Corso di Laurea in INFORMATICA

INGEGNERIA DELLA CONOSCENZA E SISTEMI ESPERTI (a.a. 2010-2011)

MODULO 2 – Parte I

1.4

CLIPS: C Language Integrated Production System

Funzioni e Costrutti Procedurali

Conoscenza procedurale

Il CLIPS supporta anche un paradigma procedurale per la rappresentazione della conoscenza come quello dei linguaggi Pascal e C.

Le **funzioni (deffunctions)** e le **funzioni generiche** (generic functions) permettono all'utente di definire nuovi elementi eseguibili che provocano un effetto collaterale oppure che restituiscono un valore.

I message-handlers permettono all'utente di definire il comportamento degli oggetti specificando la loro risposta ai messaggi.

Deffunctions, generic functions e message-handlers sono porzioni di codice procedurale specificati dall'utente che il CLIPS esegue quando opportuno.

Defmodules permette di partizionare la base di conoscenza

Elementi di base

Una **funzione** in CLIPS è una porzione di codice eseguibile identificata da uno specifico nome che restituisce un valore o ha un effetto collaterale.

Il costrutto **deffunction** permette all'utente di definire nuove funzioni, direttamente nell'ambiente CLIPS.

COOL

Le funzioni generiche possono essere definite utilizzando i costrutti **defgeneric** e **defmethod**. Questo tipo di funzioni permettono l'esecuzione di diverse porzioni di codice in base agli argomenti passati alla funzione. Quindi, un singolo nome di funzione può essere **sovraccaricato** con più porzioni di codice.

Clips utilizza la *notazione prefissa* per le funzioni

(* 8 (+ 3 (* 2 3 4) 9) (* 3 4))

Deffunction

Le deffunctions sono equivalenti alle altre funzioni, l'unica differenza è che le deffunctions sono scritte in CLIPS ed interpretate, mentre le funzioni esterne sono scritte in un linguaggio esterno (come il C) ed eseguite direttamente dal CLIPS.

Le deffunctions permetto l'aggiunta di nuove funzioni senza ricompilare il CLIPS.

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

Una deffunction deve avere un unico nome, non esiste l'overloading, e deve essere dichiarata prima di essere usata, ad eccezioni delle deffunctions ricorsive.

Deffunction

```
CLIPS> (clear)
CLIPS>
(deffunction stampa-argomenti (?a ?b $?c)
       (printout t ?a " " ?b " e " (length ?c)
                 " aggiuntivo: " ?c crlf))
CLIPS> (stampa-argomenti 1 2)
1 2 e 0 aggiuntivo: ()
CLIPS> (stampa-argomenti a b c d)
a b e 2 aggiuntivo: (c d)
CLIPS>
```

Quando una deffunction viene invocata, le sue azioni sono eseguite nell'ordine.

Il valore restituito da una deffunction è la valutazione dell'ultima azione.

Se una deffunction non ha azioni essa restituisce il simbolo FALSE.

Se si verifica un errore tutte le azioni non ancora eseguite verranno abortite, e verrà restituito il simbolo FALSE.

Costrutti Procedurali

If...then...else

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

Esempio:

```
(defrule valvola-chiusa
 (temperatura alta)
 (valvola ?v chiusa)
 =>
 (if (= ?v 6)
  then
   (printout t "La valvola speciale " ?v " è chiusa!"
crlf)
   (assert (esegui operazione speciale))
  else
   (printout t "La Valvola " ?v " è normalmente chiusa"
crlf)))
```

Costrutti Procedurali

While

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

```
(defrule valvole-aperte
   (valvola aperta ?v)
   =>
   (while (> ?v 0)
      (printout t "Valvola " ?v " è aperta" crlf)
      (bind ?v (- ?v 1))))
```

Costrutti Procedurali

Loop-for-count

```
(loop-for-count <range-spec> [do] <action>*)
<range-spec> ::= <end-index> |
    (<loop-variable> <start-index> <end-index>) |
    (<loop-variable> <end-index>)
<start-index> ::= <integer-expression>
<end-index> ::= <integer-expression>
```

Esempio:

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


CLIPS>
(loop-for-count (?cnt1 2 4) do
    (loop-for-count (?cnt2 1 3) do
        (printout t ?cnt1 " " ?cnt2 crlf)))
```

```
2 1
2 2
2 3
3 1
3 2
3 3
4 1
4 2
4 3
FALSE
CLIPS>
```

Costrutti Procedurali

```
(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
CLIPS>
```

Costrutti Procedurali

(break)

```
CLIPS>
(deffunction iterate (?num)
   (bind ?i 0)
   (while TRUE do
      (if (>= ?i ?num) then
         (break))
      (printout t ?i " ")
      (bind ?i (+ ?i 1)))
      (printout t crlf))
CLIPS> (iterate 1)
0
CLIPS> (iterate 10)
0 1 2 3 4 5 6 7 8 9
CLIPS>
```

Costrutti Procedurali

```
(switch <test-expression> <case-statement>*
       [<default-statement>])
<case-statement> ::=
       (case <comparison-expression> then <action>*)
<default-statement> ::= (default <action>*)
```

Esempio:

```
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
CLIPS>
```