffunction.  
`(deffunction illegal-1 (*x* ?y)  
 (+ ?x* y))`
- Can only be used on the LHS of a rule if contained within a function call.  
`(defrule illegal-2 (data-1 ?x*) =>  
 (defrule illegal-3 (data-1 ?x&~?*x*) =>)`



# Global Commands

---

1. *ppdefglobal* – displays text representation of a defglobal
2. *undefglobal* – deletes a defglobal
3. *list-defglobals* – displays list of defglobals defined in CLIPS
4. *show-defglobals* – displays names and values of defglobals defined in CLIPS
5. *get-defglobal-list* – returns multifield value containing the list of defglobals.

# The Use of Global Variables

---

- *reset* command can be used to reset the value of a global variable to its origin defined value.
- The reset behavior can be disabled by calling the *set-reset-globals* command.
  - e.g. (set-reset-globals FALSE)
- Defglobals and pattern matching
  - Consider the following:  
(defglobal ?\*z\* = 4)  
(defrule global-example  
 (data ?z&:(> ?z ?\*z\*))  
 =>)
  - Try to assert facts to activate the rule and reassign value to the global variable

# The Use of Global Variables

---

- Use defglobals to define a meaningful symbols
  - e.g. define `?*water-freezing-point-Fahrenheit*` = 32 instead of using the value 32 directly.
- Use defglobals to help debugging
  - e.g.  

```
(defglobal?*debug-print* = nil)  
(defrule debug-example  
  (data ?x)  
=>  
  (printout?*debug-print* "Debug-example ?x = " ?x crlf))
```

# The Defgeneric and Defmethod Constructs

---

- **Generic functions** is a group of related functions sharing a common name.
- **Overloaded functions** are those w/more than one method.
- When called, the method w/signature matching is the one executed – process called **generic dispatch**

# Defgeneric / Defmethod

---

- General formats:

```
(defgeneric <defgeneric-name>  
  [<optional-comment>])
```

```
(defmethod <defgeneric-name> [index]  
  [<optional-comment>  
   [<regular-parameter-restriction>*  
    [<wildcard-parameter-restriction>]]  
  <expression>*)
```

# Defmethods

---

- Specific defmethods do not have names but are assigned a unique integer index that can be used to reference the method.
- You can assign a specific index or CLIPS will assign one for you.
- For a given generic function, each method must have a unique signature from the other methods in that generic function.

# Regular Parameter Restrictions

---

Each `<regular-parameter-restriction>` can be one of two forms:

- A single-field variable (as in a deffunction), or
- Of the form `( <single-field-variable>  
<type>* [ <query> ] )`

# Defmethods

- Revisit the check-input function. What if we use it for player to input the number of sticks he/she wants to pick from the pile?

- (check-input “How many sticks do you wish to take?” 1 2 3)

- A revised version is possible

```
(defmethod check-input ((?question STRING) (?value1 INTEGER) (?value2 INTEGER))
  (printout t ?question “ (“ ?value1 “-” ?value2 “) “)
  (bind ?answer (read))
  (while (or (not (integerp ?answer)) (< ?answer ?value1) (> ?answer ?value2))
    (printout t ?question “ (“ ?value1 “-” ?value2 “) “)
    (bind ?answer (read)))
  (return ?answer))
```

New version:

```
(check-input “Pick a number” 1 3)
Pick a number (1-3)
```

Previous version:

```
(check-input “Pick a number” 1 2 3)
Pick a number (1 2 3)
```



# Method Precedence

---

- When a generic function is called, CLIPS executes only one of the methods.
- Generic dispatch is the process of determining which method will execute.
- CLIPS defines a precedence order to the methods defined for a particular method.
- The *preview-generic* command displays the precedence order.

```
CLIPS> (preview-generic check-input "Pick a number" 1 3)
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #1 () $?
```

# Precedence

---

Given two methods of a generic function, CLIPS uses the following steps to determine which method has the higher precedence – see Fig. 10.1.

1. Compare the leftmost unexamined parameter restrictions of the two methods. If only one method has parameters left, proceed to step 6. If neither method has parameters remaining, proceed to step 7. Otherwise, proceed to step 2.

# Precedence

---

2. If one parameter is a regular parameter and the other parameter is a wildcard parameter, then the method with the regular parameter has precedence. Otherwise, proceed to step 3.
3. If one parameter has type restrictions and the other parameter does not, then the method with the types restrictions has precedence. Otherwise, proceed to step 4.
4. Compare the leftmost unexamined type restriction of the two parameters. If the type restrictions on one parameter is more specific than on the other, the method with more restrictions has precedence.

# Precedence

---

5. If one parameter has a query restriction and the other does not, then the method with the query restriction has precedence. Otherwise return to step 1 and compare the next set of parameters.
6. If the next parameter of the method with parameters remaining is a regular parameter, then this method has precedence. Otherwise, the other method has precedence.
7. The method that was defined first has precedence.

# Precedence

---

#3

```
(defmethod check-input ((?question STRING) (?value1 NUMBER) (?value2 NUMBER))
```

```
CLIPS> (preview-generic check-input "Pick a number" 1 3)
```

```
check-input #2 (STRING) (INTEGER) (INTEGER)
```

```
check-input #3 (STRING) (NUMBER) (NUMBER)
```

```
check-input #1 () $?
```

#4

```
(defmethod check-input ((?question STRING)
                        (?value1 INTEGER FLOAT) (?value2 INTEGER FLOAT))
```

```
CLIPS> (preview-generic check-input "Pick a number" 1 3)
```

```
check-input #2 (STRING) (INTEGER) (INTEGER)
```

```
check-input #4 (STRING) (INTEGER FLOAT) (INTEGER FLOAT)
```

```
check-input #3 (STRING) (NUMBER) (NUMBER)
```

```
check-input #1 () $?
```

# Query Restrictions

---

- A query restriction is a defglobal reference or function that is evaluated to determine the applicability of a method when a generic function is called.
- If the query restriction evaluates to false, the method is not applicable.
- The query restriction for a parameter is not evaluated unless the type restrictions for that parameter are satisfied.
- A method w/multiple parameters can have multiple query restrictions and each must be satisfied for the method to be applicable.

# Query Restrictions

---

CLIPS>(check-input "Pick a number")

Pick a number () 3

Pick a number () 2

...

..

.

#5

```
(defmethod check-input ((?question ($?values (= (length$ ?values) 0)))  
  (return FALSE))
```

CLIPS> (preview-generic check-input "Pick a number" )

check-input #5 () (\$? <qry>)

check-input #1 () \$?

← Cause a endless loop!

# Query Restrictions

CLIPS>(check-input "Pick a number" 3 1)

Pick a number (3-1) 3

Pick a number (3-1) 2

...

..

#6

```
(defmethod check-input ((?question STRING) (?value1 INTEGER)
                        (?value2 INTEGER (< ?value2 ?value1)))
  (return FALSE))
```

or

```
(defmethod check-input ((?question STRING) (?value1 INTEGER)
                        (?value2 INTEGER (< ?value2 ?value1)))
  (check-input ?question ?value2 ?value1))
```

CLIPS> (preview-generic check-input "Pick a number" 3 1)

check-input #6 (STRING) (INTEGER) (INTEGER <qry>)

check-input #2 (STRING) (INTEGER) (INTEGER)

Cause a endless loop!





# Defmethod Commands

---

1. *ppdefmethod* – displays the text representation of a defmethod
2. *undefmethod* – deletes a defmethod
3. *list-defmethods* – displays the list of defmethods defined in CLIPS
4. *get-defmethod-list* – returns a multifield value containing the list of defmethods

# Defgeneric Commands

---

1. *ppdefgeneric* – displays the text representation of a defgeneric
2. *undefgeneric* – deletes a defgeneric
3. *list-defgenerics* – displays the list of defgenerics defined in CLIPS
4. *get-defgeneric-list* – returns a multifield value containing the list of defgenerics

# Procedural Constructs and Defmodules

---

- Similar to deftemplate constructs, defglobal, and defgeneric constructs can be imported and exported by modules.
- Four of the export/import statements previously discussed apply to procedural constructs:
  1. `(export ?ALL)`
  2. `(export ?NONE)`
  3. `(import <module-name> ?ALL)`
  4. `(import <module-name> ?NONE)`

# Procedural Constructs and Defmodules

---

1. This format will export all exportable constructs from a module.
2. Indicates that no constructs are exported.
3. Imports all exported constructs from the specified module.
4. Imports none of the exported constructs from the specified module.

# Useful Commands / Functions

---

1. **load-facts / save-facts** – allows facts to be loaded from / saved to a file.
2. **system** command – allows execution of O/S commands within CLIPS – not all O/S provides functionality for this command.
3. **batch** command – allows commands and responses normally entered at the top-level to be read from a file.
4. **dribble-on / dribble-off** – session capture

# Commands / Functions

---

5. *random* – generates random integer values

```
(deffunction roll-die ()  
  (random 1 6))
```

6. *string-to-field* – converts a string-field value into a field.

7. *apropos* command – displays all symbols defined in CLIPS containing a specific substring (apropos deftemplate)

8. *sort* function – sorts a list of fields

```
(deffunction string> (?a ?b)  
  (> (str-compare ?a ?b) 0))
```

```
CLIPS> (sort > 4 3 5 7 2 7)  
(2 3 4 5 7 7)
```

```
CLIPS> (sort string> ax aa bk mn ft m)  
(aa ax bk ft m mn)
```

# Summary

---

- The *if*, *while*, *switch*, *loop-for-count*, *progn*\$, and *break* functions can alter the flow in a *deffunction*, *generic function*, or on the RHS of a rule – overuse is considered poor programming.
- The *deffunction construct* allows for defining new functions (similar to procedural languages)
- The *defglobal construct* allows one to define global variables that retain values outside the scope of constructs.

# Summary

---

- **Generic functions** (w/defgeneric / defmethod) provide more power and flexibility than deffunctions by associating multiple procedural methods with a generic function name.
- When generic functions are called, **generic dispatch** examines the types of arguments passed to the function and evaluates any associated query restrictions to determine the appropriate method to invoke.



# Summary

---

- Several utility commands were introduced, including:

<code>save-facts</code>	<code>load-facts</code>
<code>system</code>	<code>batch</code>
<code>dribble-on/off</code>	<code>random</code>
<code>sort</code>	<code>apropos</code>
<code>string-to-field</code>	