Università degli Studi di Bari
Dipartimento di Informatica

Laboratorio di ICSE

CLIPS

- Parte 3 -

Claudia d'Amato
claudia.damato@di.uniba.it

Claudio Taranto
claudio.taranto@di.uniba.it

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

```
(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)))
```

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))))
```

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>
```

(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>
```

(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: funzioni utili su Multifield

Creazione di un valore multifield

(create$ <expression>*)

CLIPS> (create$ hammer drill saw screw pliers wrench)

(hammer drill saw screw pliers wrench)

CLIPS> (create$ (+ 3 4) (* 2 3) (/ 8 4))

(7 6 2)

Restituzione di un campo specificato all'interno di un multifield

(nth$ <integer-expression> <multifield-expression>)

Interrogazione se un singolo valore è in un multifield

(member$ <expression> <multifield-expression>)

Es. CLIPS> (member$ blue (create$ red 3 "text" 8.7 blue)) 5

Verifica se un multifield è sottoinsieme di un altro multifield

(subsetp <multifield-expression> <multifield-expression>)

CLIPS: funzioni utili su Multifield

Cancellazione di campi in un multifield

(delete$ <multifield-expression>
<begin-integer-expression>
<end-integer-expression>)

Esempio

```
CLIPS> (delete$ (create$ hammer drill saw pliers wrench) 3 4)
(hammer drill wrench)
CLIPS> (delete$ (create$ computer printer hard-disk) 1 1)
(printer hard-disk)
```

Creazione di un Multifield da Stringhe

(explode$ <string-expression>)

Esempio

```
CLIPS> (explode$ "hammer drill saw screw")
(hammer drill saw screw)
CLIPS> (explode$ "1 2 abc 3 4 \"abc\" \"def\"")
(1 2 abc 3 4 "abc" "def")
CLIPS> (explode$ "?x ~ )")
("?x" "~" ")")
```

CLIPS: funzioni utili su Multifield

Creazione di stringhe da valori di un Multifield

(implode$ <multifield-expression>)

Esempio

CLIPS> (implode$ (create$ hammer drill screwdriver))

"hammer drill screwdriver“

CLIPS> (implode$ (create$ 1 "abc" def "ghi" 2))

"1 "abc" def "ghi" 2“

CLIPS> (implode$ (create$ "abc def ghi"))

""abc def ghi"“

Estrazione di una sottosequenza da un Multifield

(subseq$ <multifield-value>

<begin-integer-expression>

<end-integer-expression>)

CLIPS> (subseq$ (create$ hammer drill wrench pliers) 3 4)

(wrench pliers)

CLIPS> (subseq$ (create$ 1 "abc" def "ghi" 2) 1 1)

(1)

CLIPS: funzioni utili su Multifield

Rimpiazzamento di campi in un Multifield

(replace$ <multifield-expression>
<begin-integer-expression> <end-integer-expression>
<single-or-multi-field-expression>+)

Esempio

CLIPS> (replace$ (create$ drill wrench pliers) 3 3 machete)
(drill wrench machete)
CLIPS>(replace$ (create$ a b c d) 2 3 x y (create$ q r s))
(a x y q r s d)
CLIPS>

Inserimento di campi in un Multifield

(insert$ <multifield-expression> <integer-expression>
<single-or-multi-field-expression>+)

Esempio

CLIPS> (insert$ (create$ a b c d) 1 x)
(x a b c d)
CLIPS> (insert$ (create$ a b c d) 4 y z)
(a b c y z d)
CLIPS> (insert$ (create$ a b c d) 5 (create$ q r))
(a b c d q r)

CLIPS: funzioni utili su Multifield

Richiesta del Primo Elemento di un Multifield

(first$ <multifield-expression>)

Esempio

```
CLIPS> (first$ (create$ a b c))
       (a)
CLIPS> (first$ (create$))
       ()
CLIPS>
```

Restituzioni di tutti gli elementi di un Multifield tranne il primo

(rest$ <multifield-expression>)

Esempio

```
CLIPS> (rest$ (create$ a b c))
       (b c)
CLIPS> (rest$ (create$))
       ()
CLIPS>
```

CLIPS: funzioni utili su Multifield

Determinazione del numero di campi in un Multifield

```
CLIPS> (length$ (create$ a b c d e f g))
7
CLIPS>
```

Cancellazione di Specifici Valori in un Multifield

```
CLIPS> (delete-member$ (create$ a b a c) b a)
(c)
CLIPS> (delete-member$ (create$ a b c c b a) (create$ b a))
(a b c c)
CLIPS>
```

CLIPS: funzioni utili su Stringhe

String Functions

(str-cat <expression>*) → Concatenazione Stringhe

(sym-cat <expression>*) → Concatenazione Symbol

(sub-string <integer-expression> <integer-expression> <string-expression>)

(str-index <lexeme-expression> <lexeme-expression>) → ritorna la posizione di una stringa in un'altra

(eval <string-or-symbol-expression>)

(build <string-or-symbol-expression>)

(upcase <string-or-symbol-expression>)

(lowcase <string-or-symbol-expression>)

(str-compare <string-or-symbol-expression> <string-or-symbol-expression>)

(str-length <string-or-symbol-expression>)

CLIPS I/O

Logical Names: stdin, stdout, wclips, wdialog, wdisplay, werror, wwarning, wtrace.

Common I/O Functions: open, close, printout, read, readline, format, rename, remove

(open <file-name> <logical-name> [<mode>])
(close [<logical-name>])
(printout <logical-name> <expression>*)
(read [<logical-name>])
(readline [<logical-name>])
(format <logical-name> <string-expression> <expression>*)
(rename <old-file-name> <new-file-name>)
(remove <file-name>)

I moduli

CLIPS supporta lo sviluppo e l'esecuzione modulare della Conoscenza.

I moduli CLIPS, creati con il costrutto defmodule, permettono il raggruppamento di un insieme di costrutti in modo da controllare esplicitamente l'accesso ai costrutti da altri moduli.

Questo tipo di controllo è simile allo scope globale e locale utilizzato in linguaggi come il C o Ada.

Restringendo l'accesso ai costrutti deftemplate e defclass, i moduli possono funzionare come blackboards, permettendo solo a certi fatti e a certe istanze di essere visibili in altri moduli.

I moduli sono usati anche da regole per fornire il controllo dell'esecuzione.

I moduli: sintassi

(defmodule <module-name> [<comment>] <port-spec>*)
<port-spec> ::=
    (export <port-item>) |
    (import <module-name> <port-item>)
<port-item> ::=
    ?ALL |
    ?NONE |
    <port-construct> ?ALL |
    <port-construct> ?NONE |
    <port-construct> <construct-name>+
<port-construct> ::=
    deftemplate |
    defclass |
    defglobal |
    deffunction |
    defgeneric

I moduli: tipi di esportazione

Export specifications

Un modulo può esportare tutti i suoi costrutti visibili: export? ALL

Un modulo può esportare tutti i suoi costrutti visibili di un tipo particolare: export name-of-the-construct ?ALL

Un modulo può esportare specifici costrutti visibili:
export name-of-the-construct-type
name-of-one-or-more-visible-constructs-of-the-specified-type

(defmodule A (export ?ALL))
(defmodule B (export deftemplate ?ALL))
(defmodule C (export defglobal foo bar yak))

I moduli: tipi di input

Import Specification

Un modulo può importare tutti i costrutti visibili di un altro modulo:

import module-name?ALL

Un modulo può importare tutti i costrutti visibili di un tipo particolare da un altro modulo: import module-name name-of-the-construct-type?ALL

Un modulo può importare specifici costrutti visibili di un altro modulo:

import module-name name-of-the-construct-type

name-of-one-or-more-visible-constructs-of-the-specified-type

(defmodule A (import D ?ALL))

(defmodule B (import D deftemplate ?ALL))

(defmodule C (import D defglobal foo bar yak))

I moduli e la visibilità dei fatti

I fatti e le istanze appartengono ai moduli in cui sono definiti i corrispondenti deftemplate e defclass (e non ai moduli che li creano).

I fatti e le istanze sono visibili nei moduli che importano i corrispondenti deftemplate o defclass. Questo permette una base di conoscenza partizionata in modo che le regole e gli altri costrutti possono vedere solo le istanze e i fatti cui sono interessati.

Il deftemplate initial-fact deve essere esplicitamente importato dal modulo MAIN

I moduli e la visibilità dei fatti: Esempio

```
CLIPS> (clear)
CLIPS> (defmodule A (export deftemplate foo bar))
CLIPS> (deftemplate A::foo (slot x))
CLIPS> (deftemplate A::bar (slot y))
CLIPS> (deffacts A::info (foo (x 3)) (bar (y 4)))
CLIPS> (defmodule B (import A deftemplate foo))
CLIPS> (reset)
CLIPS> (facts A)
f-1 (foo (x 3))
f-2 (bar (y 4))
For a total of 2 facts.
CLIPS> (facts B)
f-1 (foo (x 3)) → Visibile in B perchè importato
For a total of 1 fact.
CLIPS>
```

I moduli: esempio definizione

(defmodule **FOO** → nome modulo che si sta definendo
(import **BAR** ?ALL) → nome modulo da importare
(import YAK deftemplate ?ALL)
(import GOZ defglobal x y z)
(export defclass ?ALL))

(defrule DETECTION::Find-Fault
(sensor (name ?name) (value bad))
=>
(assert (fault (name ?name)))
)

I moduli

Definito il modulo è possibile specificare i costrutti che ne faranno parte.

I costrutti deffacts, deftemplate, defrule, deffunction, defgeneric, defclass, e definstances specificano il modulo in cui saranno presenti come parte del nome.

Il modulo del costrutto defglobal è indicato specificando il nome del modulo dopo la parola chiave defglobal.

```
(defrule FOO::Find-Fault
(sensor (name ?name) (value bad))
=>
(assert (fault (name ?name)))
)
(defglobal DETECTION ?*count* = 0)
```

```
CLIPS> (clear)
CLIPS> (defmodule A)
CLIPS> (defmodule B)
CLIPS> (defrule foo =>)
CLIPS> (defrule A::bar =>)
CLIPS> (list-defrules)
bar
For a total of 1 defrule.
CLIPS> (set-current-module B)
A
CLIPS> (list-defrules)
foo
For a total of 1 defrule.
CLIPS>
```

I moduli

I comandi undefrule (cancellazione di una regola)
e ppdefrule (visualizzazione testo di regola)
richiedono il nome del costrutto sul quale operare.

Con i moduli è possibile avere costrutti con lo stesso nome
ma in moduli diversi.

Module specifier: è la specifica esplicita del modulo di
un nome (un symbol), module-name::.

Per esempio MAIN::find-stuff, indica che il costrutto find-
stuff è nel modulo MAIN (il modulo standard del CLIPS).
Un modulo può anche essere specificato implicitamente
poiché esiste sempre un modulo “corrente”.
Il modulo corrente viene cambiato ogni volta che si utilizza
il costrutto defmodule oppure la funzione set-
current-module

I moduli

```
CLIPS> (clear)
CLIPS> (defmodule A)
CLIPS> (defglobal A ?*x* = 0)
CLIPS> (defmodule B)
CLIPS> (defglobal B ?*y* = 1)
CLIPS> (ppdefglobal y)
(defglobal B ?*y* = 1)
CLIPS> (ppdefglobal B::y)
(defglobal B ?*y* = 1)
CLIPS> (ppdefglobal x)
[PRNTUTIL1] Unable to find defglobal x
CLIPS> (ppdefglobal A::x)
(defglobal A ?*x* = 0)
CLIPS>
```

I moduli

In generale i costrutti di un modulo non possono essere utilizzati da un altro modulo. Un costrutto si dice visibile in un modulo se il costrutto può essere utilizzato in quel modulo.

Esempio: il modulo B vuole utilizzare il deftemplate foo definito nel modulo A, allora il modulo A deve esportare il deftemplate foo e il modulo B lo deve importare.

```
CLIPS> (clear)
CLIPS> (defmodule A)
CLIPS> (deftemplate A::foo (slot x))
CLIPS> (defmodule B)
CLIPS> (defrule B::bar (foo (x 3)) =>)
[PRNTUTIL2] Syntax Error: Check appropriate syntax for defrule
ERROR:
(defrule B::bar
(foo (
CLIPS> (clear)
CLIPS> (defmodule A (export deftemplate foo))
CLIPS> (deftemplate A::foo (slot x))
CLIPS> (defmodule B (import A deftemplate foo))
CLIPS> (defrule B::bar (foo (x 3)) =>)
CLIPS>
```

I moduli

Ogni modulo ha la propria rete di pattern-matching per le sue regole e la propria agenda.

Quando vien dato il comando run, viene eseguita l'agenda del modulo attivo. L'esecuzione delle regole continua finché non diventa attivo un altro modulo, non ci sono altre regole nell'agenda, oppure viene eseguita la funzione return da una RHS di una regola.

Quando un modulo attivo termina l'esecuzione delle regole in agenda, il current focus viene rimosso dal focus stack e diventa attivo il successivo modulo sul focus stack.

Prima che una regola venga eseguita, il modulo corrente diventa quello in cui la regola è definita.

Il current focus può essere definito utilizzando il comando focus.

```
CLIPS> (clear)
CLIPS> (defmodule MAIN (export ?ALL))
CLIPS>
(defrule MAIN::focus-example
     =>
     (printout t "Firing rule in module MAIN." crlf)
     (focus A B))
CLIPS> (defmodule A (import MAIN deftemplate initial-fact))
CLIPS> (defrule A::example-rule
     =>
     (printout t "Firing rule in module A." crlf))
CLIPS> (defmodule B (import MAIN deftemplate initial-fact))
CLIPS> (defrule B::example-rule
     =>
    (printout t "Firing rule in module B." crlf))
CLIPS> (reset)
CLIPS> (run)
Firing rule in module MAIN.
Firing rule in module A.
Firing rule in module B.
CLIPS>
```