Università degli Studi di Bari
Dipartimento di Informatica

Laboratorio di ICSE

CLIPS

- Parte 2 -

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

Claudio Taranto
claudio.taranto@di.uniba.it

Wildcards Single e Multifield

CLIPS ha due simboli wildcard che possono essere utilizzati per il matching nei campi di un pattern

single-field wildcard, denotato dal simbolo **'?'**, mappa ogni valore memorizzato in esattamente un campo nel pattern entity

multifield wildcard, denotato dai simboli **'$?'**, mappa ogni valore in zero o più campi in un pattern entity

Wildcards Single e Multifield

I Multifield wildcard e i literal constraints possono essere utilizzati insieme per fornire delle potenti capacità di pattern-Matching

Il pattern

```
(data $? YELLOW $?)
```

soddisfa il matching con:

```
(data YELLOW blue red green)
(data YELLOW red)
(data red YELLOW)
(data YELLOW)
(data YELLOW data YELLOW)
```

Wildcards Single e Multifield: Esempio

Data la fact list

```
CLIPS> (facts)
f-0 (initial-fact)
f-1 (data 1.0 blue "red") → NO perchè “red” != red
f-2 (data 1 blue) → NO perchè manca red
f-3 (data 1 blue red) SI
f-4 (data 1 blue RED) → NO perchè RED != red
f-5 (data 1 blue red 6.9) SI
For a total of 6 facts.
```

La seguente regola attiva i fatti:

```
CLIPS> (defrule find-data (data ? blue red $?) =>)
CLIPS> (reset)
CLIPS> (agenda)
0 find-data: f-5
0 find-data: f-3
For a total of 2 activations.
```

Wildcards Single e Multifield: Esempio

```
(deffacts startup
(maschio gianni) (femmina lucia) (maschio luca) (femmina maria)
(genitore gianni lucia) (genitore maria lucia)
)
```

I wildcard

```
(defrule figlio-gianni
    (genitore gianni ?)
    =>
    (printout t "trovato un figlio di Gianni" crlf)
)
CLIPS>(run)
trovato un figlio di Gianni
CLIPS>
```

Variabili Single e Multifield

I simboli wildcard rimpiazzano porzioni di un pattern ed accettano ogni valore.

Il valore di un campo essendo rimpiazzato può essere catturato in una variabile per confronti, visualizzazioni o altre manipolazioni successive.

Questo si ottiene facendo seguire direttamente il wildcard dal nome della variabile.

```
<constraint> ::= <constant> | ? | $? |
<single-field-variable> | <multifield-variable>
<single-field-variable> ::= ?<variable-symbol>
<multifield-variable> ::= $?<variable-symbol>
```

Uso di Variabili Single e Multifield

(maschio gianni) (femmina lucia) (maschio luca) (femmina maria)
(genitore gianni lucia) (genitore maria lucia))

Le variabili nei pattern

```
(defrule maschi
    (maschio ?m)
    =>
    (printout t "trovato un maschio" ?m crlf))
CLIPS>(run)
 trovato un maschio gianni
 trovato un maschio luca
CLIPS>
 (defrule madre
    (femmina ?f)
    (genitore ?f ?x)
     =>
    (printout t "trovata madre" ?f “di ” ?x crlf))
```

Variabili Single e Multifield

```
CLIPS> (clear)
CLIPS> (reset)
CLIPS> (assert (data 2 blue green) (data 1 blue)
(data 1 blue red))
<Fact-3>
CLIPS> (facts)
f-0 (initial-fact)
f-1 (data 2 blue green)
f-2 (data 1 blue)
f-3 (data 1 blue red)
For a total of 4 facts.
CLIPS> (defrule find-data-1 (data ?x ?y ?z) =>
(printout t ?x " : " ?y " : " ?z crlf))
CLIPS> (run)
1 : blue : red
2 : blue : green
```

Variabili Single e Multifield

```
CLIPS> (reset)
CLIPS> (assert (data 1 blue) (data 1 blue red)
(data 1 blue red 6.9))
<Fact-3>
CLIPS> (facts)
f-0 (initial-fact)
f-1 (data 1 blue)
f-2 (data 1 blue red)
f-3 (data 1 blue red 6.9)
For a total of 4 facts.
CLIPS>
(defrule find-data-1 (data ?x $?y ?z) =>
(printout t "?x = " ?x crlf "?y = " ?y crlf
"?z = " ?z crlf "------" crlf))
CLIPS> (run)
```

continua...

Variabili Single e Multifield

continua...

```
?x = 1
?y = (blue red)
?z = 6.9
------
?x = 1
?y = (blue)
?z = red
------
?x = 1
?y = ()
?z = blue
------
CLIPS>
```

Variabili Single e Multifield

```
CLIPS> (clear)
CLIPS> (deffacts data (data red green) (data purple blue)
(data purple green) (data red blue green)
(data purple blue green) (data purple blue brown))
CLIPS> (defrule find-data-1 (data red ?x) (data purple ?x) =>)
CLIPS> (defrule find-data-2 (data red $?x) (data purple $?x)
=>)
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (data red green)
f-2 (data purple blue)
f-3 (data purple green)
f-4 (data red blue green)
f-5 (data purple blue green)
f-6 (data purple blue brown)
For a total of 7 facts.
```

Variabili Single e Multifield

```
CLIPS> (agenda)

0 find-data-2: f-4,f-5
0 find-data-1: f-1,f-3
0 find-data-2: f-1,f-3

For a total of 3 activations.
CLIPS>
```

Indirizzi

Alcune azioni RHS, ritrattano fatti (retract command)

Per indicare su quale fatto agire, può essere unificata una variabile con i fact-address di un pattern CE.

```
<assigned-pattern-CE> ::=
?<variable-symbol> <- <pattern-CE>
```

Esempio

```
(defrule dummy
     (data 1)
     ?fact <- (dummy pattern)
     =>
     (retract ?fact))
```

LHS di una Regola: Sintassi Completa

Sintassi

<conditional-element> ::= <pattern-CE> | <assigned-pattern-CE> |
<not-CE> | <and-CE> | <or-CE> |<logical-CE> |
<test-CE> |<exists-CE> | <forall-CE>

<pattern-CE> ::= <ordered-pattern-CE> |<template-pattern-CE> |<object-Pattern-CE>

<assigned-pattern-CE> ::= ?<variable-symbol> <- <pattern-CE>

<test-CE> ::= (test <function-call>)
<not-CE> ::= (not <conditional-element>)
<and-CE> ::= (and <conditional-element>+)
<or-CE> ::= (or <conditional-element>+)
<exists-CE> ::= (exists <conditional-element>+)
<forall-CE> ::= (forall <conditional-element><conditional-element>+)
<logical-CE> ::= (logical <conditional-element>+)

Not-CE

Il **not** conditional element fornisce la possibilità di rappresentare l’assenza di informazione.

Il not CE è soddisfatto solo se il conditional element che contiene non è soddisfatto.

<not-CE> ::= (not <conditional-element>)

```
(defrule temperatura-alta
   (temperatura alta)
   (valvola aperta)
   (not (error-status confermato))
   =>
   (printout t "Raccomandazione: chiudere valvola causa
alta temperatura" crlf))

(defrule controlla-valvola
   (check-status ?valvola)
   (not (valvola-rotta ?valvola))
   =>
   (printout t "Dispositivo " ?valve " è OK" crlf))
```

Or-CE

Il CE **or** consente l'attivazione di una regola se **almeno uno** dei CE all'interno del CE or è soddisfatto.

Il CE or ha lo stesso effetto di scrivere più regole con LHS e RHS simili.

```
<or-CE> ::= (or <conditional-element>+)
(defrule guasto-sistema
   (error-status sconosciuto)
   (or (temperatura alta)
       (valvola rotta)
       (pompa (status off))
    )
  =>
(printout t "Il Sistema ha un guasto." crlf))
```

And-CE

Il CLIPS assume che tutte le regole abbiano un **and-CE implicito** su tutti i conditional elements della LHS.

<===> Tutti i conditional elements nella LHS devono essere soddisfatti per poter attivare la regola.

Il CLIPS fornisce anche un and CE esplicito .

```
<and-CE> ::= (and <conditional-element>+)
(defrule controllo-sistema
     (error-status confermato)
     (or (and (temperatura alta)
              (valvola chiusa) )
         (and (temperatura bassa)
              (valvola aperta) ))
     =>
     (printout t "Il sistema ha un problema di
     funzionamento"))
```

Test-CE

```
<test-CE> ::= (test <function-call>)
```

Il test CE è soddisfatto se la funzione chiamata restituisce non-FALSE
è insoddisfatto se la funzione chiamata restituisce FALSE

```
CLIPS> (clear)
CLIPS> (defrule example-1
(data ?x) (value ?y)
(test (>= (abs (- ?y ?x)) 3)) =>)
CLIPS> (assert (data 6) (value 9))
<Fact-1>
CLIPS> (agenda)
0 example-1: f-0,f-1
For a total of 1 activation.
CLIPS>
```

Exists-CE

Un exists CE è vero se esiste almeno un fatto corrispondente al pattern.
exists CE è utile quando si vuole che una regola venga attivata solo una volta (sebbene possono potenzialmente esistere molti fatti che la possano attivare)

```
(defrule exists-demo
     (exists (onesto ?))
     =>
    (printout t "Esiste almeno un uomo onesto!" crlf))
```

Se esiste un uomo onesto allora la regola sarà attivata una ed una sola volta (non sarà attivata per tutti gli altri che potenzialmente esistono)
exists CE non può essere combinato nello stesso pattern con un test CE

N.B: exists è equivalente (ed è implementato come) due not CE nidificati es. (exists (A)) equivale a (not (not (A)))

Forall-CE

forall CE fornisce un meccanismo per determinare se un gruppo di CE è soddisfatto per ogni occorrenza di un altro specificato CE.

Utilizzato quando si vogliono esprimere regole che vengano attivate una sola volta nel caso in cui esistano situazioni del tipo “per ogni fatto (a ?x) esiste un fatto (b ?x)”

```
<forall-CE> ::= (forall <conditional-element>
<conditional-element>+)
(defrule example
(forall (a ?x) (b ?x) (c ?x))
=>)
```

Vincoli su Pattern Conditional Element

E' possibile definire vincoli all'interno di un singolo Pattern-CE (field constraints)

Esistono diversi tipi di field constraints:

- **literal** constraints
- **connective** constraints
- **predicate** constraints
- **return value** constraints

Literal Constraint

E' il vincolo più elementare che può essere utilizzato in un pattern CE

- è quello che definisce precisamente il valore esatto da mappare con un campo. Tutti i vincoli in un literal pattern devono avere esattamente un matching con tutti i campi di un pattern entity.

Un literal pattern CE è costituito da costanti (floats, integers, symbols, strings, e instance names). Non contiene variabili o wildcards.

Può essere:

- Un ordered pattern CE contenente solo literal:

```
(<constant-1> ... <constant-n>)
```

- Un deftemplate pattern CE contenente solo literal :

```
(<deftemplate-name> (<slot-name-1> <constant-1>)
...
(<slot-name-n> <constant-n>))
```

Connective Constraints

Sono disponibili tre connective constraints per collegare singoli vincoli e variabili.

- & (and): soddisfatto se i due vincoli adiacenti sono soddisfatti;

- | (or): soddisfatto se uno dei due vincoli adiacenti è soddisfatto;

- ~ (not): soddisfatto se il successivo vincolo non si verifica.

```
<term-1>&<term-2> ... &<term-3>
<term-1>|<term-2> ... |<term-3>
~<term>
<constraint> ::= ? | $? | <connected-constraint>
<connected-constraint> ::= <single-constraint> |
<single-constraint> & <connected-constraint> |
<single-constraint> | <connected-constraint>
<single-constraint> ::= <term> | ~<term>
<term> ::= <constant> | <single-field-variable> |
<multifield-variable>
```

Connective Constraints: Esempio

```
CLIPS> (deftemplate data-B (slot value))
CLIPS> (deffacts AB (data-A green) (data-A blue)
(data-B (value red)) (data-B (value blue)))
CLIPS> (defrule example1-1 (data-A ~blue) =>)
CLIPS> (defrule example1-2 (data-B (value ~red&~green)) =>)
CLIPS> (defrule example1-3 (data-B (value green|red)) =>)
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (data-A green)
f-2 (data-A blue)
f-3 (data-B (value red))
f-4 (data-B (value blue))
For a total of 5 facts.
CLIPS> (agenda)
0 example1-1: f-1
0 example1-2: f-4
0 example1-3: f-3
For a total of 3 activations.
```

Connective Constraints: Esempio

```
CLIPS> (clear)
CLIPS> (deftemplate data-B (slot value))

CLIPS> (deffacts B (data-B (value red))
(data-B (value blue)))

CLIPS> (defrule example2-1 (data-B (value ?x&~red&~green))
        =>
        (printout t "?x in example2-1 = " ?x crlf))

CLIPS> (defrule example2-2 (data-B (value ?x&green|red))
        =>
       (printout t "?x in example2-2 = " ?x crlf))

CLIPS> (reset)

CLIPS> (run)
?x in example2-1 = blue
?x in example2-2 = red
```

Predicate Constraints

A volte è necessario vincolare un campo in base alla verità di una data espressione booleana.

Un predicate constraint permette che una predicate function (che ritorna il simbolo FALSE se non soddisfatta e il simbolo non-FALSE se soddisfatta) venga invocata durante il processo di pattern-matching.

Tipicamente, i predicate constraint sono utilizzati in congiunzione con i connective constraint e unificazione di variabile (cioè bisogna unificare la variabile da testare e connetterla al predicate constraint).

```
:<function-call>
<term> ::= <constant> | <single-field-variable> |
<multifield-variable> | :<function-call>
```

Predicate Constraints: Esempio

```
CLIPS> (clear)

CLIPS> (defrule example-1 (data ?x&:(numberp ?x)) =>)

CLIPS> (assert (data 1) (data 2) (data red))
<Fact-2>

CLIPS> (agenda)
0 example-1: f-1
0 example-1: f-0
For a total of 2 activations.
CLIPS>
```

Predicate Constraints: Esempio

```
CLIPS> (clear)
CLIPS> (defrule example-4
(data ?y)
(data ?x&:(> ?x ?y))
=>)
CLIPS> (assert (data 3) ; f-0
(data 5) ; f-1
(data 9)) ; f-2
<Fact-2>
CLIPS> (agenda)
0 example-4: f-0,f-2
0 example-4: f-1,f-2
0 example-4: f-0,f-1
For a total of 3 activations.
CLIPS>
```

Alcune Predicate Functions Predefinite

- (numberp <expression>) *Testing for numbers*
- (floatp <expression>) *Testing for floats*
- (integerp <expression>) *Testing for integers*
- (stringp <expression>) *Testing for Strings*
- (symbolp <expression>) *Testing for Symbols*
- (lexemep <expression>) *Testing for Strings or Symbols*
- (evenp <integer-expression>) *Testing for Even (Pari) Numb*
- (oddp <integer-expression>) *Testing for Odd (Disp.) Numb*
- (multifieldp <expression>) *Testing for Multifield Values*
- (pointerp <expression>) *Testing for External-Address*
- (eq <expression> <expression>+) *Comparing for Equality*
- (neq <expression> <expression>+) *Comparing for Inequality*

Operazioni di Confronto

Comparing Numbers for Equality and Inequality

(= <numeric-expression> <numeric-expression>+)
(<> <numeric-expression> <numeric-expression>+)
(> <numeric-expression> <numeric-expression>+)
(>= <numeric-expression> <numeric-expression>+)
(< <numeric-expression> <numeric-expression>+)
(<= <numeric-expression> <numeric-expression>+)

Clips utilizza la **notazione prefissa** per le funzioni
Es. 1: (* 8 (+ 3 (* 2 3 4) 9) (* 3 4))
Es. 2: (>= ?x 3)

Funzioni Matematiche

Addition:
(+ <numeric-expression> <numeric-expression>+)

Subtraction:
(- <numeric-expression> <numeric-expression>+)

Multiplication:
(* <numeric-expression> <numeric-expression>+)

Division:
(/ <numeric-expression> <numeric-expression>+)

Integer Division:
(div <numeric-expression> <numeric-expression>+)

Maximum and Minimum Numeric Value:
(max <numeric-expression>+)
(min <numeric-expression>+)

Funzioni Matematiche

- (abs <numeric-expression>)
- (float <numeric-expression>)
- (integer <numeric-expression>)
- acos (arccosine), acosh (hyperbolic arccosine), acot(arccotangent), acoth (hyperbolic arccotangent), acsc(arccosecant), acsch (hyperbolic arccosecant), asec(arcsecant), asech (hyperbolic arcsecant), asin(arcsine), asinh (hyperbolic arcsine), atan (arctangent),atanh (hyperbolic arctangent), cos (cosine), cosh(hyperbolic cosine), cot (cotangent), coth (hyperbolic
- tangent), csc (cosecant), csch (hyperbolic cosecant), sec(secant), sech (hyperbolic secant), sin (sine), sinh(hyperbolic sine), tan (tangent), tanh (hyperbolictangent).
- (deg-grad <numeric-expression>), (deg-rad <numeric-expression>),
- (grad-deg <numeric-expression>), (rad-deg <numeric-expression>), (pi)

Funzioni Matematiche

- (sqrt <numeric-expression>)
- (** <numeric-expression> <numeric-expression>)
- (exp <numeric-expression>)
- (log <numeric-expression>)
- (log10 <numeric-expression>)
- (round <numeric-expression>)
- (mod <numeric-expression> <numeric-expression>)

Return Value Constraints

È possibile utilizzare il valore restituito da una funzione esterna per vincolare il valore di un campo.

Il **return value constraint** (=) permette all'utente di chiamare una funzione esterna da un pattern. Il valore è incorporato direttamente nel pattern nella posizione in cui la funzione è stata chiamata, come se fosse stato un literal constraint, ed ogni matching patterns deve mappare questo valore come se la regola fosse stata scritta con questo valore. Si noti che la funzione viene valutata ogni volta che viene controllato il vincolo.

=<function-call>

<term> ::= <constant> | <single-field-variable> |

<multifield-variable> | :<function-call> |

=<function-call>

Return Value Constraints: Esempio

```
CLIPS> (clear)

CLIPS> (deftemplate data (slot x) (slot y))

CLIPS> (defrule twice (data (x ?x) (y =(* 2 ?x))) =>)

CLIPS> (assert (data (x 2) (y 4)) ; f-0
(data (x 3) (y 9))) ; f-1

CLIPS> (agenda)
0 twice: f-0
For a total of 1 activation.

CLIPS>
```