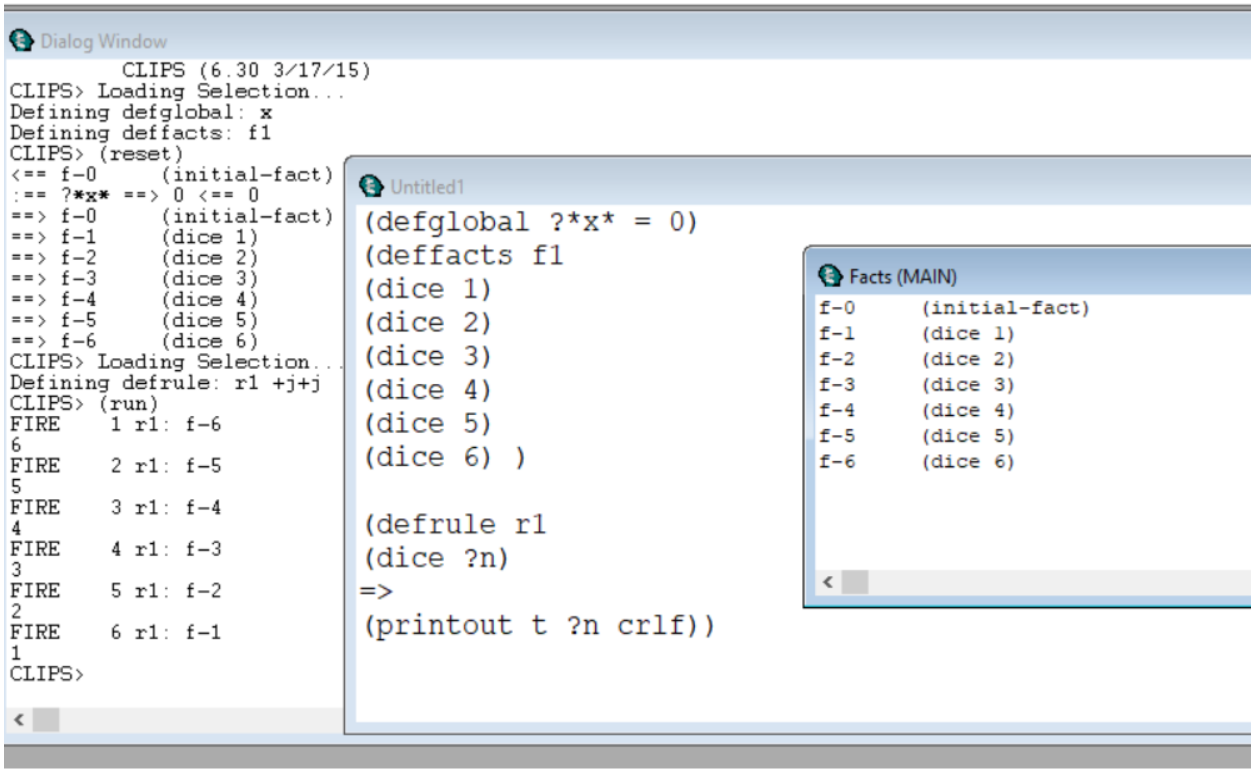
Q1. Write a construct /rule to print all faces of the dice:



Dialog Window

```
           CLIPS (6.30 3/17/15)
CLIPS> Loading Selection...
Defining defglobal: x
Defining deffacts: f1
CLIPS> (reset)
<== f-0     (initial-fact)
:== ?*x* ==> 0 <== 0
==> f-0     (initial-fact)
==> f-1     (dice 1)
==> f-2     (dice 2)
==> f-3     (dice 3)
==> f-4     (dice 4)
==> f-5     (dice 5)
==> f-6     (dice 6)
CLIPS> Loading Selection...
Defining defrule: r1 +j+j
CLIPS> (run)
FIRE    1 r1: f-6
6
FIRE    2 r1: f-5
5
FIRE    3 r1: f-4
4
FIRE    4 r1: f-3
3
FIRE    5 r1: f-2
2
FIRE    6 r1: f-1
1
CLIPS>
```

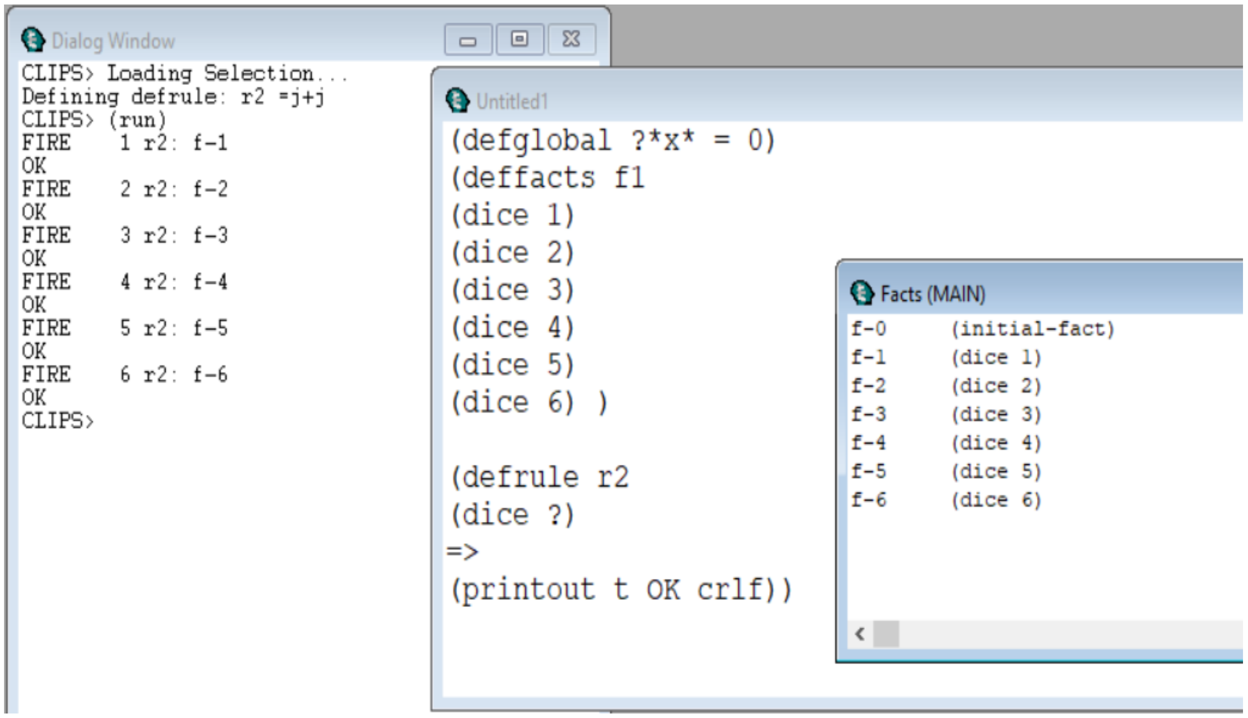
Untitled1

```
(defglobal ?*x* = 0)
(deffacts f1
(dice 1)
(dice 2)
(dice 3)
(dice 4)
(dice 5)
(dice 6) )

(defrule r1
(dice ?n)
=>
(printout t ?n crlf))
```

Facts (MAIN)

```
f-0     (initial-fact)
f-1     (dice 1)
f-2     (dice 2)
f-3     (dice 3)
f-4     (dice 4)
f-5     (dice 5)
f-6     (dice 6)
```

Q2. Write a rule to check all values if it presents in facts:



Dialog Window

```
CLIPS> Loading Selection...
Defining defrule: r2 =j+j
CLIPS> (run)
FIRE    1 r2: f-1
OK
FIRE    2 r2: f-2
OK
FIRE    3 r2: f-3
OK
FIRE    4 r2: f-4
OK
FIRE    5 r2: f-5
OK
FIRE    6 r2: f-6
OK
CLIPS>
```

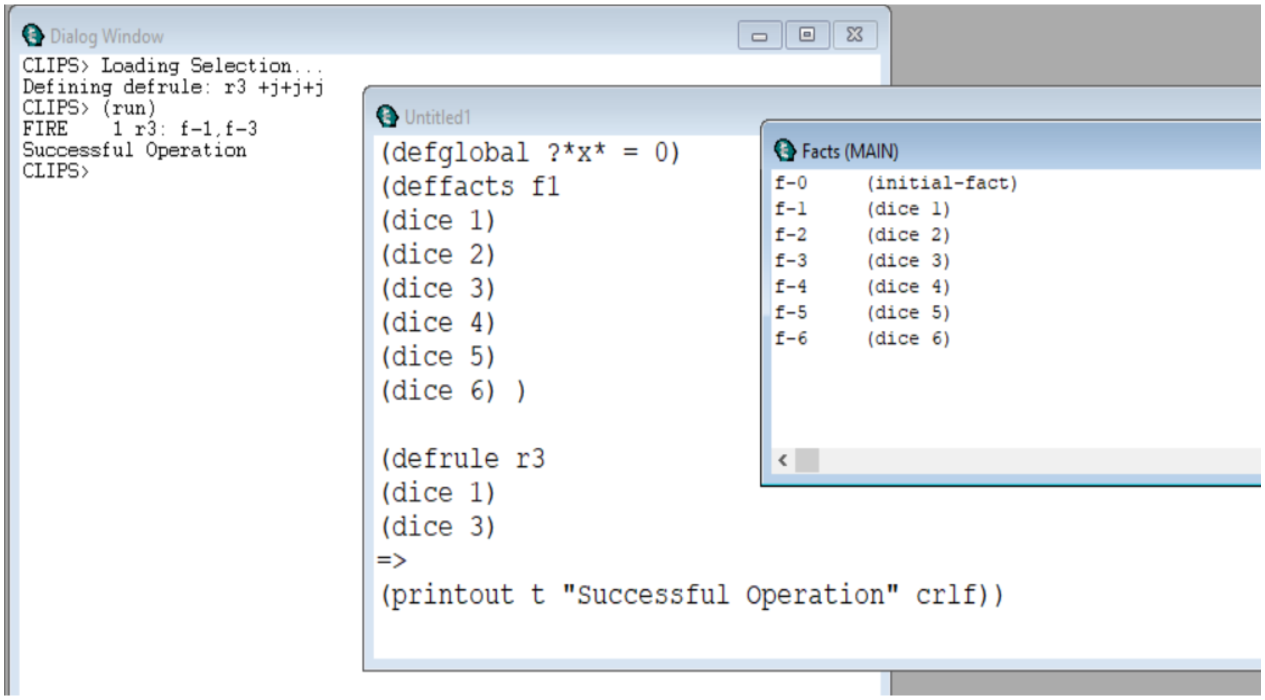
Untitled1

```
(defglobal ?*x* = 0)
(deffacts f1
(dice 1)
(dice 2)
(dice 3)
(dice 4)
(dice 5)
(dice 6) )

(defrule r2
(dice ?)
=>
(printout t OK crlf))
```

Facts (MAIN)

```
f-0     (initial-fact)
f-1     (dice 1)
f-2     (dice 2)
f-3     (dice 3)
f-4     (dice 4)
f-5     (dice 5)
f-6     (dice 6)
```

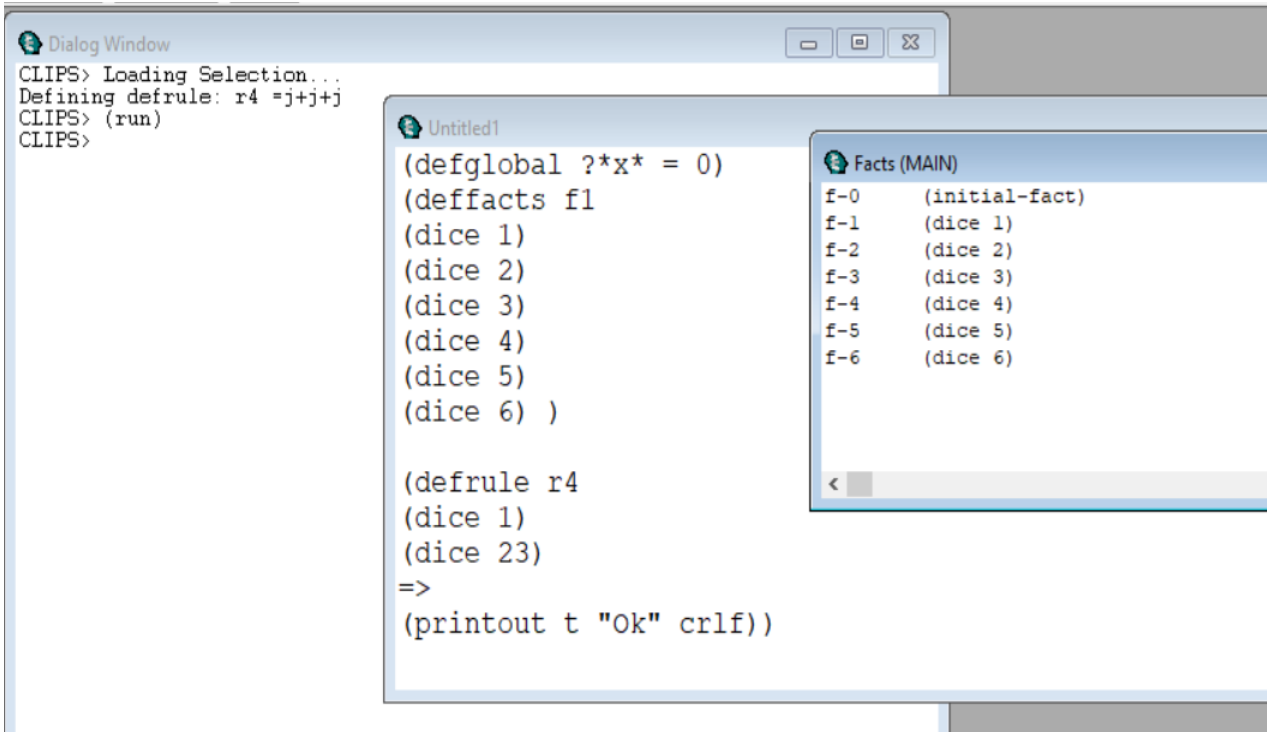
Q3. Write a rule to check if the two facts are stored or not:



```
CLIPS> Loading Selection...
Defining defrule: r3 +j+j+j
CLIPS> (run)
FIRE    1 r3: f-1,f-3
Successful Operation
CLIPS>
```

```
(defglobal ?*x* = 0)
(deffacts f1
(dice 1)
(dice 2)
(dice 3)
(dice 4)
(dice 5)
(dice 6) )

(defrule r3
(dice 1)
(dice 3)
=>
(printout t "Successful Operation" crlf))
```

```
f-0     (initial-fact)
f-1     (dice 1)
f-2     (dice 2)
f-3     (dice 3)
f-4     (dice 4)
f-5     (dice 5)
f-6     (dice 6)
```

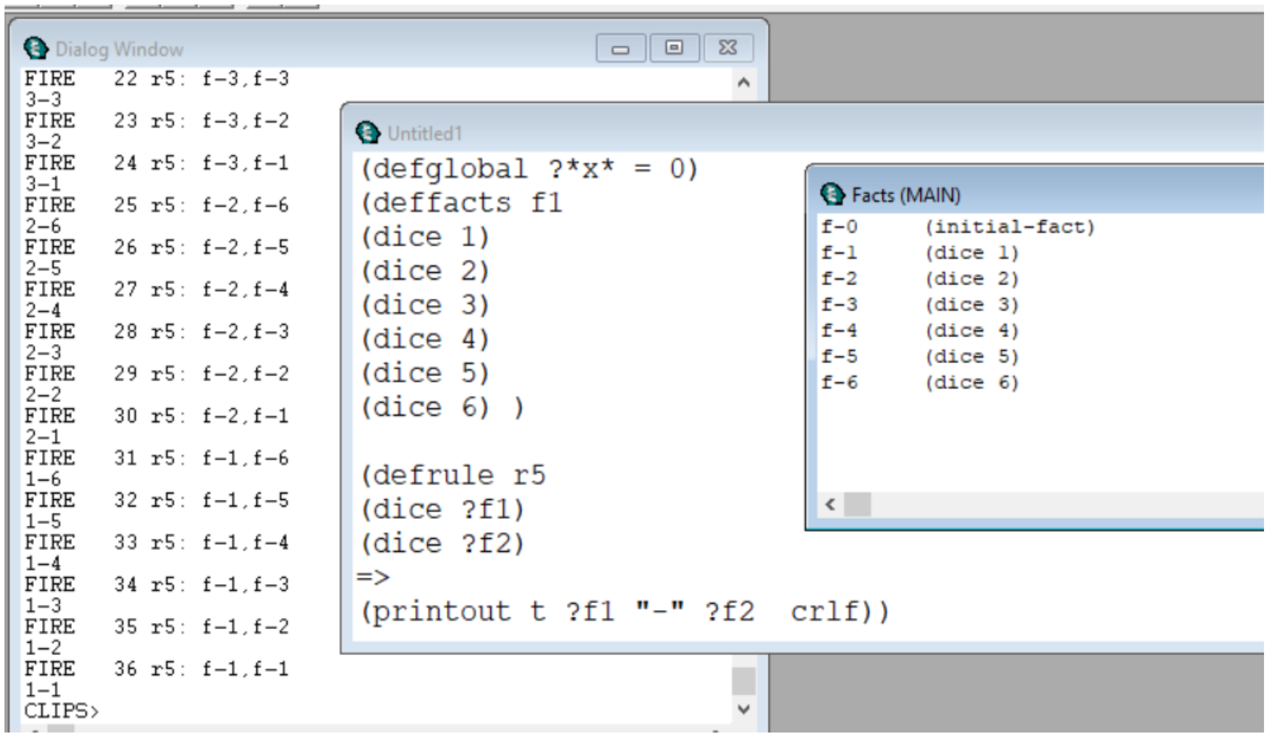


```
CLIPS> Loading Selection...
Defining defrule: r4 =j+j+j
CLIPS> (run)
CLIPS>
```

```
(defglobal ?*x* = 0)
(deffacts f1
(dice 1)
(dice 2)
(dice 3)
(dice 4)
(dice 5)
(dice 6) )

(defrule r4
(dice 1)
(dice 23)
=>
(printout t "Ok" crlf))
```

```
f-0     (initial-fact)
f-1     (dice 1)
f-2     (dice 2)
f-3     (dice 3)
f-4     (dice 4)
f-5     (dice 5)
f-6     (dice 6)
```

Q4. Write a rule to print possible probabilities of the sides of the dice:

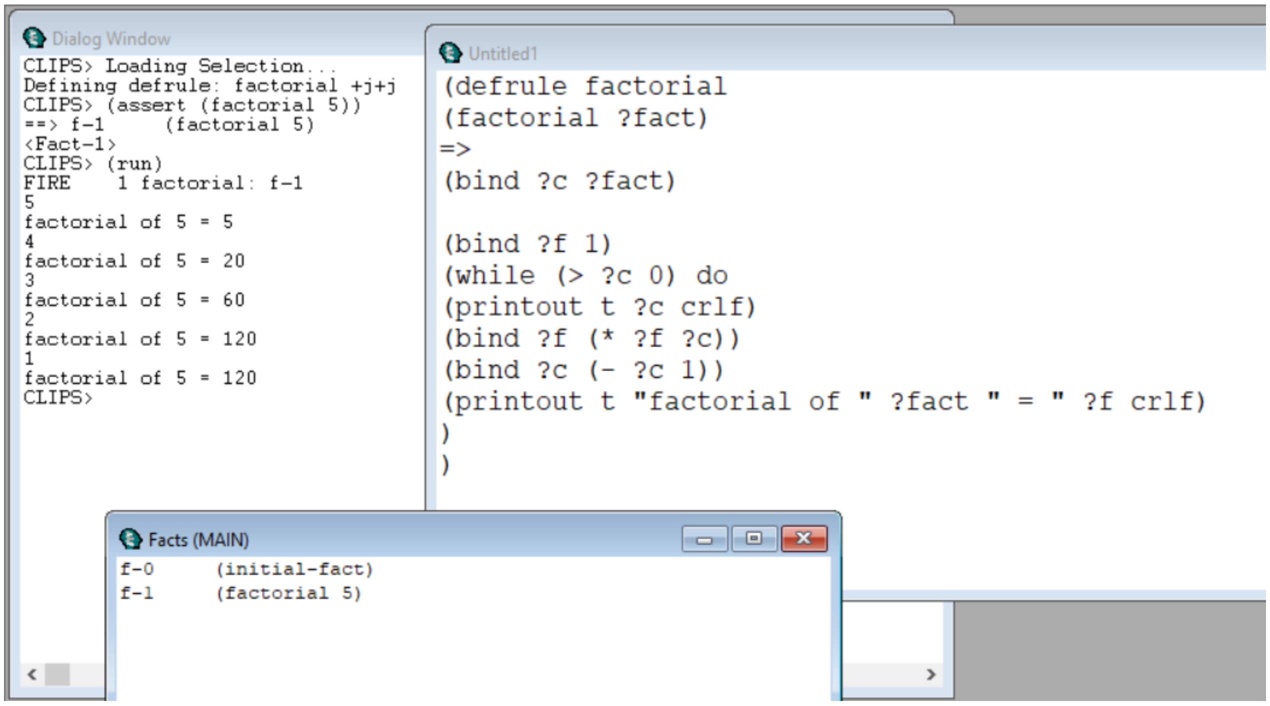


```
FIRE    22 r5: f-3,f-3
3-3
FIRE    23 r5: f-3,f-2
3-2
FIRE    24 r5: f-3,f-1
3-1
FIRE    25 r5: f-2,f-6
2-6
FIRE    26 r5: f-2,f-5
2-5
FIRE    27 r5: f-2,f-4
2-4
FIRE    28 r5: f-2,f-3
2-3
FIRE    29 r5: f-2,f-2
2-2
FIRE    30 r5: f-2,f-1
2-1
FIRE    31 r5: f-1,f-6
1-6
FIRE    32 r5: f-1,f-5
1-5
FIRE    33 r5: f-1,f-4
1-4
FIRE    34 r5: f-1,f-3
1-3
FIRE    35 r5: f-1,f-2
1-2
FIRE    36 r5: f-1,f-1
1-1
CLIPS>
```

```
(defglobal ?*x* = 0)
(deffacts f1
(dice 1)
(dice 2)
(dice 3)
(dice 4)
(dice 5)
(dice 6) )

(defrule r5
(dice ?f1)
(dice ?f2)
=>
(printout t ?f1 "-" ?f2  crlf))
```

```
f-0      (initial-fact)
f-1      (dice 1)
f-2      (dice 2)
f-3      (dice 3)
f-4      (dice 4)
f-5      (dice 5)
f-6      (dice 6)
```

Q5. Write a construction to print the factorial of numbers:

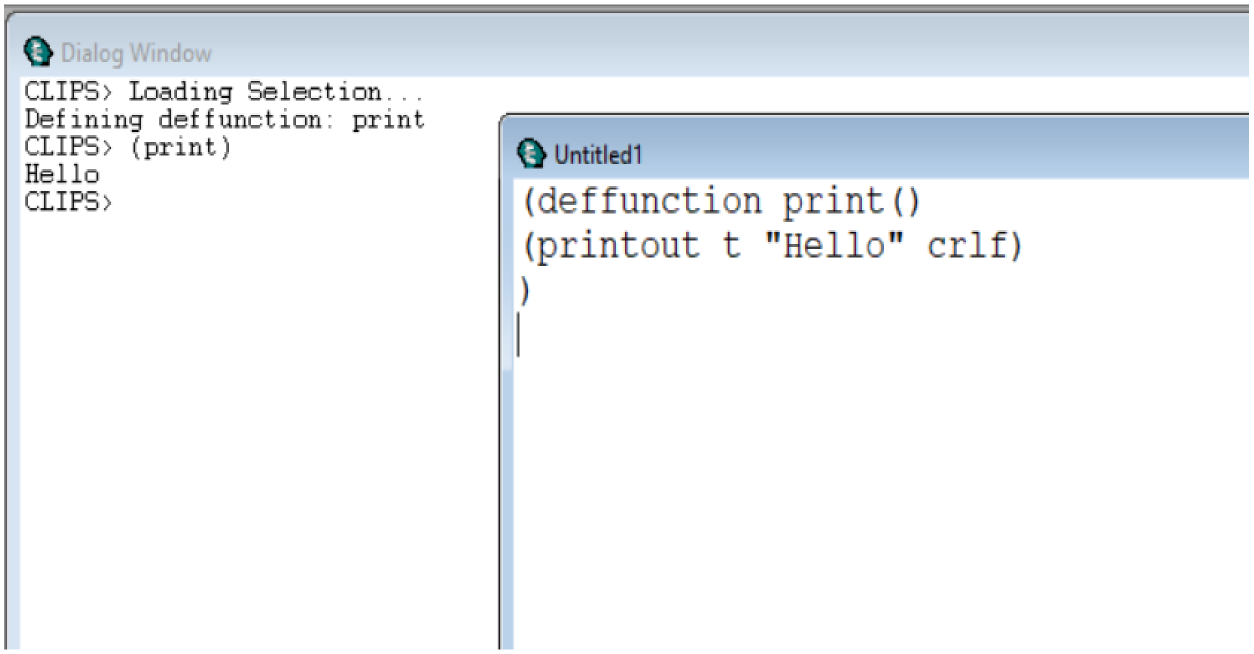


```
CLIPS> Loading Selection...
Defining defrule: factorial +j+j
CLIPS> (assert (factorial 5))
==> f-1     (factorial 5)
<Fact-1>
CLIPS> (run)
FIRE    1 factorial: f-1
5
factorial of 5 = 5
4
factorial of 5 = 20
3
factorial of 5 = 60
2
factorial of 5 = 120
1
factorial of 5 = 120
CLIPS>
```

```
(defrule factorial
(factorial ?fact)
=>
(bind ?c ?fact)

(bind ?f 1)
(while (> ?c 0) do
(printout t ?c crlf)
(bind ?f (* ?f ?c))
(bind ?c (- ?c 1))
(printout t "factorial of " ?fact " = " ?f crlf)
)
)
```

```
f-0      (initial-fact)
f-1      (factorial 5)
```

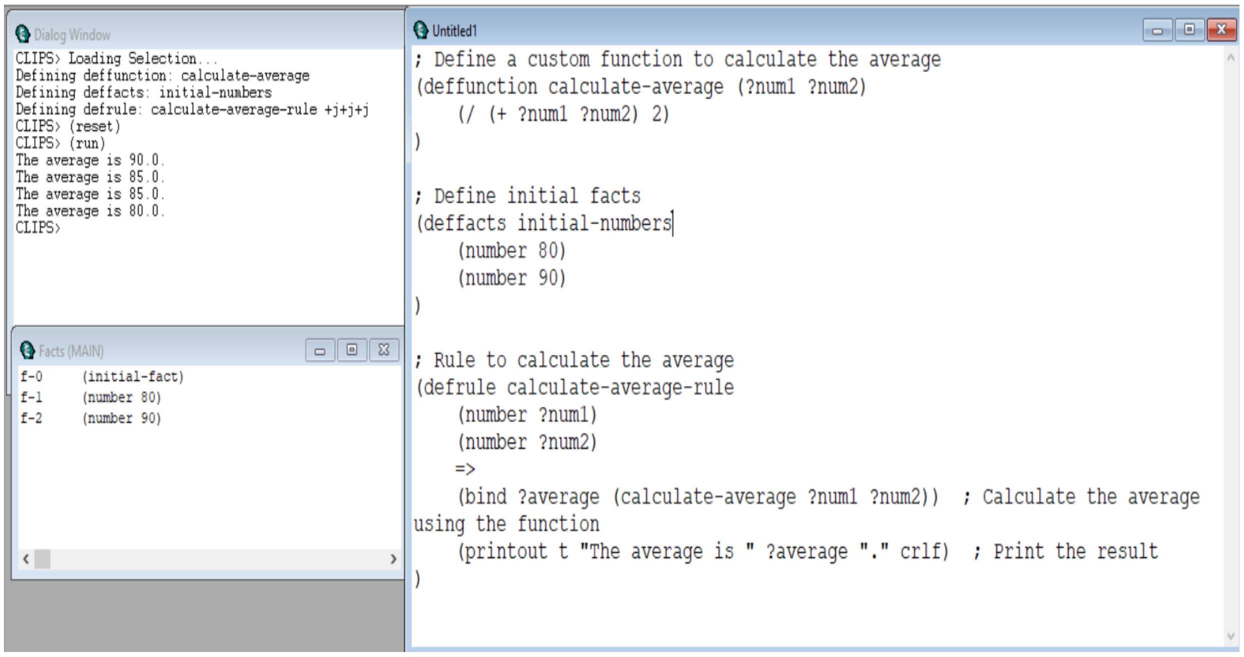
Q5. declaring function in clips:



```
CLIPS> Loading Selection...
Defining deffunction: print
CLIPS> (print)
Hello
CLIPS>
```

```
(deffunction print()
(printout t "Hello" crlf)
)
```

Q6. Define a custom function to calculate the average

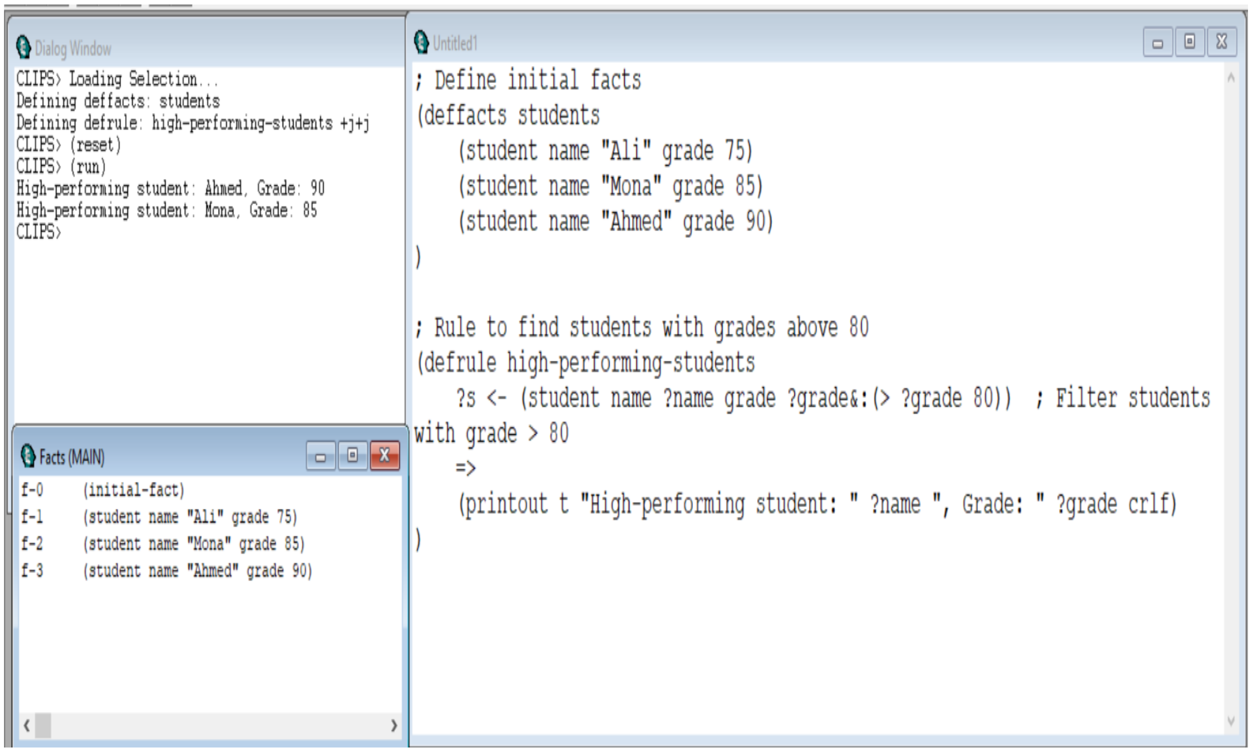


```
CLIPS> Loading Selection...
Defining deffunction: calculate-average
Defining deffacts: initial-numbers
Defining defrule: calculate-average-rule +j+j+j
CLIPS> (reset)
CLIPS> (run)
The average is 90.0.
The average is 85.0.
The average is 85.0.
The average is 80.0.
CLIPS>
```

```
f-0     (initial-fact)
f-1     (number 80)
f-2     (number 90)
```

```
; Define a custom function to calculate the average
(deffunction calculate-average (?num1 ?num2)
    (/ (+ ?num1 ?num2) 2)
)

; Define initial facts
(deffacts initial-numbers
    (number 80)
    (number 90)
)

; Rule to calculate the average
(defrule calculate-average-rule
    (number ?num1)
    (number ?num2)
    =>
    (bind ?average (calculate-average ?num1 ?num2))  ; Calculate the average
using the function
    (printout t "The average is " ?average "." crlf)  ; Print the result
)
```

Q7. Write a fact that includes the student's degree then print the students they got grade > 80:

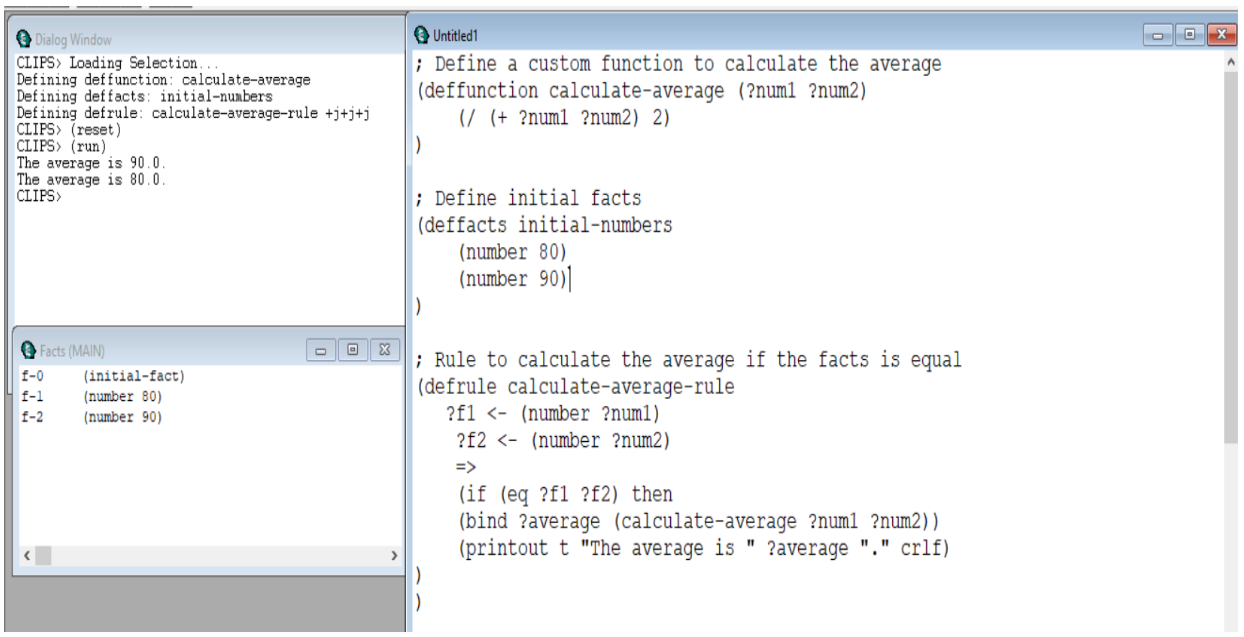


```
CLIPS> Loading Selection...
Defining deffacts: students
Defining defrule: high-performing-students +j+j
CLIPS> (reset)
CLIPS> (run)
High-performing student: Ahmed, Grade: 90
High-performing student: Mona, Grade: 85
CLIPS>
```

```
f-0     (initial-fact)
f-1     (student name "Ali" grade 75)
f-2     (student name "Mona" grade 85)
f-3     (student name "Ahmed" grade 90)
```

```
; Define initial facts
(deffacts students
    (student name "Ali" grade 75)
    (student name "Mona" grade 85)
    (student name "Ahmed" grade 90)
)

; Rule to find students with grades above 80
(defrule high-performing-students
    ?s <- (student name ?name grade ?grade&:(> ?grade 80))  ; Filter students
with grade > 80
    =>
    (printout t "High-performing student: " ?name ", Grade: " ?grade crlf)
)
```

Q8. Write a function to calculate the average between two numbers if the two facts are equal:

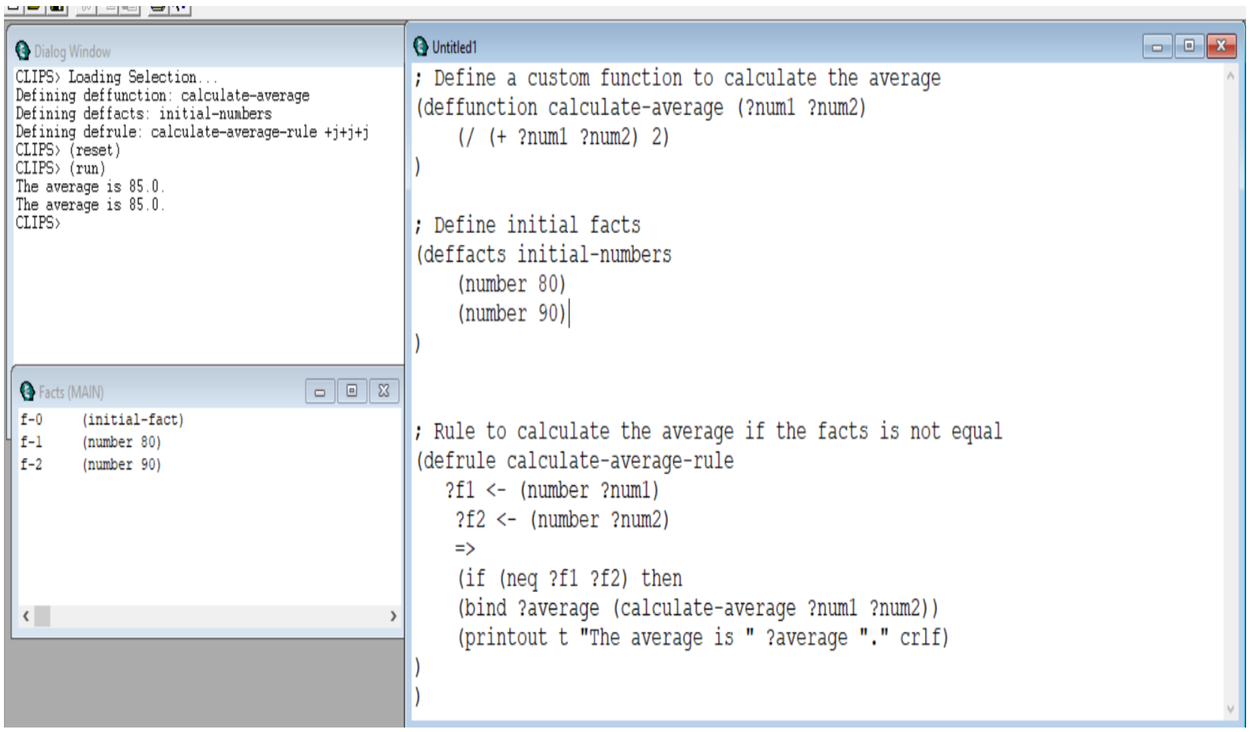


```
CLIPS> Loading Selection...
Defining deffunction: calculate-average
Defining deffacts: initial-numbers
Defining defrule: calculate-average-rule +j+j+j
CLIPS> (reset)
CLIPS> (run)
The average is 90.0.
The average is 80.0.
CLIPS>
```

```
f-0     (initial-fact)
f-1     (number 80)
f-2     (number 90)
```

```
; Define a custom function to calculate the average
(deffunction calculate-average (?num1 ?num2)
    (/ (+ ?num1 ?num2) 2)
)

; Define initial facts
(deffacts initial-numbers
    (number 80)
    (number 90)
)

; Rule to calculate the average if the facts is equal
(defrule calculate-average-rule
   ?f1 <- (number ?num1)
    ?f2 <- (number ?num2)
    =>
    (if (eq ?f1 ?f2) then
    (bind ?average (calculate-average ?num1 ?num2))
    (printout t "The average is " ?average "." crlf)
)
)
```

Q9. Write a function to calculate the average between two numbers if the two facts are not equal:

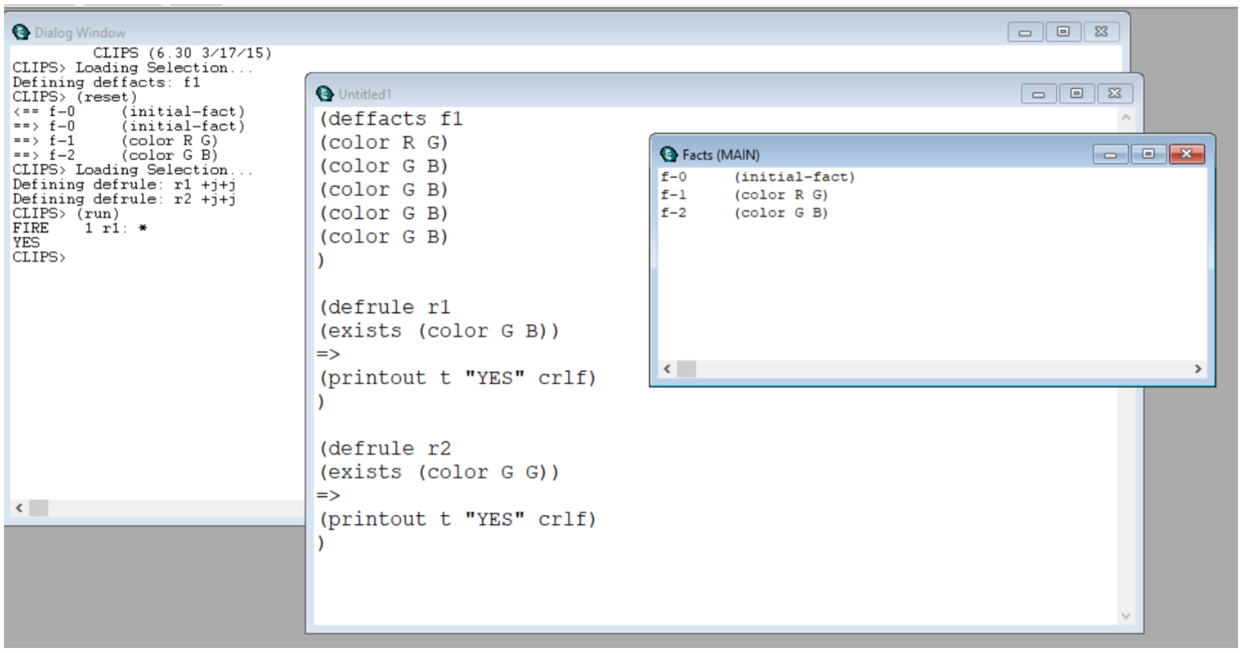


```
CLIPS> Loading Selection...
Defining deffunction: calculate-average
Defining deffacts: initial-numbers
Defining defrule: calculate-average-rule +j+j+j
CLIPS> (reset)
CLIPS> (run)
The average is 85.0.
The average is 85.0.
CLIPS>
```

```
f-0     (initial-fact)
f-1     (number 80)
f-2     (number 90)
```

```
; Define a custom function to calculate the average
(deffunction calculate-average (?num1 ?num2)
    (/ (+ ?num1 ?num2) 2)
)

; Define initial facts
(deffacts initial-numbers
    (number 80)
    (number 90)
)


; Rule to calculate the average if the facts is not equal
(defrule calculate-average-rule
   ?f1 <- (number ?num1)
    ?f2 <- (number ?num2)
    =>
    (if (neq ?f1 ?f2) then
    (bind ?average (calculate-average ?num1 ?num2))
    (printout t "The average is " ?average "." crlf)
)
)
```

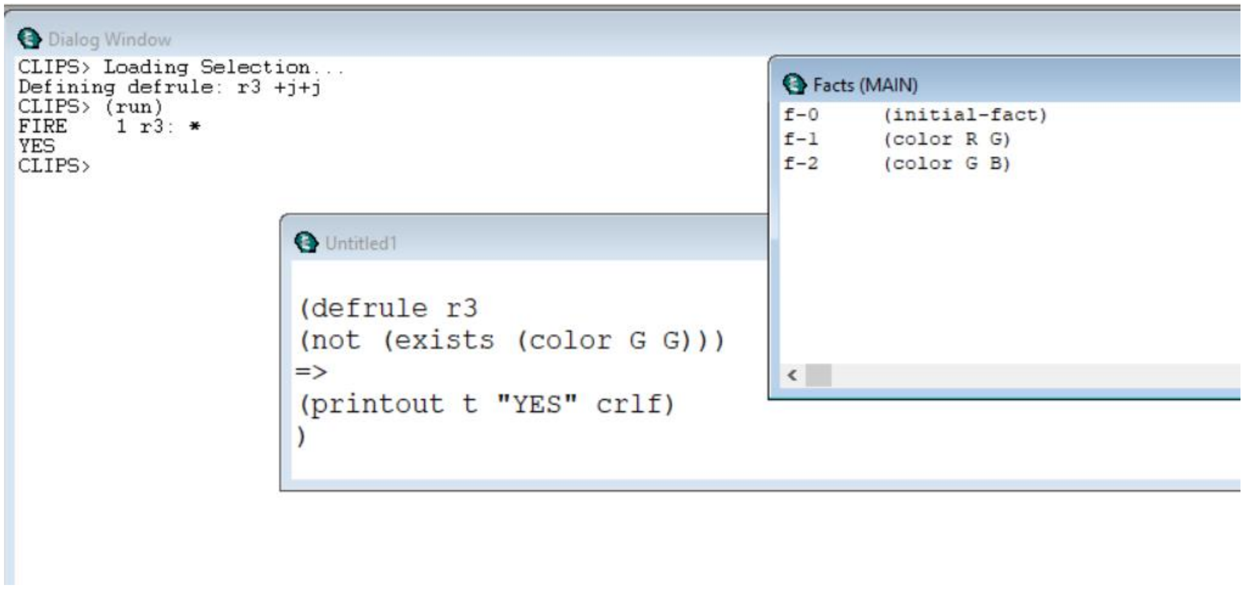
Q10: Write a fact that contains some colors then check if this color exists or not:



```
         CLIPS (6.30 3/17/15)
CLIPS> Loading Selection...
Defining deffacts: f1
CLIPS> (reset)
<== f-0     (initial-fact)
==> f-0     (initial-fact)
==> f-1     (color R G)
==> f-2     (color G B)
CLIPS> Loading Selection...
Defining defrule: r1 +j+j
Defining defrule: r2 +j+j
CLIPS> (run)
FIRE    1 r1: *
YES
CLIPS>
```

```
(deffacts f1
(color R G)
(color G B)
(color G B)
(color G B)
(color G B)
)

(defrule r1
(exists (color G B))
=>
(printout t "YES" crlf)
)

(defrule r2
(exists (color G G))
=>
(printout t "YES" crlf)
)
```

```
f-0     (initial-fact)
f-1     (color R G)
f-2     (color G B)
```



```
CLIPS> Loading Selection...
Defining defrule: r3 +j+j
CLIPS> (run)
FIRE    1 r3: *
YES
CLIPS>
```

```
f-0     (initial-fact)
f-1     (color R G)
f-2     (color G B)
```

```
(defrule r3
(not (exists (color G G)))
=>
(printout t "YES" crlf)
)
```

General definition of class:

```
(defclass className (is-a classType)

(variableType   variableName)

)
```

Class Type:

1) Super -> in this case we write (is-a USER)
2) Inherit -> in this case we write a name of the class we inherit from it (is-a A), which A is a super class

Variable Type:

(slot b) -> this variable has a single value

(multislot b) -> this variable has a multi value

Ex1: Make a Super class:

```
(defclass student (is-a USER)

(slot n)

(slot a)

)


(defclass doctor (is-a USER)

(slot n)

(slot s)

)
```

Make instance from classes:

We use (make-instance [instance Name] of className (VariableName VariableValue))

We use rule to make instance.

Ex2:

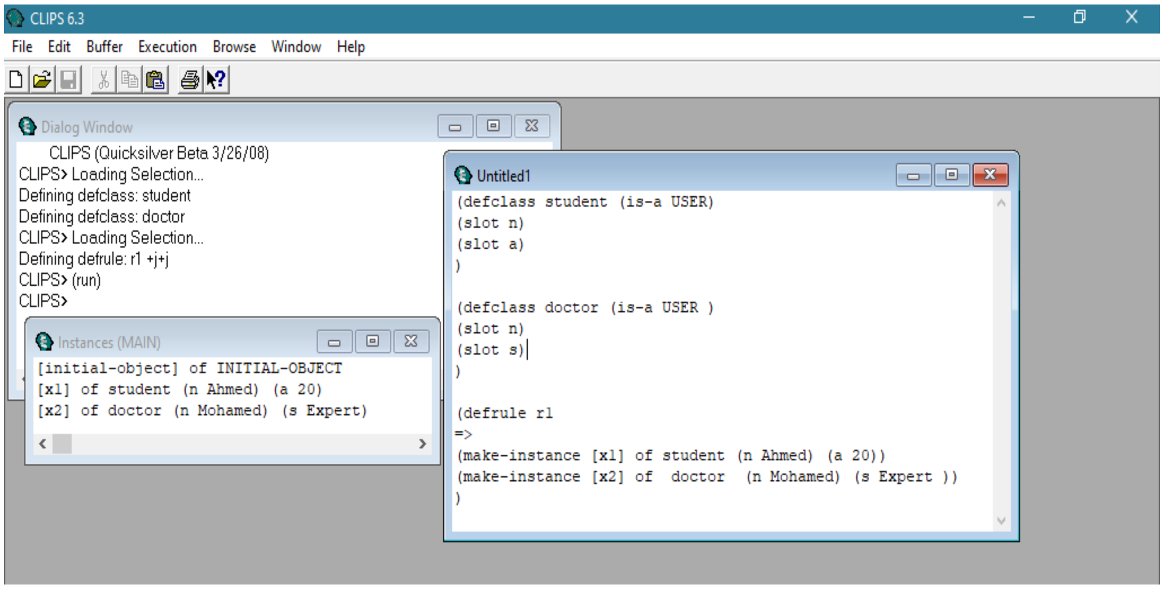
(defrule r1

=>

(make-instance [x1] of student (n Ahmed) (a 20))

(make-instance [x2] of doctor (n Mohamed) (s Expert))

)



```
CLIPS (Quicksilver Beta 3/26/08)
CLIPS> Loading Selection...
Defining defclass: student
Defining defclass: doctor
CLIPS> Loading Selection...
Defining defrule: r1 +j+j
CLIPS> (run)
CLIPS>
```

```
[initial-object] of INITIAL-OBJECT
[x1] of student (n Ahmed) (a 20)
[x2] of doctor (n Mohamed) (s Expert)
```

```
(defclass student (is-a USER)
(slot n)
(slot a)
)

(defclass doctor (is-a USER )
(slot n)
(slot s)
)

(defrule r1
=>
(make-instance [x1] of student (n Ahmed) (a 20))
(make-instance [x2] of  doctor  (n Mohamed) (s Expert ))
)
```

Git values of instant and print it:

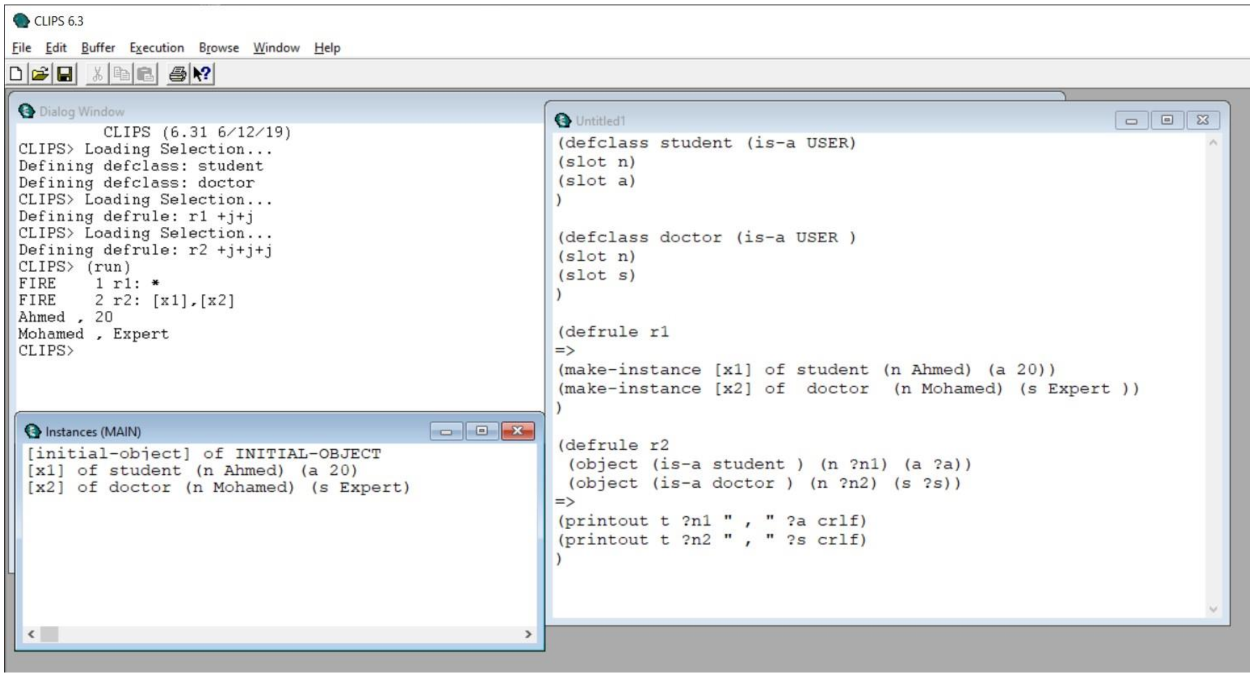
we use (object (is-a ClassName) (VariableName ?n)) in the rule

Ex3:

(defrule r2

(object (is-a student) (n ?n1) (a ?a))

```
 (object (is-a doctor ) (n ?n2) (s ?s))

=>

(printout t ?n1 " , " ?a crlf)

(printout t ?n2 " , " ?s crlf)

)
```



```
         CLIPS (6.31 6/12/19)
CLIPS> Loading Selection...
Defining defclass: student
Defining defclass: doctor
CLIPS> Loading Selection...
Defining defrule: r1 +j+j
CLIPS> Loading Selection...
Defining defrule: r2 +j+j+j
CLIPS> (run)
FIRE    1 r1: *
FIRE    2 r2: [x1],[x2]
Ahmed , 20
Mohamed , Expert
CLIPS>
```

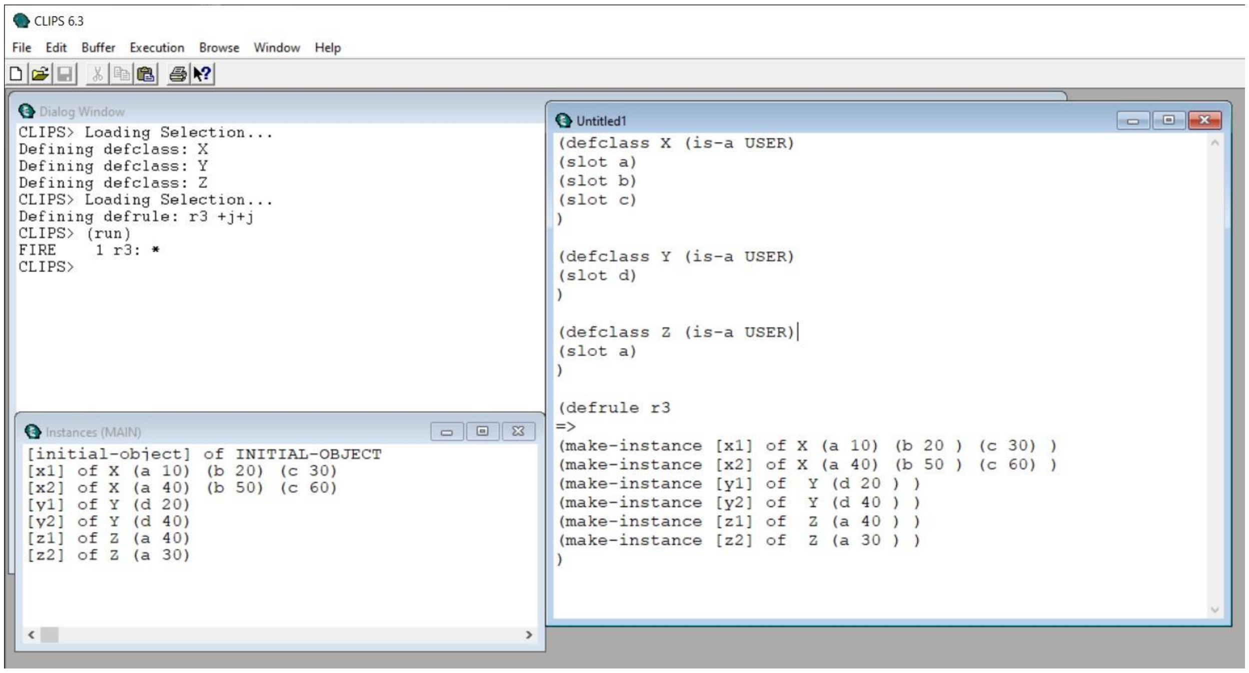
```
[initial-object] of INITIAL-OBJECT
[x1] of student (n Ahmed) (a 20)
[x2] of doctor (n Mohamed) (s Expert)
```

```
(defclass student (is-a USER)
(slot n)
(slot a)
)

(defclass doctor (is-a USER )
(slot n)
(slot s)
)

(defrule r1
=>
(make-instance [x1] of student (n Ahmed) (a 20))
(make-instance [x2] of  doctor  (n Mohamed) (s Expert ))
)

(defrule r2
 (object (is-a student ) (n ?n1) (a ?a))
 (object (is-a doctor ) (n ?n2) (s ?s))
=>
(printout t ?n1 " , " ?a crlf)
(printout t ?n2 " , " ?s crlf)
)
```

<u>Ex4: Make Three super classes X ,Y,Z and make 2 instance of all class:</u>

```
(defclass X (is-a USER)

(slot a)

(slot b)

(slot c)

)
```

```
(defclass Y (is-a USER)
(slot d)
)
(defclass Z (is-a USER)
(slot a)
)
(defrule r3
=>
(make-instance [x1] of X (a 10) (b 20 ) (c 30) )
(make-instance [x2] of X (a 40) (b 50 ) (c 60) )
(make-instance [y1] of  Y (d 20 ) )
(make-instance  [y2]  of  Y (d 40 ) )
(make-instance  [z1]  of  Z (a 40  ) )
(make-instance [z2] of  Z (a 30 ) ))
```



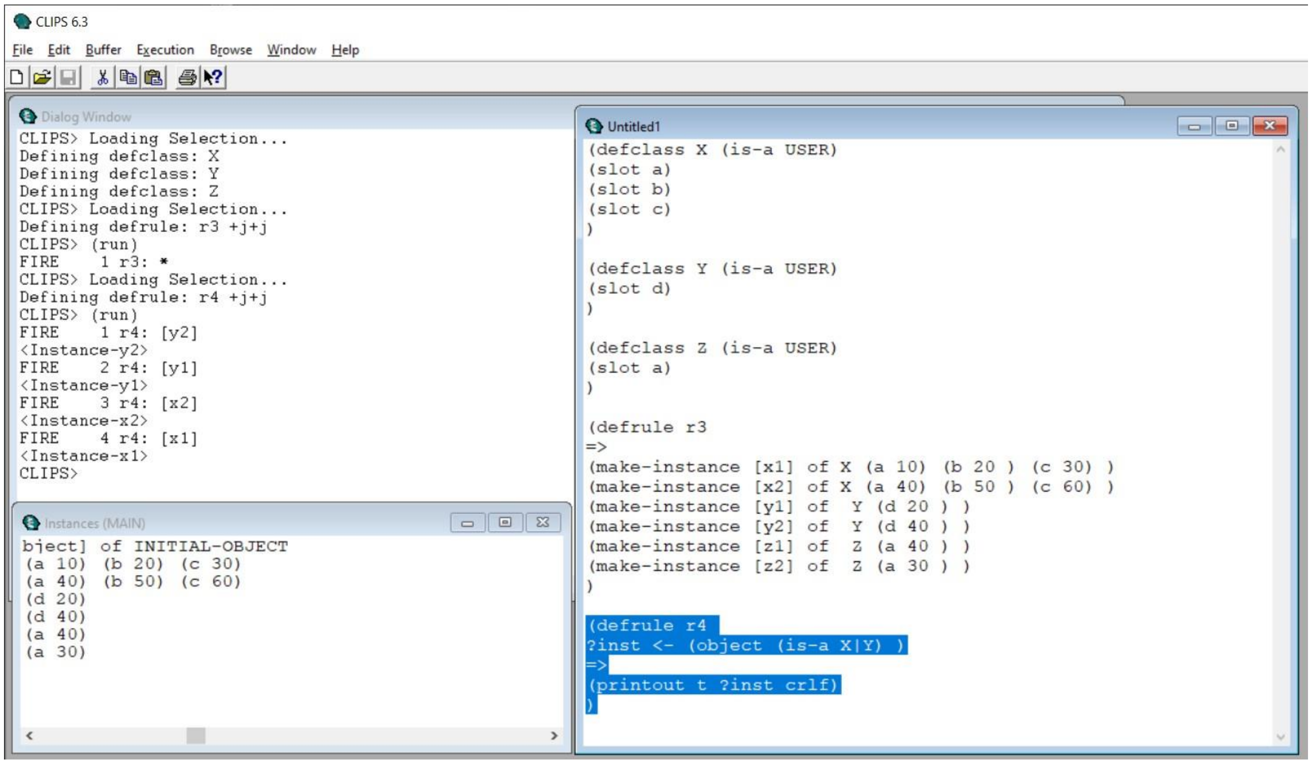
```
CLIPS> Loading Selection...
Defining defclass: X
Defining defclass: Y
Defining defclass: Z
CLIPS> Loading Selection...
Defining defrule: r3 +j+j
CLIPS> (run)
FIRE    1 r3: *
CLIPS>
```

```
[initial-object] of INITIAL-OBJECT
[x1] of X (a 10) (b 20) (c 30)
[x2] of X (a 40) (b 50) (c 60)
[y1] of Y (d 20)
[y2] of Y (d 40)
[z1] of Z (a 40)
[z2] of Z (a 30)
```

```
(defclass X (is-a USER)
(slot a)
(slot b)
(slot c)
)

(defclass Y (is-a USER)
(slot d)
)

(defclass Z (is-a USER)
(slot a)
)

(defrule r3
=>
(make-instance [x1] of X (a 10) (b 20 ) (c 30) )
(make-instance [x2] of X (a 40) (b 50 ) (c 60) )
(make-instance [y1] of  Y (d 20 ) )
(make-instance [y2] of  Y (d 40 ) )
(make-instance [z1] of  Z (a 40 ) )
(make-instance [z2] of  Z (a 30 ) )
)
```

Ex5: Print values of instance of class X or Y:

```
(defrule r4
?inst <- (object (is-a X|Y) )
=>
(printout t ?inst crlf)
)
```



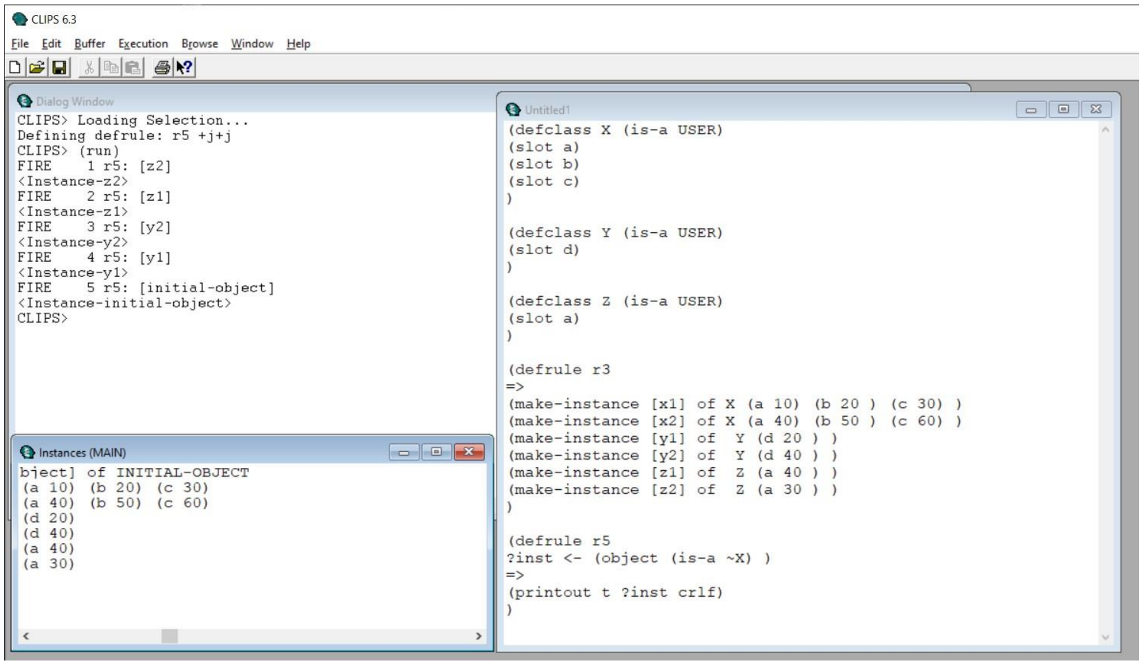
```
CLIPS> Loading Selection...
Defining defclass: X
Defining defclass: Y
Defining defclass: Z
CLIPS> Loading Selection...
Defining defrule: r3 +j+j
CLIPS> (run)
FIRE    1 r3: *
CLIPS> Loading Selection...
Defining defrule: r4 +j+j
CLIPS> (run)
FIRE    1 r4: [y2]
<Instance-y2>
FIRE    2 r4: [y1]
<Instance-y1>
FIRE    3 r4: [x2]
<Instance-x2>
FIRE    4 r4: [x1]
<Instance-x1>
CLIPS>
```

```
bject] of INITIAL-OBJECT
 (a 10) (b 20) (c 30)
 (a 40) (b 50) (c 60)
 (d 20)
 (d 40)
 (a 40)
 (a 30)
```

```
(defclass X (is-a USER)
(slot a)
(slot b)
(slot c)
)

(defclass Y (is-a USER)
(slot d)
)

(defclass Z (is-a USER)
(slot a)
)

(defrule r3
=>
(make-instance [x1] of X (a 10) (b 20 ) (c 30) )
(make-instance [x2] of X (a 40) (b 50 ) (c 60) )
(make-instance [y1] of  Y (d 20 ) )
(make-instance [y2] of  Y (d 40 ) )
(make-instance [z1] of  Z (a 40 ) )
(make-instance [z2] of  Z (a 30 ) )
)

(defrule r4
?inst <- (object (is-a X|Y) )
=>
(printout t ?inst crlf)
)
```

Ex6: Print all values of instance which not X:

```
(defrule r5
?inst <- (object (is-a ~X) )
=>
(printout t ?inst crlf)
)
```



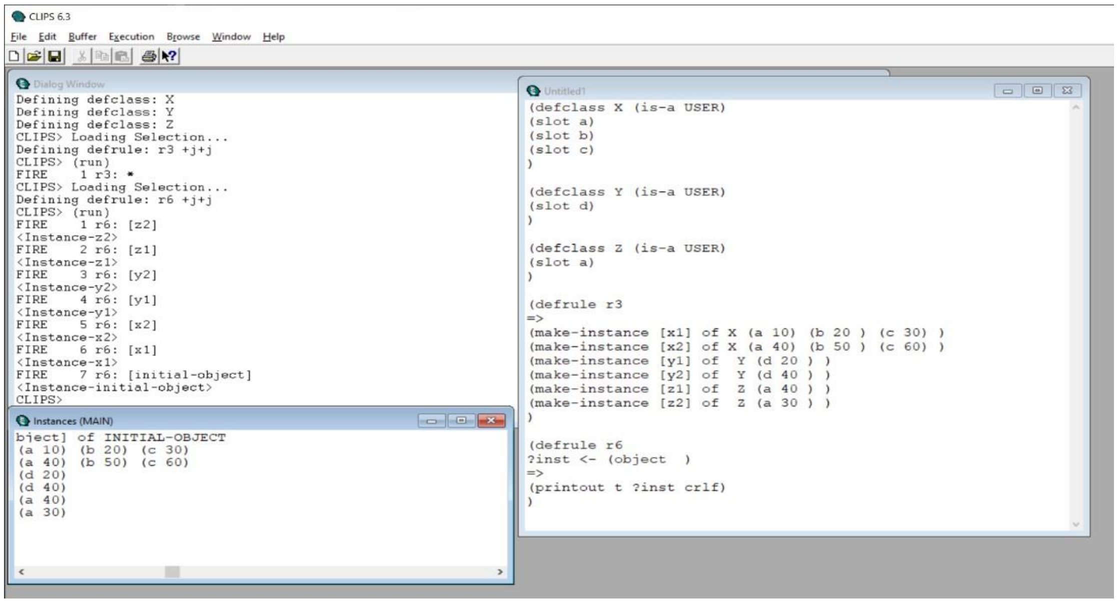
```
CLIPS> Loading Selection...
Defining defrule: r5 +j+j
CLIPS> (run)
FIRE    1 r5: [z2]
<Instance-z2>
FIRE    2 r5: [z1]
<Instance-z1>
FIRE    3 r5: [y2]
<Instance-y2>
FIRE    4 r5: [y1]
<Instance-y1>
FIRE    5 r5: [initial-object]
<Instance-initial-object>
CLIPS>
```

```
bject] of INITIAL-OBJECT
 (a 10) (b 20) (c 30)
 (a 40) (b 50) (c 60)
 (d 20)
 (d 40)
 (a 40)
 (a 30)
```

```
(defclass X (is-a USER)
(slot a)
(slot b)
(slot c)
)

(defclass Y (is-a USER)
(slot d)
)

(defclass Z (is-a USER)
(slot a)
)

(defrule r3
=>
(make-instance [x1] of X (a 10) (b 20 ) (c 30) )
(make-instance [x2] of X (a 40) (b 50 ) (c 60) )
(make-instance [y1] of  Y (d 20 ) )
(make-instance [y2] of  Y (d 40 ) )
(make-instance [z1] of  Z (a 40 ) )
(make-instance [z2] of  Z (a 30 ) )
)

(defrule r5
?inst <- (object (is-a ~X) )
=>
(printout t ?inst crlf)
)
```

Ex7: print values of all instance:

```
(defrule r6
?inst <- (object  )
=>
(printout t ?inst crlf) )
```



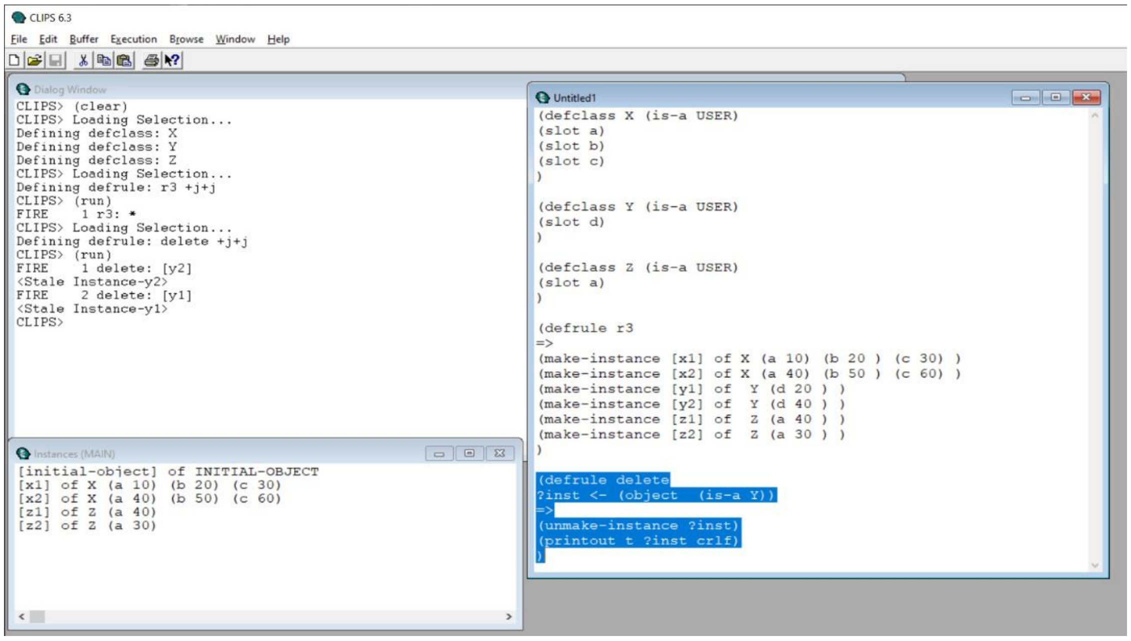
```
Defining defclass: X
Defining defclass: Y
Defining defclass: Z
CLIPS> Loading Selection...
Defining defrule: r3 +j+j
CLIPS> (run)
FIRE    1 r3: *
CLIPS> Loading Selection...
Defining defrule: r6 +j+j
CLIPS> (run)
FIRE    1 r6: [z2]
<Instance-z2>
FIRE    2 r6: [z1]
<Instance-z1>
FIRE    3 r6: [y2]
<Instance-y2>
FIRE    4 r6: [y1]
<Instance-y1>
FIRE    5 r6: [x2]
<Instance-x2>
FIRE    6 r6: [x1]
<Instance-x1>
FIRE    7 r6: [initial-object]
<Instance-initial-object>
CLIPS>
```

```
bject] of INITIAL-OBJECT
 (a 10) (b 20) (c 30)
 (a 40) (b 50) (c 60)
 (d 20)
 (d 40)
 (a 40)
 (a 30)
```

```
(defclass X (is-a USER)
(slot a)
(slot b)
(slot c)
)

(defclass Y (is-a USER)
(slot d)
)

(defclass Z (is-a USER)
(slot a)
)

(defrule r3
=>
(make-instance [x1] of X (a 10) (b 20 ) (c 30) )
(make-instance [x2] of X (a 40) (b 50 ) (c 60) )
(make-instance [y1] of  Y (d 20 ) )
(make-instance [y2] of  Y (d 40 ) )
(make-instance [z1] of  Z (a 40 ) )
(make-instance [z2] of  Z (a 30 ) )
)

(defrule r6
?inst <- (object  )
=>
(printout t ?inst crlf)
)
```

Ex8: To delete instance we use (unmake-instance):

```
(defrule delete
?inst <- (object  (is-a Y)
=>
(unmake-instance ?inst)
(printout t ?inst crlf)
)
```



```
CLIPS> (clear)
CLIPS> Loading Selection...
Defining defclass: X
Defining defclass: Y
Defining defclass: Z
CLIPS> Loading Selection...
Defining defrule: r3 +j+j
CLIPS> (run)
FIRE    1 r3: *
CLIPS> Loading Selection...
Defining defrule: delete +j+j
CLIPS> (run)
FIRE    1 delete: [y2]
<Stale Instance-y2>
FIRE    2 delete: [y1]
<Stale Instance-y1>
CLIPS>
```

```
[initial-object] of INITIAL-OBJECT
[x1] of X (a 10) (b 20) (c 30)
[x2] of X (a 40) (b 50) (c 60)
[z1] of Z (a 40)
[z2] of Z (a 30)
```

```
(defclass X (is-a USER)
(slot a)
(slot b)
(slot c)
)

(defclass Y (is-a USER)
(slot d)
)

(defclass Z (is-a USER)
(slot a)
)

(defrule r3
=>
(make-instance [x1] of X (a 10) (b 20 ) (c 30) )
(make-instance [x2] of X (a 40) (b 50 ) (c 60) )
(make-instance [y1] of  Y (d 20 ) )
(make-instance [y2] of  Y (d 40 ) )
(make-instance [z1] of  Z (a 40 ) )
(make-instance [z2] of  Z (a 30 ) )
)

(defrule delete
?inst <- (object  (is-a Y))
=>
(unmake-instance ?inst)
(printout t ?inst crlf)
)
```