CLIPS

● Breve historia y aplicaciones del lenguaje asignado

CLIPS: es una herramienta de desarrollo para la producción y ejecución de sistemas expertos. Fue creado en el año 1984 en el space center de la NASA Lyndon B. Johnson. Su nombre es un acrónimo de C **L**anguage **I**ntegrated **P**roduction **S**ystem (Sistema de Producción Integrado en Lenguaje C).

Durante el año 1984, en el Lyndon B. Johnson Space Center, la sección de inteligencia artificial había desarrollado alrededor de una docena de prototipos de sistemas expertos usando hardware y software de aquella época. A pesar del demostrado potencial de los sistemas expertos, la mayoría de aquellos prototipos no estaban siendo usados regularmente, según la NASA esto se debió a que el lenguaje de programación usado para el desarrollo de estas aplicaciones era LISP. Se encontraron varias debilidades de

LISP, de las cuales se destacan tres:
- No estaba disponible para una amplia variedad de equipos de cómputo.
- No era fácilmente integrable con otras aplicaciones.
- Su costo era muy elevado.

Sus aplicaciones pueden comprender diferentes tipos de algoritmos:
- Búsqueda (ciega, con heurística, con coste y heurística)
- Sistemas de control
- Árboles de decisión

● Palabras reservadas seleccionadas

Palabras reservadas seleccionadas de CLIPS			
defmodule	assert	slot	retract
deftemplate	import	deffacts	modify
defmethod	export	defrule	bind
deffunction	type	declare	printout
multislot	range	test	read

● Expresión regular para los identificadores

Una expresión regular denota un conjunto de secuencias de símbolos válidos que se construyen en base al alfabeto existente de un lenguaje. Las expresiones regulares para los identificadores son **[a-zA-Z][a-zA-Z0-9_]{1,}.**

Para el caso de este lenguaje tendríamos:

(letra(letra dígito)) donde letra = A+...+Z+...+a+...z y dígito = A+...+Z+...+a+...+z+...+0+1+2+...+9

● Expresión regular para números enteros y reales

Las expresiones regulares para los números enteros están denotadas por **[0-9]{1,}** mientras que para números reales están denotadas por **[0-9]{1,}\.[0-9]{1,}**

● Lista de operadores y caracteres especiales seleccionados

Operadores y caracteres especiales de CLIPS			
!=	diferente	&	restricción
*	multiplicación	=	igual
=>	entonces	>	mayor que
+	suma	>=	mayor igual
-	resta	?	comodines o *wildcards*.
/	división	(	delimitador de inicio
<	menor que	)	delimitador de cierre
<=	menor igual	~	restricción

● Consideraciones especiales

- Restricción tilde: La restricción conectiva tilde actúa negando el valor sobre el que actúa. Para aclarar este concepto, supongamos que queremos escribir una regla que niegue el paso a toda persona de la base de conocimiento que no tenga de nombre Juan. La regla sería algo parecido a:

```
(defrule prohibirpaso
    (persona (nombre ~"Juan"))
    =>
    (printout t "prohibdo el paso" crlf)
    (assert (pasar no))
)
```

Como podemos comprobar en el ejemplo, la regla se activará para todos los hechos del tipo persona cuyo nombre no sea Juan, y actuará dando un mensaje prohibiendo el paso y produciendo el hecho (pasar no), que se podría utilizar en una regla que sirviera, por ejemplo, para controlar la apertura de la puerta.

- Restricción barra: La restricción conectiva barra se utiliza para combinar varios hechos. Supongamos que queremos modificar la regla anterior para que se permita el paso a Juan y Pedro. La regla que realizaría esta acción sería:

```
(defrule prohibirpaso
    (persona (nombre ~"Juan"))
    =>
    (printout t "prohibdo el paso" crlf)
    (assert (pasar no))
)
```

- Restricción &: Esta última restricción se utiliza para conectar varias restricciones en unión.

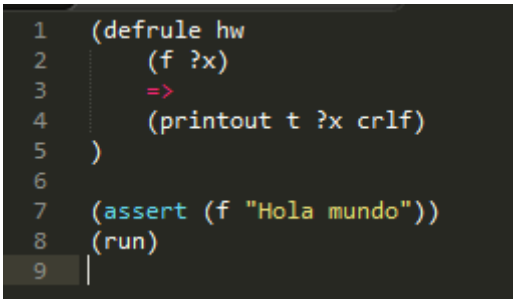
```
(defrule permitirpaso2
     (persona (nombre ?name&"Juan"|"Pedro"))
=>
    (printout t "Puedes pasar" ?name crlf)
    (assert (pasar si))
)
```

- Forma de construcción de comentarios en el lenguaje

La construcción de comentarios en CLIPS comienza con un punto y coma. Todo lo que le sigue al punto y coma será ignorado por el compilador.

```
(defrule duck                          ; Rule header
    "Here comes the quack"             ; Comment
    (animal-is duck)                   ; Pattern
    =>                                 ; THEN arrow
    (assert (sound-is quack))          ; Action
)
```

- Un ejemplo de programa para escribir “Hola mundo” en el lenguaje



```
(defrule hw
    (f ?x)
    =>
    (printout t ?x crlf)
)

(assert (f "Hola mundo"))
(run)
```

Tipos de datos

1. Object
2. String
3. Integer
4. Float
5. Multifield (equivalente a las listas/arreglos)

Tipos de bucles iterativos:

while

(while <expression> [do] <action>*)

Ejemplo:

```
(defrule open-valves
        (valves-open-through ?v)
        =>
        (while (> ?v 0)
          (printout t "Valve " ?v " is open" crlf)
          (bind ?v (- ?v 1))))
```

loop-for-count

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

<range-spec> ::= <end-index> | (<loop-variable> [<start-index> <end-index>])

<start-index> ::= <integer-expression>

<end-index>   ::= <integer-expression>

Ejemplo:

CLIPS> (loop-for-count 2 (printout t "Hello world" crlf))

Hello world

Hello world

FALSE

Funciones de entrada y salida de datos

Open

(open <file-name> <logical-name> [<mode>]) si no se pone modo, asume solo lectura

Ejemplo:

CLIPS> (open "myfile.clp" writeFile "w")

CLIPS> (open "MS-DOS\\directory\\file.clp" readFile)

Close

(close [<logical-name>]) si no se pone el nombre, cierra todos los flujos abiertos

Ejemplo

(open "myfile.clp" writeFile "w")

(close writeFile)

(close)

Printout

(printout <logical-name> <expression>*)

Imprime tanto en consola como en un archivo de texto, dependiendo si el nombre logico hace referencia a un archivo

(printout t "Hello there!" crlf) **crlf** fuerza a que se haga el salto de línea y retorno de carro

Ejemplo

(open "data.txt" mydata "w")

(printout mydata "red green") en este caso imprime en el archivo

(close)

Read

(read [<logical-name>])

lee todo el flujo hasta encontrar un delimitador, si no hay nombre lógico lee desde consola

Ejemplo

CLIPS> (open "data.txt" mydata "w")
TRUE
CLIPS> (printout mydata "red green")
CLIPS> (close)
TRUE
CLIPS> (open "data.txt" mydata)
TRUE
CLIPS> (read mydata)
red
CLIPS> (read mydata)
green
CLIPS> (read mydata)
EOF
CLIPS> (close)
TRUE

Funciones predefinidas

1. (+ <numeric-expression> <numeric-expression>+)
2. (- <numeric-expression> <numeric-expression>+)
3. (* <numeric-expression> <numeric-expression>+)
4. (/ <numeric-expression> <numeric-expression>+)
5. (div <numeric-expression> <numeric-expression>+) CLIPS> (div 5 2) -> 2

6. (max <numeric-expression>+)
7. (min <numeric-expression>+)
8. (abs <numeric-expression>)
9. (float <numeric-expression>)
10. (integer <numeric-expression>)
11. (break)
12. (create$ <expression>*)
13. (nth$ <integer-expression> <multifield-expression>)
14. (member$ <single-field-expression> <multifield-expression>)
15. (delete$ <multifield-expression> <begin-integer-expression> <end-integer-expression>)
16. (explode$ <string-expression>)
17. (implode$ <multifield-expression>)
18. (replace$ <multifield-expression> <begin-integer-expression> <end-integer-expression> <single-or-multi-field-expression>+)
19. (insert$ <multifield-expression> <integer-expression> <single-or-multi-field-expression>+)
20. (first$ <multifield-expression>)
21. (rest$ <multifield-expression>)
22. (floatp <expression>)
23. (integerp <expression>)
24. (stringp <expression>)
25. (evenp <expression>)
26. (oddp <expression>)
27. (multifieldp <expression>)
28. (eq <expression> <expression>+);
29. (neq <expression> <expression>+) not equal
30. (= <numeric-expression> <numeric-expression>+)
31. (<> <numeric-expression> <numeric-expression>+)
32. (> <numeric-expression> <numeric-expression>+)
33. (>= <numeric-expression> <numeric-expression>+)
34. (< <numeric-expression> <numeric-expression>+)
35. (<= <numeric-expression> <numeric-expression>+)
36. (and <expression>+)
37. (or <expression>+)
38. (not <expression>)

Estructuras condicionales

If

(if <expression> then   <action>* [else <action>*])

Ejemplo:

```
(defrule closed-valves
        (temp high)
        (valve ?v closed)
        =>
        (if (= ?v 6)
          then
          (printout t "The special valve " ?v " is closed!" crlf)
          (assert (perform special operation))
          else
          (printout t "Valve " ?v " is normally closed" crlf)))
```

Switch

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

<case-statement> ::= (case <comparison-expression> then <action>*)

<default-statement> ::= (default <action>*)

Ejemplo:

```
CLIPS> (defglobal ?*x* = 0)
CLIPS> (defglobal ?*y* = 1)
CLIPS>
(deffunction foo (?val)
        (switch ?val
        (case ?*x* then *x*)
```

```
        (case ?*y* then *y*)
        (default none)))
CLIPS> (foo 0)
*x*
CLIPS> (foo 1)
*y*
CLIPS> (foo 2)
none
```

Subalgoritmos

Deffunction

```
(deffunction <name> [<comment>]
        (<regular-parameter>* [<wildcard-parameter>])
          <action>*)
<regular-parameter>   ::= <single-field-variable>
<wildcard-parameter>  ::= <multifield-variable>
```

Ejemplo

```
CLIPS>
(deffunction print-args (?a ?b $?c)
   (printout t ?a " " ?b " and " (length ?c) " extras: " ?c crlf))

CLIPS> (print-args 1 2)
1 2 and 0 extras: ()
CLIPS> (print-args a b c d)
```

a b and 2 extras: (c d)

Return

(return [<expression>])

CLIPS>

```
(deffunction sign (?num)
  (if (> ?num 0) then
                (return 1))
  (if (< ?num 0) then
                (return -1))
  0)
```

CLIPS> (sign 5)

1

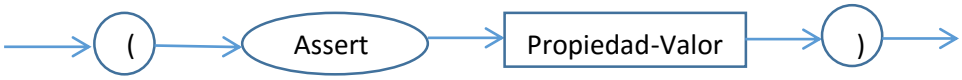
CLIPS> (sign -10)

-1

CLIPS> (sign 0)

0

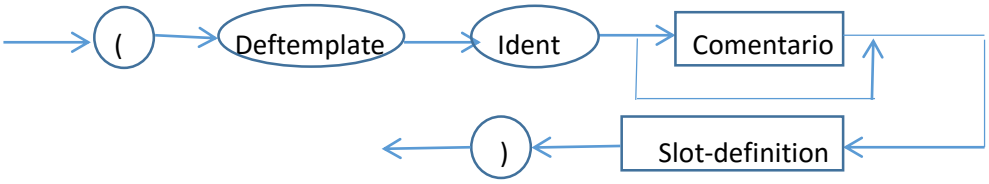
Assert -> funcion – call (general)



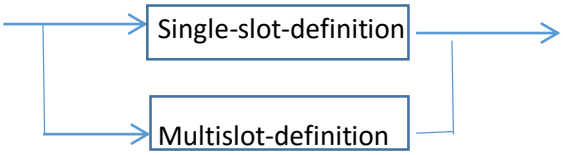
Retract

Propiedad-valor

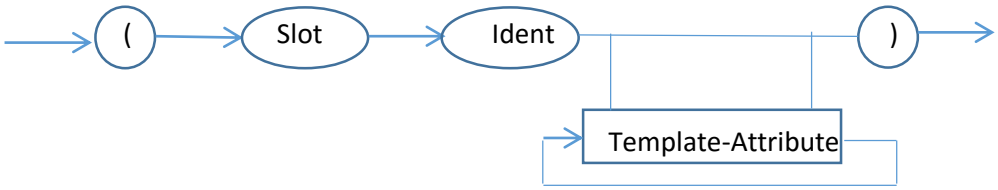
Deftemplate



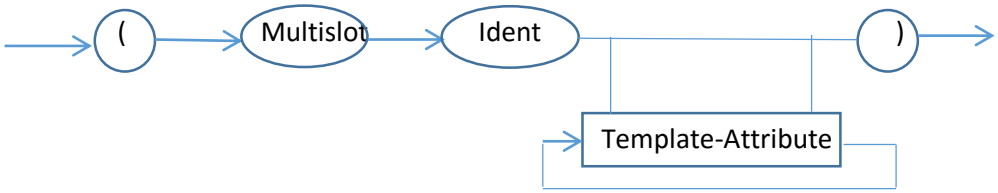
Slot-definition



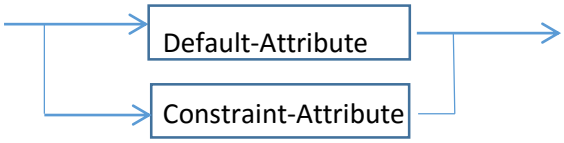
Sinlge-slot-definition



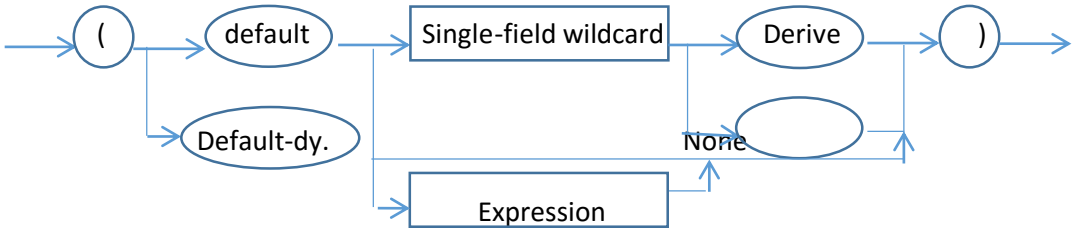
Multislot-definition



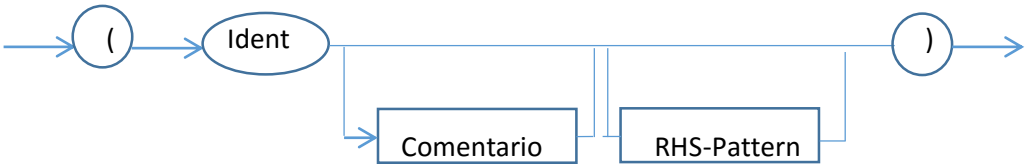
Template-attribute



Default-attribute

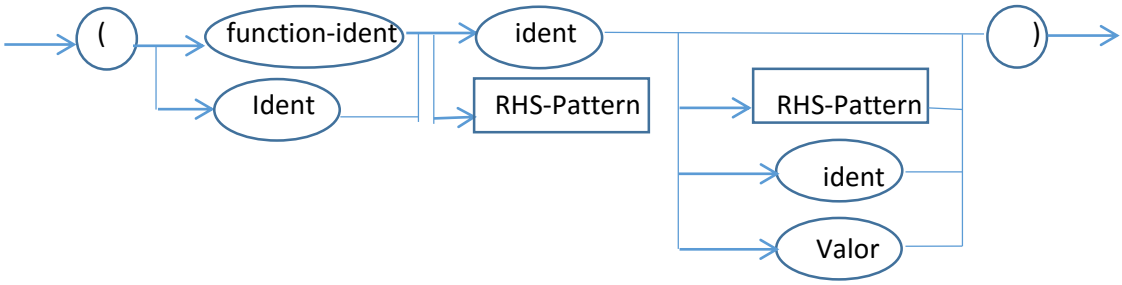


Deffacts

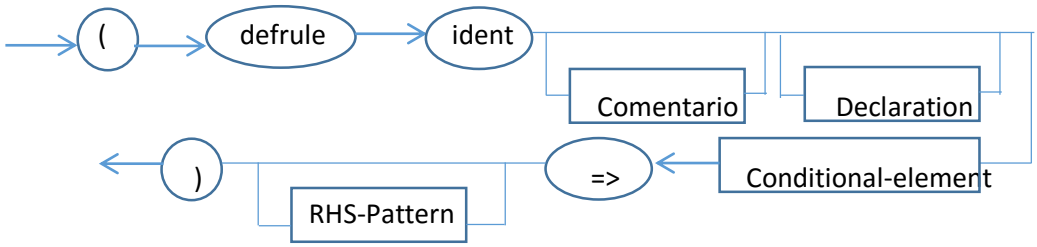


Comentario

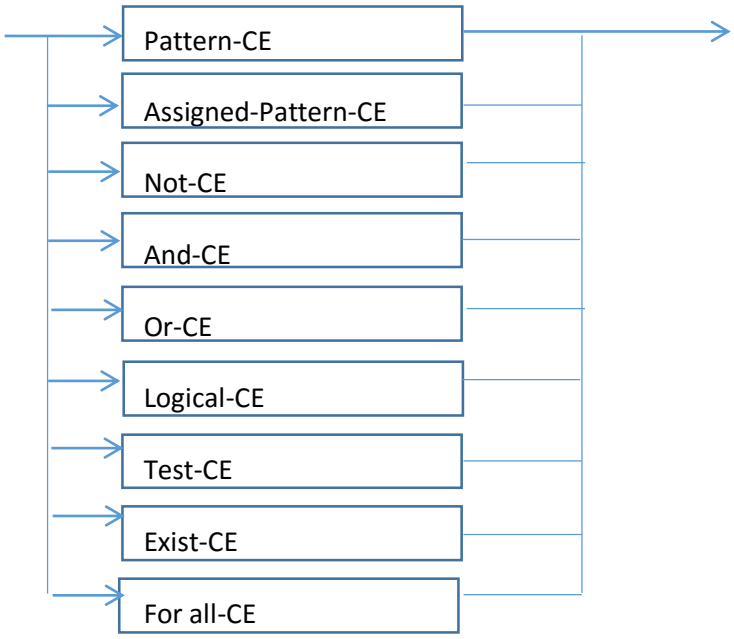
RHS-Pattern |function-call



Defrule

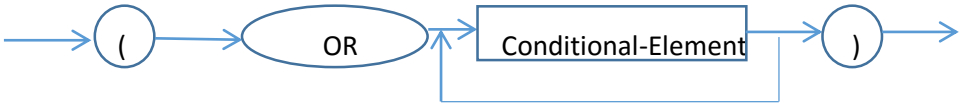


Conditional-element

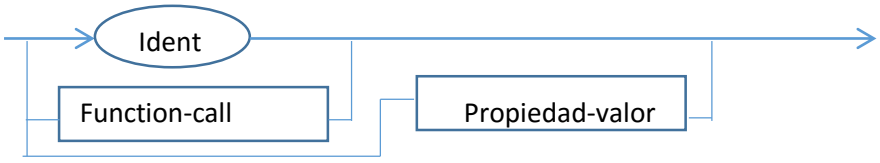


Test-CE

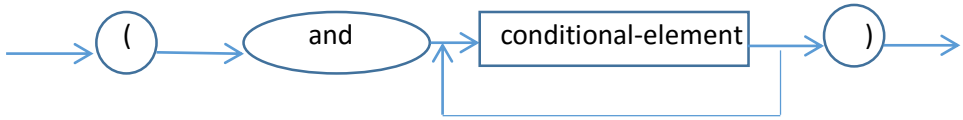
Or-CE



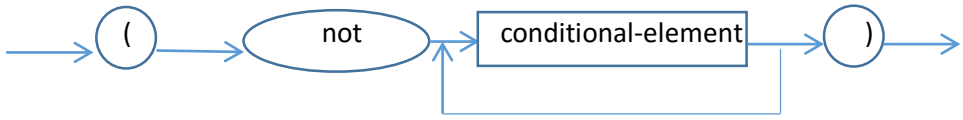
Conditional-element



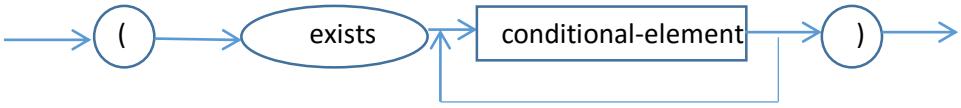
And-CE



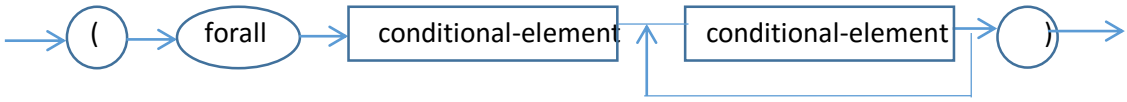
Not-CE



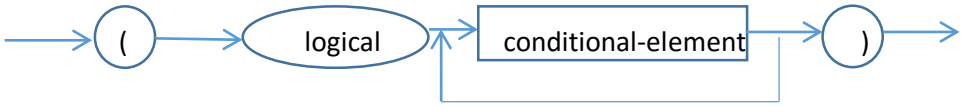
Exists-CE



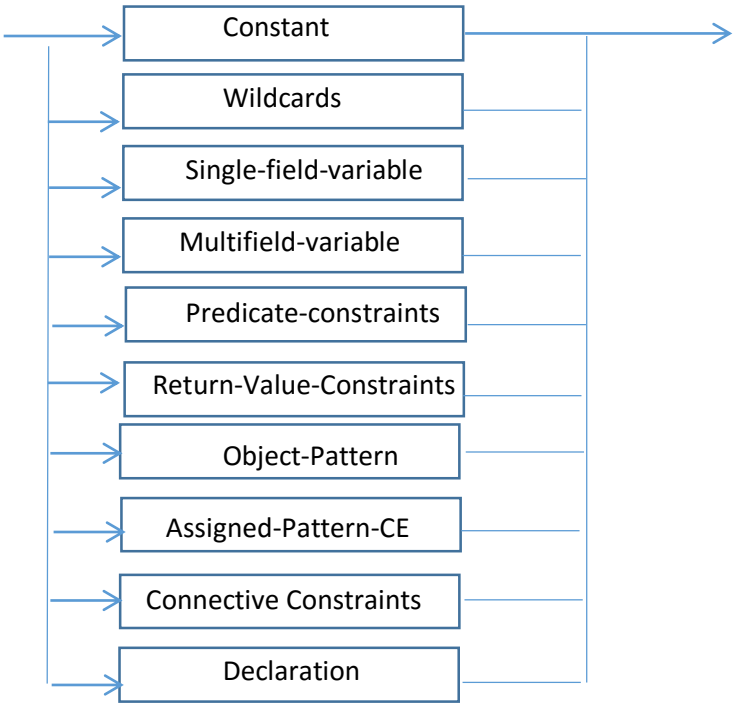
For all-CE



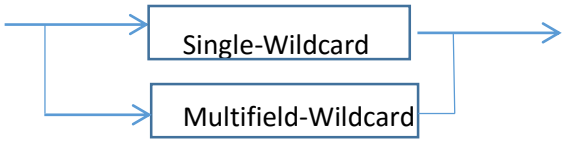
Logical-CE



Pattern-CE



Wildards



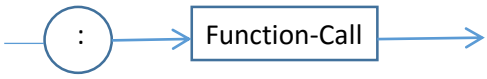
Single-wildcard

Multifield-wildcard

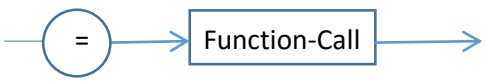
Single-field-variable

Multifield-wildcard

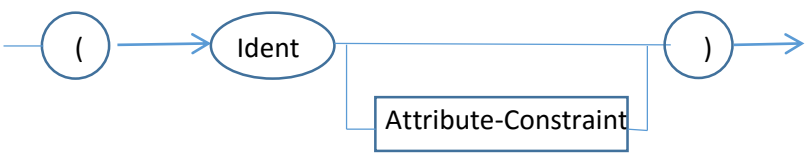
Predicate-constraints



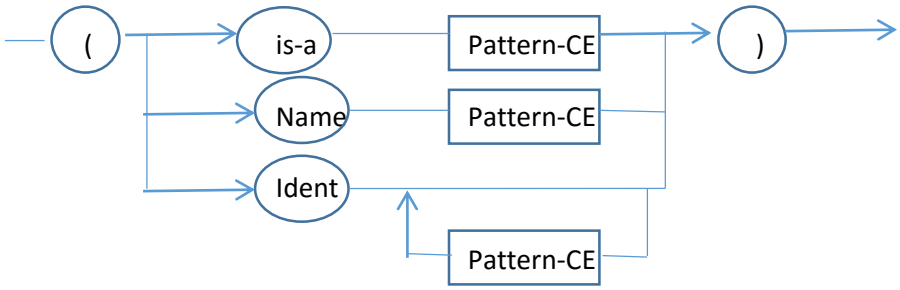
Return-value-constraints



Object-pattern

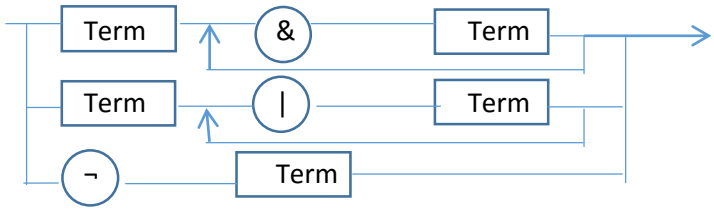


Attribute-constraint

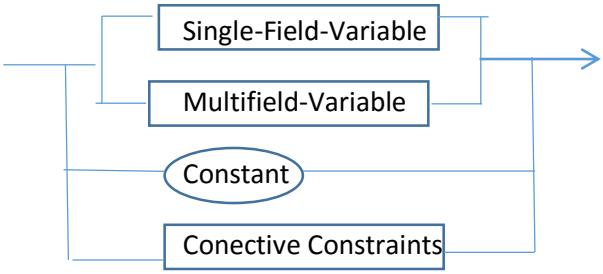


Assigned-pattern-CE

Conective constraints



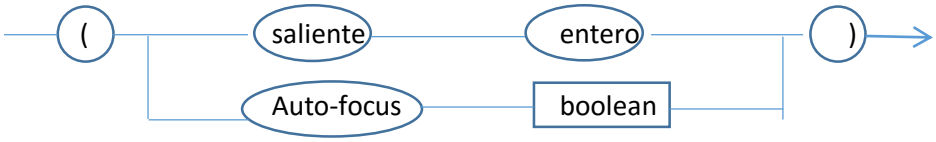
Term



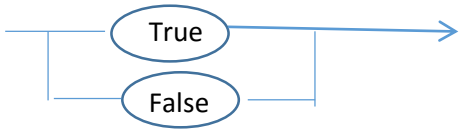
Constraint

Declaration

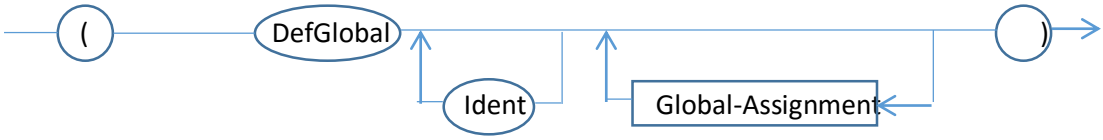
Rule property



Boolean

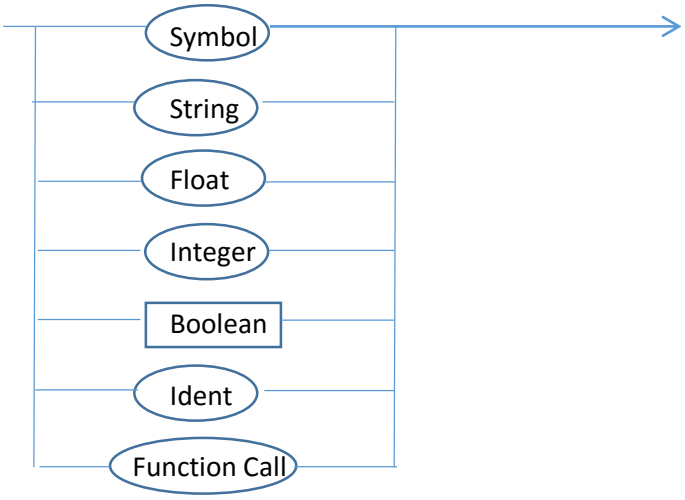


DefGlobal

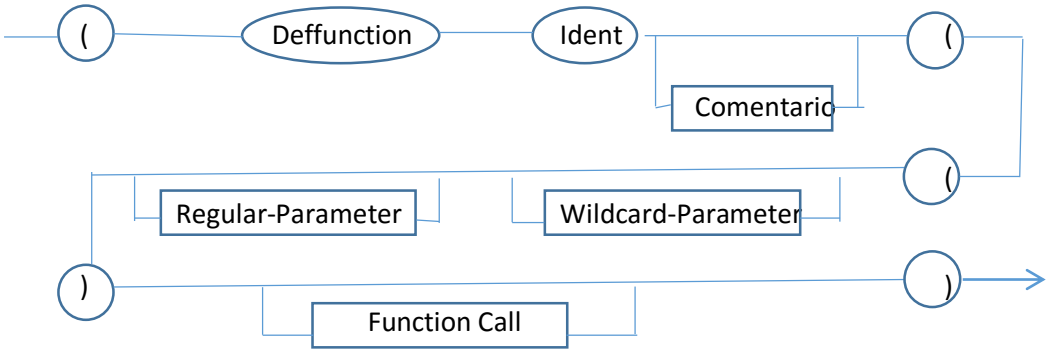


Global-Assignment

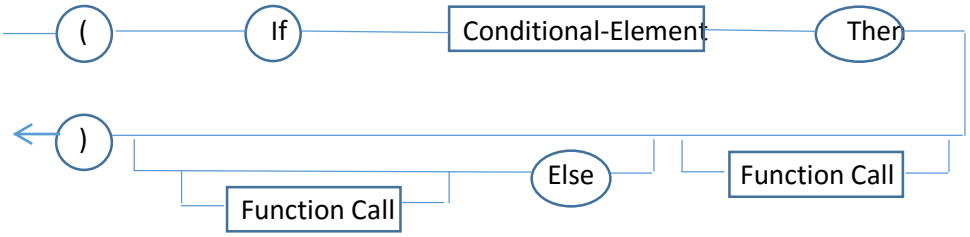
Expression



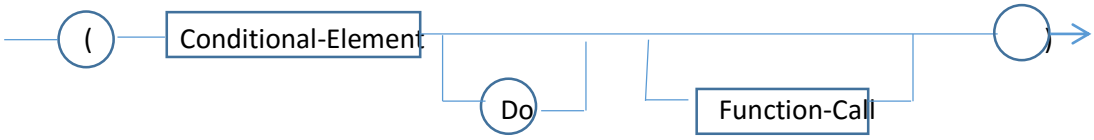
Deffunction



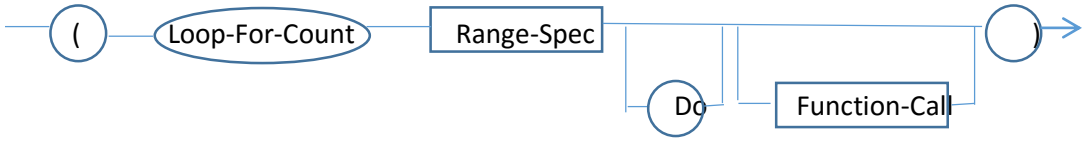
If



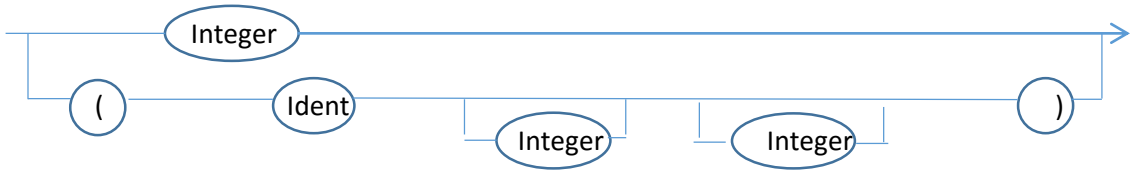
While



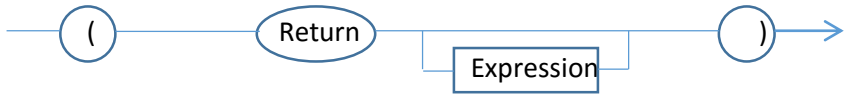
Loop-for-count



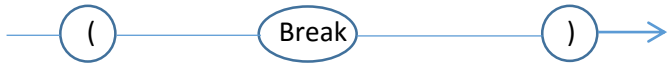
Range-Spec



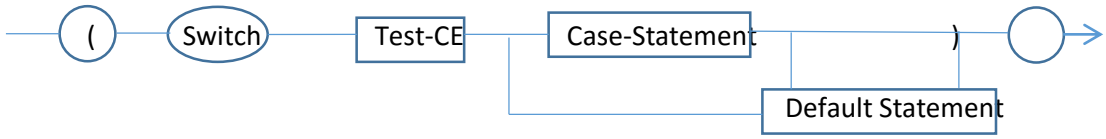
Return



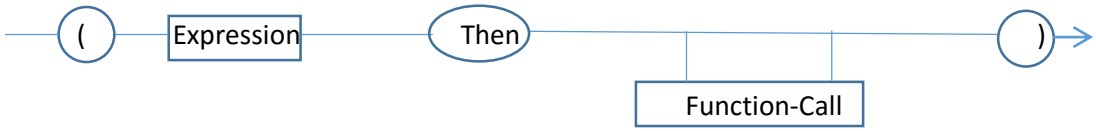
Break



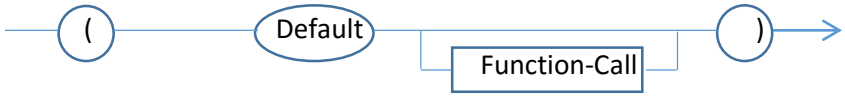
Switch



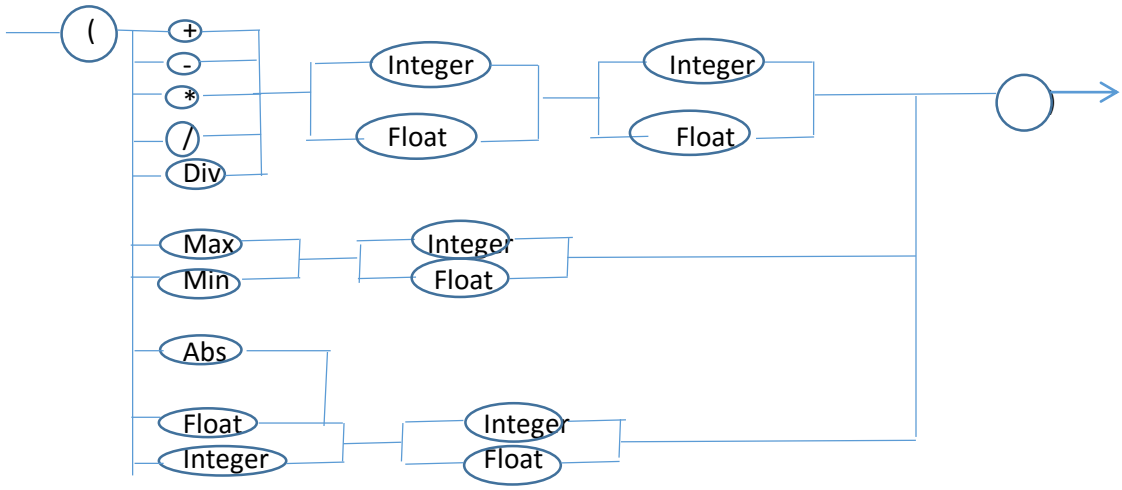
Case-Statement



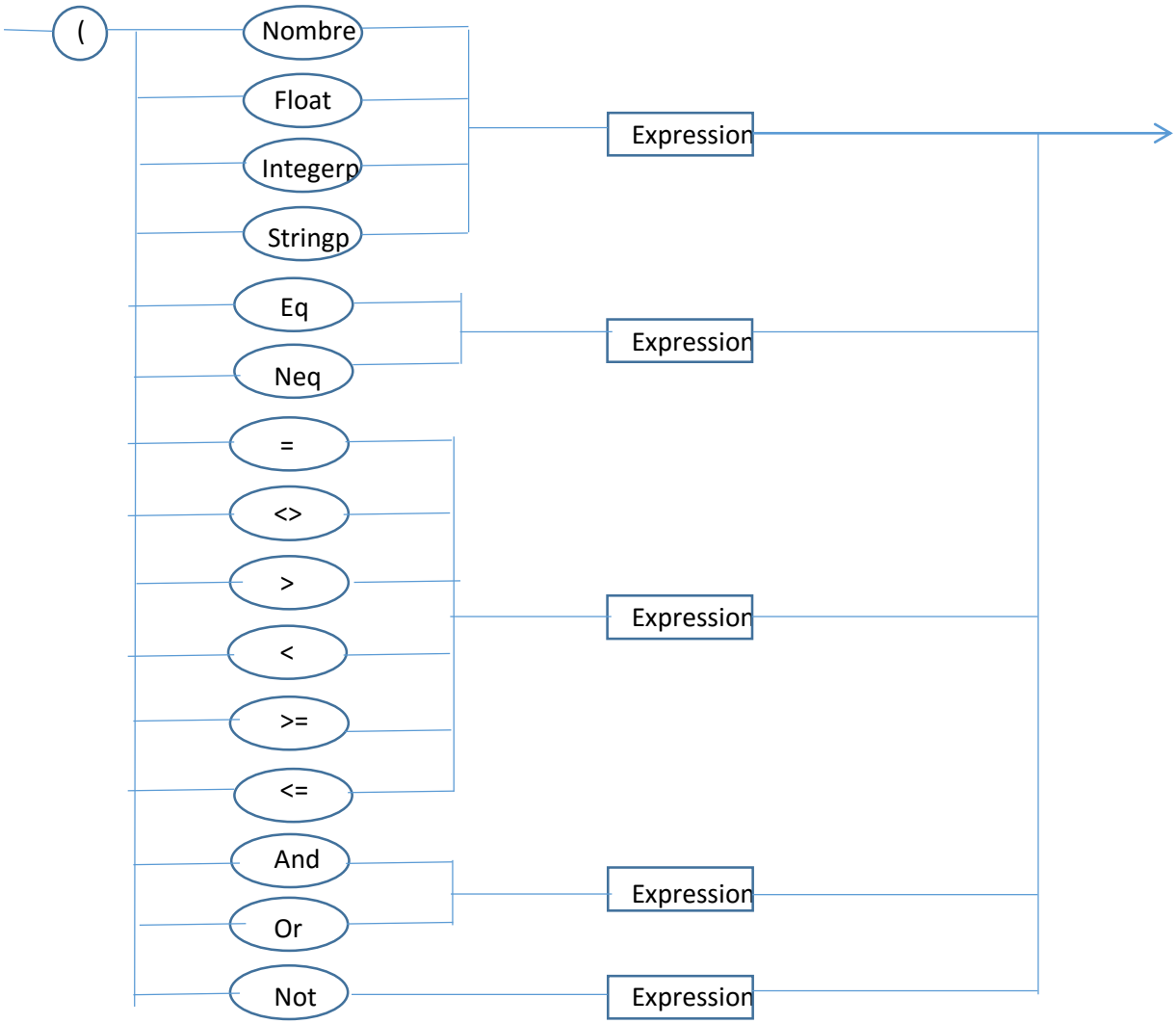
Default-Statement



Math Functions

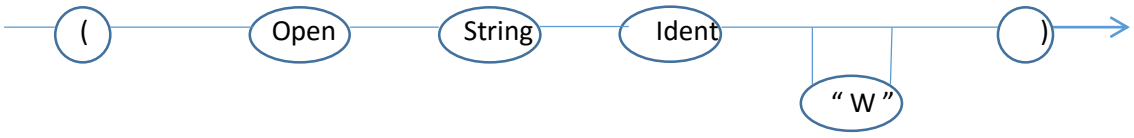


Predicate Functions

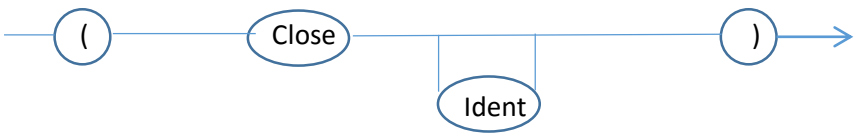


Common 1/0 function

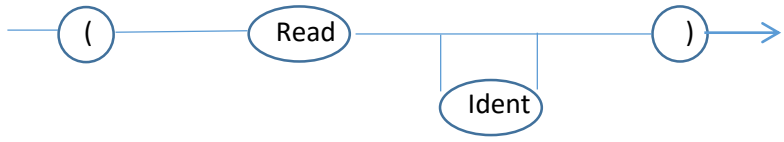
Open



Close

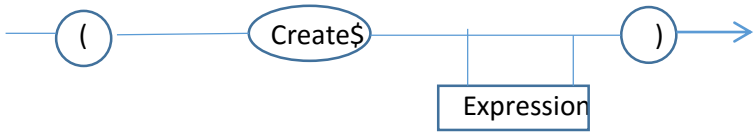


Read

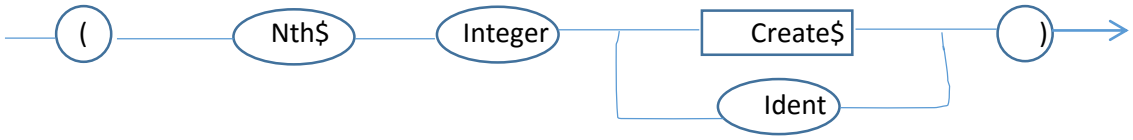


Multifield Functions

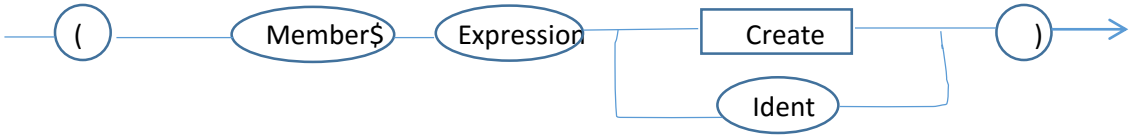
Create$



Specifying

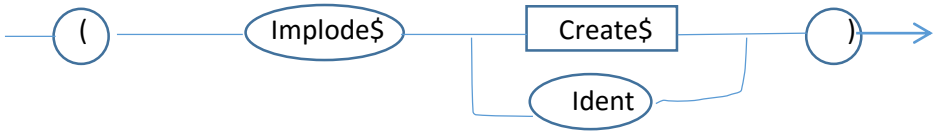


Member$

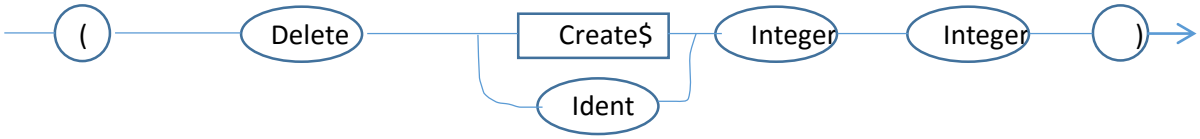


Explode$

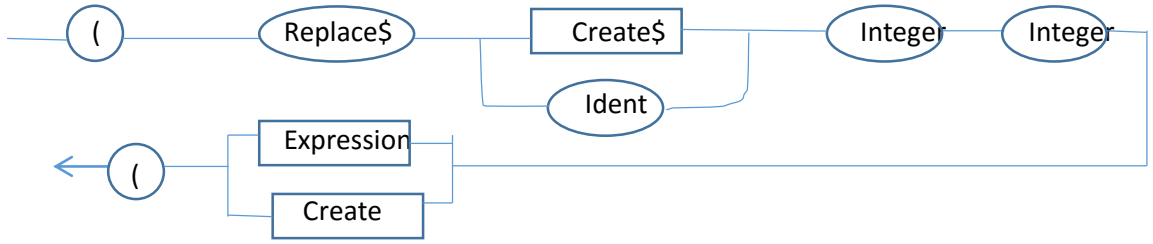
Implode$



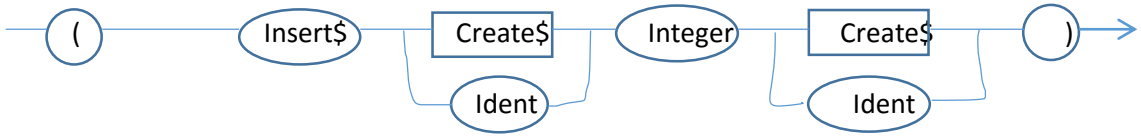
Delete$



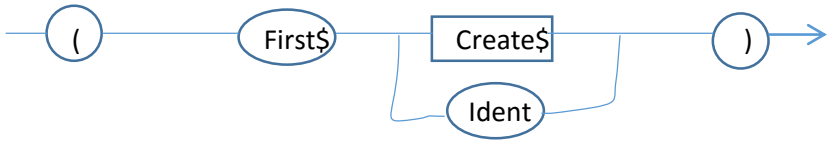
Replace$



Insert$



First$



Rest$