

Artificial Intelligence

Section 3

Clips

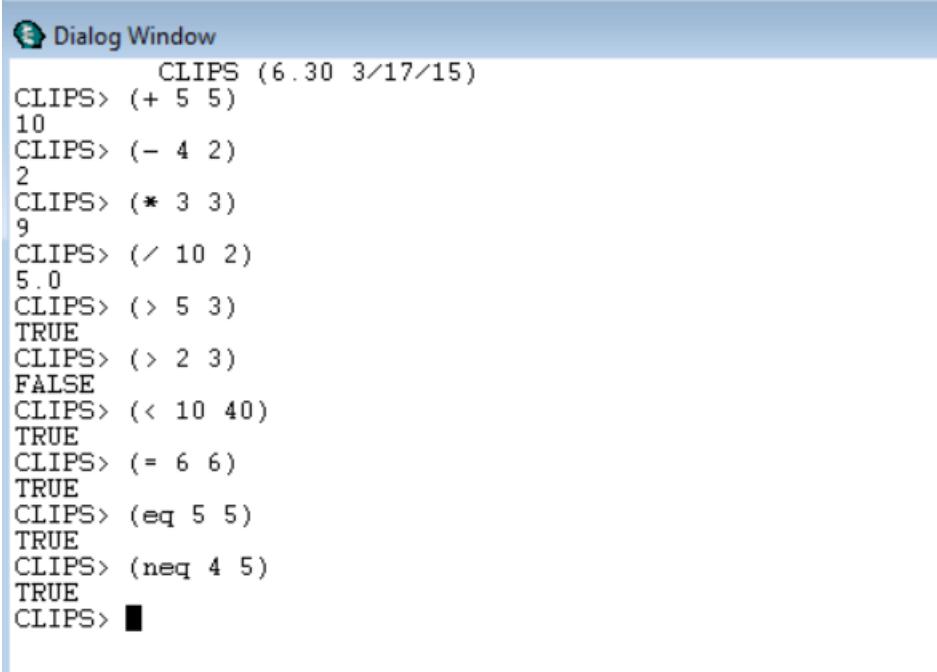
i. What is clips?

1. **CLIPS (C Language Integrated Production System)** is a public-domain software tool for building Expert Systems.
2. **CLIPS is a multiparadigm programming language** that provides support for (Rule-based – Object-oriented – Procedural programming).
3. **CLIPS** supports only forward-chaining rules.
4. **CLIPS** is case sensitive.

■ Core Components of a CLIPS Expert System:

- **Knowledge Base:** Contains the rules and objects that define the system's expertise.
- **Fact-List:** A global memory where data and information are stored as facts.
- **Inference Engine:** The component that controls the execution of rules, deciding which rules to activate and when.

ii. Operations in clips



The screenshot shows a window titled "Dialog Window" with the title bar "CLIPS (6.30 3/17/15)". The main area contains the following CLIPS session transcript:

```
CLIPS> (+ 5 5)
10
CLIPS> (- 4 2)
2
CLIPS> (* 3 3)
9
CLIPS> (/ 10 2)
5.0
CLIPS> (> 5 3)
TRUE
CLIPS> (> 2 3)
FALSE
CLIPS> (< 10 40)
TRUE
CLIPS> (= 6 6)
TRUE
CLIPS> (eq 5 5)
TRUE
CLIPS> (neq 4 5)
TRUE
CLIPS> ■
```

iii. Facts in clips:

1) Ordered Facts:

- a) Using assert ()
- b) Using deffacts ()

The screenshot shows the CLIPS IDE interface. On the left, the 'Dialog Window' contains the CLIPS command: `(assert (a 5) (b 10))`. The response shows two facts being asserted: `f-1 (a 5)` and `f-2 (b 10)`. On the right, the 'Facts (MAIN)' window displays the asserted facts: `f-0 (initial-fact)`, `f-1 (a 5)`, and `f-2 (b 10)`.

The screenshot shows the CLIPS IDE interface. On the left, the 'Dialog Window' contains the CLIPS commands: `(deffacts f1
(a 30)
(a 40)
(c 40))` and `(reset)`. The response shows facts being defined and reset: `f-0 (initial-fact)`, `f-1 (initial-fact)`, `f-2 (a 30)`, `f-3 (a 40)`, and `f-4 (c 50)`. On the right, the 'Facts (MAIN)' window displays the facts: `f-0 (initial-fact)`, `f-1 (a 30)`, `f-2 (a 40)`, and `f-3 (c 50)`.

The screenshot shows the CLIPS IDE interface. On the left, the 'Dialog Window' contains the CLIPS command: `Loading Selection...`. The response shows facts being defined and reset: `f-0 (initial-fact)`, `f-1 (initial-fact)`, `f-2 (a 10)`, and `f-3 (b 20)`. On the right, the 'Facts (MAIN)' window displays the facts: `f-0 (initial-fact)`, `f-1 (a 10)`, and `f-2 (b 20)`. Below the Dialog Window, there is an 'Untitled1' editor window containing the CLIPS code: `(deffacts f1
(a 10)
(b 20)
)`.

iv. Q3. Create rules in clips:

The screenshot shows the CLIPS environment. On the left, the 'Dialog Window' displays the command-line interaction: CLIPS> Loading Selection... Defining defrule: r2 =j+j CLIPS>. Below it, the 'Untitled1' editor window contains the CLIPS rule definition:

```
(defrule r2
  (a 10)
  =>
  (printout t "True" crlf))
```

On the right, the 'Agenda (MAIN)' window shows the agenda with two entries:

0	r2: f-1
0	r1: f-1

The screenshot shows the CLIPS environment. On the left, the 'Dialog Window' displays the command-line interaction: CLIPS> Loading Selection... Defining defrule: r2 =j+j CLIPS> (run) FIRE 1 r2: f-1 True. Below it, the 'Untitled1' editor window contains the same CLIPS rule definition as before.

On the right, the 'Facts (MAIN)' window shows the initial facts:

f-0	(initial-fact)
f-1	(a 10)
f-2	(b 20)

v. Print values of facts using rules:

The screenshot shows the CLIPS environment. On the left, the 'Dialog Window' displays the command-line interaction: CLIPS> Loading Selection... Defining deffacts: f1 CLIPS> (reset) <== f-0 (initial-fact) ==> f-0 (initial-fact) ==> f-1 (a 10) ==> f-2 (b 20) ==> f-3 (c 5) CLIPS> Loading Selection... Defining defrule: r1 +j+j+j CLIPS>. Below it, the 'Untitled1' editor window contains the CLIPS code:

```
(deffacts f1
  (a 10)
  (b 20)
  (c 5)
  )

(defrule r1
  (a ?x)
  (b ?y)
  =>
  (printout t ?x " " ?y crlf))
```

On the right, the 'Agenda (MAIN)' window shows the agenda with one entry:

0	r1: f-1, f-2
<	>

Dialog Window

```

CLIPS> Loading Selection...
Defining deffacts: f1
CLIPS> (reset)
<== f-0      (initial-fact)
==> f-0      (initial-fact)
==> f-1      (a 10)
==> f-2      (b 20)
==> f-3      (c 5)
CLIPS> Loading Selection...
Defining defrule: r1 +j+j+j
CLIPS> (run)
FIRE    1 r1: f-1,f-2
10 20
CLIPS>

```

Facts (MAIN)

```

f-0      (initial-fact)
f-1      (a 10)
f-2      (b 20)
f-3      (c 5)

```

Untitled1

```

(deffacts f1
(a 10)
(b 20)
(c 5)
)

(defrule r1
(a ?x)
(b ?y)
=>
(printout t ?x " " ?y crlf)
)

```

Dialog Window

```

CLIPS> (run)
FIRE    1 r2: f-6,f-4,f-5
5 50 5
FIRE    2 r2: f-6,f-3,f-5
5 90 5
FIRE    3 r2: f-2,f-3,f-5
50 90 5
FIRE    4 r2: f-1,f-3,f-5
10 90 5
FIRE    5 r2: f-2,f-4,f-5
50 50 5
FIRE    6 r2: f-1,f-4,f-5
10 50 5
FIRE    7 r1: f-6
Yes
CLIPS>

```

Facts (MAIN)

```

f-0      (initial-fact)
f-1      (a 10)
f-2      (a 50)
f-3      (b 90)
f-4      (b 50)
f-5      (c 5)
f-6      (a 5)

```

Untitled2

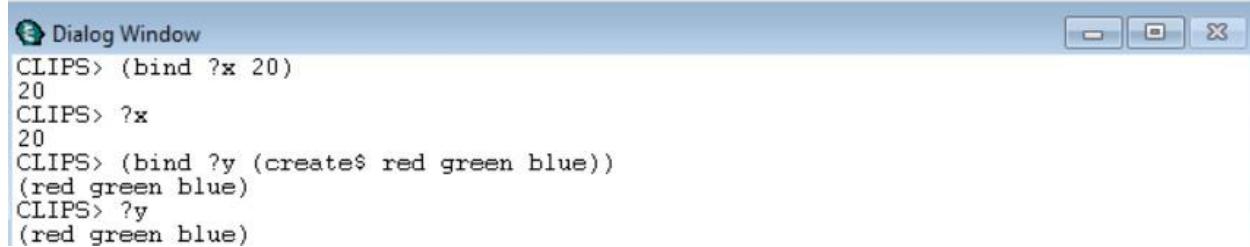
```

(defrule r1
(a 5)
=>
(printout t "Yes" crlf) )

(defrule r2
(a ?x)
(b ?y)
(c ?z)
=>
(printout t ?x " " ?y " " ?z " " crlf)
)

```

vi. Add value to variable using bind:

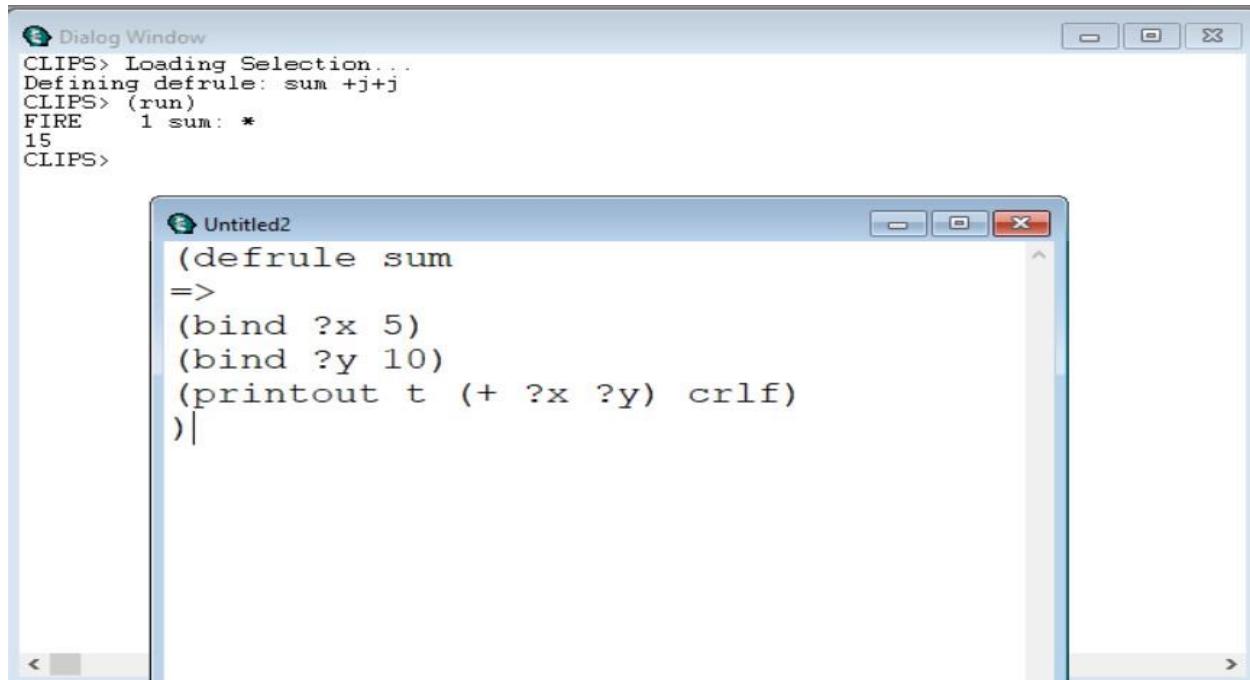


```
Dialog Window
CLIPS> (bind ?x 20)
20
CLIPS> ?x
20
CLIPS> (bind ?y (create$ red green blue))
(red green blue)
CLIPS> ?y
(red green blue)
```

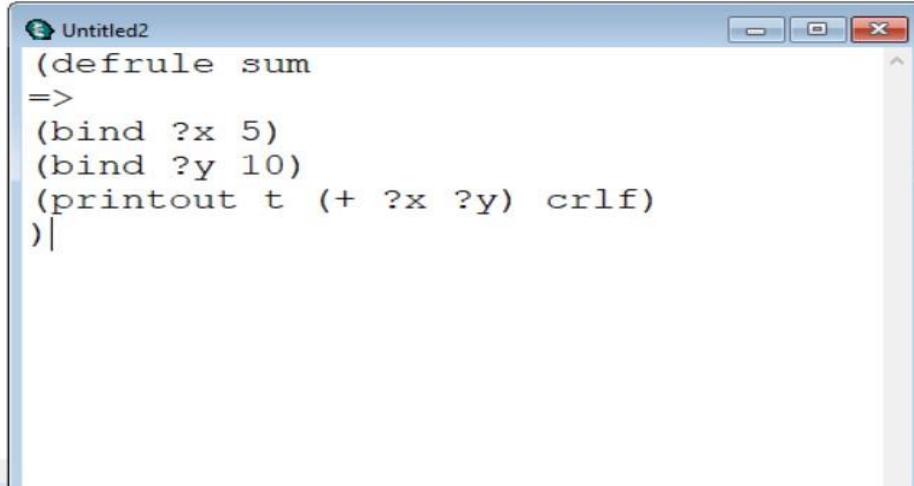
vii. Add value using global variable:

```
CLIPS> (defglobal ?*a* = 10)
CLIPS> ?*a*
10
CLIPS> (defglobal ?*b* = (create$ 5 10 15))
CLIPS> ?*b*
(5 10 15)
CLIPS> (printout t "Expert System" crlf)
Expert System
CLIPS> █
```

viii. sum two numbers:



```
Dialog Window
CLIPS> Loading Selection...
Defining defrule: sum +j+j
CLIPS> (run)
FIRE    1 sum: *
15
CLIPS>
```

```
Untitled2
(defrule sum
=>
(bind ?x 5)
(bind ?y 10)
(printout t (+ ?x ?y) crlf)
)|
```

ix. Using rules to create loop:
We use built-in function called (loop-for-count)

The screenshot shows the CLIPS 6.3 interface. On the left, the 'Dialog Window' displays the command: `(defrule r1 => (loop-for-count(?n 1 5) do(printout t ?n crlf)))`. On the right, the 'CLIPS>' prompt shows the execution of the rule with the output: `1
2
3
4
5`.

x. Using rules to show facts:

The screenshot shows the CLIPS 6.3 interface. The 'Dialog Window' on the left shows the creation of a fact database: `(deffacts f1 (a 1) (a 2) (a 3))` and a rule: `(defrule r1 (a ?f) => (printout t ?f crlf))`. The 'Facts (MAIN)' window on the right lists the facts: `f-0 (initial-fact)
f-1 (a 1)
f-2 (a 2)
f-3 (a 3)`.

xi. Using rules to summation two numbers:

The screenshot shows the CLIPS 6.3 interface. The 'Dialog Window' on the left shows the creation of a fact database: `(defrule sum => (printout t "Enter first number" crlf) (bind ?n (read)) (printout t "Enter second number" crlf) (bind ?m (read)) (printout t (+ ?n ?m) crlf))`. The 'CLIPS>' prompt shows the rule being defined.

xii. Dividing Two numbers using Rule:

The screenshot shows the CLIPS 6.3 interface. At the top is a menu bar with File, Edit, Buffer, Execution, Browse, Window, and Help. Below the menu is a toolbar with icons for New, Open, Save, Print, and Help. A 'Dialog Window' is open on the left, titled 'CLIPS> Loading Selection...', showing the rule definition: '(defrule divide ...)'. The main window is titled 'Untitled' and contains the rule code: '(defrule divide => (printout t "Enter first number" crlf) (bind ?n (read)) (printout t "Enter second number" crlf) (bind ?m (read)) (if(<> ?m 0) then (printout t (/ ?n ?m) crlf) else (printout t "Error: divide by zero" crlf)))'. To the right of the code, there is a command-line interface window titled 'CLIPS>' where the user has entered '50' and '5'.

xiii. Multi-field functions in clips:

- Creating multi-field values (**create\$**)
Ex: (create\$ red green blue)
- Specifying multi-field values (**nth\$**)
Ex: (nth\$ 2 (create\$ 30 40 50))
40
Ex: (nth\$ 4 (create\$ 30 40 50))
nil
- Finding an element multi-field value (**member\$**)
Ex: (member\$ yellow (create\$ red green blue))
FALSE
Ex: (member\$ green (create\$ red green blue))
2

xiv. Predicate functions:

- **lexemep**=> check if the argument is the symbol or string return true, otherwise return false.
Ex: (lexemep Asmaa)=>true
Ex: (lexemep "Mohamed")=>true
Ex: (lexemep 10)=>false
- **symbolp**=> check if the argument is the symbol return true, otherwise it will return false.
Ex: (symbolp Ahmed)=> true
Ex: (symbolp “Ahmed”)=> false

- **wordp** => check if the argument is the symbol return true, otherwise return false.

Ex: (wordp Asmaa) =>true

- **evenp** => check if the argument is even return true, otherwise return false.

Ex: (evenp 6) => true

Ex: (evenp 5) => false

- **oddp** => check if the argument is odd return true, otherwise return false.

Ex: (oddp 3) => true

Ex: (oddp 4) => false

- **Numberp** =>return the symbol true if its argument is a float or integer, otherwise return false

Ex: (numberp 10) => true

Ex: (numberp -5) => true

Ex: (numberp Cairo) => false

- **floatp** => return the symbol true if its argument is a float, otherwise return false

Ex: (floatp 5.3) => true

Ex: (floatp -4.6) => true

Ex: (floatp "hello") => false

- **integerp** => return the symbol true if its argument is integer, otherwise return false

Ex: (integerp 5) => true

Ex: (floatp "hello") => false

- **multifieldp** => function returns the symbol true if its argument is a multifield value, otherwise return false.

Ex: (multifieldp M) => false

Ex: (multifieldp (create\$ a b c d)) => true

Ex: (sequencep (create\$10 20 30 40)) => true

xv. comparing equality and not equality

Ex: (eq M N) => false

Ex: (eq A A) => true

Ex: (eq 3 3 3) => true

Ex: (neq 10 20) => true

Ex: (neq 5 5) => false

Ex: (neq A B) => true

Ex: (neq A a) => true