Appunti di programmazione di base sul linguaggio CLIPS

Laboratorio di EINN anno accademico '97/'98
Gruppo composto da: **Correndo Gianluca** **st930411@educ.di.unito.it**
Dellavalle Paola **st930189@educ.di.unito.it**
Raviola Paolo **raviola@educ.di.unito.it**

Introduzione

CLIPS (C Language Integrated Production System) si presenta come un linguaggio a regole di produzione integrato con un linguaggio object-oriented (COOL).
La necessità dello sviluppo di questo nuovo linguaggio da parte della NASA - l'ente spaziale americano - deriva principalmente da alcuni difetti del LISP - linguaggio storicamente usato per lo sviluppo di software in intelligenza artificiale - quali la sua limitata diffusione, la necessità di macchine dedicate e la sua totale mancanza di integrazione con altri linguaggi.
CLIPS ha attraversato diverse fasi di sviluppo partendo da un linguaggio a regole di produzione puro con forward chaining e arrivando infine a presentare in modo integrato anche caratteristiche procedurali e object-oriented.
Nel seguito saranno introdotte brevemente alcune nozioni sulle regole di produzione e sul paradigma Object-Oriented utili per comprendere meglio il linguaggio. In seguito saranno presentati i costrutti che CLIPS offre per la rappresentazione della conoscenza e nozioni su come esso li usa durante l'esecuzione.

Regole di produzione

Le regole di produzione sono un formalismo per la rappresentazione della conoscenza ; esse si compongono di due parti : la parte sinistra (left-hand side o LHS) che descrive le condizioni necessarie per applicare la regola e la parte destra (right-hand side o RHS) che contiene invece le azioni che si devono eseguire e/o le conclusioni che si possono trarre quando si applica la regola.
Come si vede le regole sono un modo dichiarativo per esprimere la cosiddetta conoscenza euristica che si ha su un determinato dominio, la conoscenza del tipo 'quando si hanno queste condizioni allora si può fare questo o si può dedurre che..'. Volendo si possono vedere le regole come un costrutto IF-THEN, anche se questa visione non è del tutto precisa come si vedrà in seguito.
Questo formalismo di rappresentazione della conoscenza è usato nei sistemi a regole di produzione. Essi sono sistemi che, sfruttando questa conoscenza euristica o generica su un particolare dominio, sono in grado di inferire a partire da dati iniziali relativi ad uno specifico problema sul dominio - detti fatti - altri dati; di arrivare cioè - sulla base della conoscenza espressa dalle regole e sui fatti dati - a delle conclusioni. Tipico esempio : diagnosi di una malattia a partire dai sintomi.
L'architettura di questi sistemi è piuttosto semplice : essi sono costituiti dalla Knowledge-base (KB) che è una struttura in cui sono memorizzate tutte le regole di produzione, la working-memory (WM) in cui ci sono i fatti e di un motore inferenziale che, come dice il nome, sfrutta la KB e la WM per inferire nuovi fatti che va ad inserire nella WM.

Sostanzialmente un sistema a regole compie il seguente ciclo :

1. Confronta le regole con i fatti presenti nella WM e attiva - cioè inserisce in un insieme - tutte le istanze di regole le cui condizioni sono soddisfatte dai fatti e che perciò sono potenzialmente applicabili ; si dice che tali regole sono attivate ;
2. Si effettua la conflict-resolution ; si sceglie cioè una sola istanza di regola tra quelle attivate. Si può scegliere tra una vasta gamma di criteri per la scelta della regola da applicare ;
3. Si esegue la regola prescelta ; si eseguono cioè le azioni indicate dalla regola. Queste azioni possono essere un'aggiunta di un nuovo fatto nella WM, oppure la modifica o la cancellazione di un fatto già presente nella WM, oppure ancora l'esecuzione di una specifica azione come per esempio il calcolo di una funzione ;
4. Si ripete il ciclo dal passo 1.

Si noti che come effetto delle azioni della regola applicata, l'insieme delle regole attivate può cambiare : alcune regole cioè possono essere disattivate e altre attivate.
Alla luce di come funziona un sistema a regole si possono vedere le regole come un costrutto WHENEVER-THEN, cioè 'ogni volta che le condizioni espresse sono soddisfatte allora attiva la regola'.

Introduzione al paradigma Object Oriented

Il paradigma OO nasce come estensione del concetto di «tipo di dato astratto». Con esso, si vogliono realizzare due obbiettivi :

- *Incapsulation* : incapsulare nella definizione del dato anche le funzioni o procedure (metodi) da usare per manipolarlo;
- *Information hiding* : possibilità di nascondere l'implementazione del dato all'esterno.

Il concetto fondamentale su cui si basa il paradigma OO è quello di *classe (di oggetti).* Definire un tipo di dato equivale a definire una classe, in altre parole definire un insieme di variabili e di metodi che rappresentano rispettivamente la descrizione ed il comportamento dei dati di quel tipo. In una classe si possono distinguere due parti: una visibile all'esterno (*interfaccia della classe*) e l'altra no. Generalmente la seconda parte corrisponde all'implementazione del dato; in questo modo si ottiene *l'indipendenza della rappresentazione del dato dal suo uso* : i programmi lavorano con «moduli» predefiniti che possono essere modificati in modo indipendente dal programma stesso (variazioni nell'implementazione dell'oggetto non comportano modifiche nel resto del programma).

Col termine *istanza* di una classe s'intende uno specifico oggetto della classe a cui saranno associati valori specifici per le variabili (distinguono le istanze di una classe). L'invocazione di un metodo su un oggetto equivale all'invio di un messaggio all'oggetto stesso, cui seguirà un comportamento identico a quello degli oggetti della sua stessa classe. Un programma, quindi, può essere visto come un insieme di definizioni di classi e di loro istanze, ma anche come un insieme d'oggetti che s'inviano messaggi.

In generale, nel paradigma OO si ha una *gerarchia di classi.* Esiste una relazione ISA tra classi: «*CL1* ISA *CL2»* significa che *CL1* è sottoclasse di *CL2* (oppure *CL2* è sopraclasse di *CL1*). Di conseguenza si ha il concetto di *ereditarietà:* ogni classe eredita dalla sua sopraclasse descrizione (variabili) e comportamento (metodi), con la possibilità di poter ridefinire (*overriding*) alcune di queste proprietà e definirne di nuove. Di conseguenza una classe avrà le variabili e metodi ereditati più un insieme di definizioni locali. L'ereditarietà è un grosso vantaggio, poiché proprietà comuni a più classi possono essere definite una sola volta in una sopraclasse ed utilizzate da più sottoclassi (riuso del software).
Si possono avere due tipi d'ereditarietà: singola (ogni classe ha una sola sopraclasse) o multipla (ogni classe ha una o più sopracclassi). Il secondo caso può portare a conflitti sulle variabili e metodi da ereditare. Per esempio una stessa variabile (o metodo) può essere definita in due sopracclassi; se la definizione è identica non ci sono problemi, altrimenti si deve decidere quale delle due ereditare (servono dei criteri di preferenza). Per evitare questi problemi alcuni linguaggi OO vietano l'ereditarietà multipla.

Per ora abbiamo parlato solo di metodi d'istanza, che s'inviano ai singoli oggetti di una classe. Esistono anche *metodi di classe,* vale a dire messaggi che si mandano ad un'intera classe. I più importanti sono i

costruttori e i *distruttori* il cui compito è rispettivamente di inizializzare e di eliminare le istanze di una classe.
A questo punto è lecito considerare ogni singola classe come istanza di un'altra classe, chiamata *metaclasse*. Quindi, quelli che abbiamo chiamato metodi di classe non saranno altro che i metodi definiti nella metaclasse di cui la classe che consideriamo è un'istanza.

CLIPS

Clips - come ogni altro linguaggio di programmazione - fornisce diversi tipi di dati primitivi per rappresentare le informazioni . Questi tipi sono i classici *float*, *integer*, *string*, a cui si aggiungono i tipi *symbol* (simboli rappresentati da letterali), *external-address* (simili ai puntatori del C), *fact-address* (indirizzo di un fatto presente nella WM), *instance-name* e *instance-address* (rispettivamente nome e indirizzo di una istanza presente in WM).

- Alcuni esempi di numeri interi sono :

578	22	+15	-64

- Esempi di numeri float sono :

237e3	15.09	+12.0	-32.3e-7

- Una stringa è una sequenza di caratteri delimitata da doppi apici. Esempi di stringhe sono :

«pippo»	«pippo e pluto»	«2 persone»

- Un simbolo è una qualsiasi sequenza di uno o più caratteri ASCII stampabili delimitata da caratteri separatori. Per sapere quali sono considerati in Clips i caratteri separatori si rimanda alla Sez. 2.1.3 del manuale. Esempi di simboli sono :

pippo	Ciao	B76-HI	stampa_documento
127A	25-12-1995	@+=-%	2each

Si presti attenzione al fatto che :
1. una sequenza di caratteri che non rispondono esattamente al formato dei numeri è trattata come un simbolo ! Si veda la sintassi per i numeri sul manuale (Sez. 2.3.1) per maggiori informazioni;
2. la stringa «pippo» e il simbolo pippo non sono la stessa cosa visto che sono considerati di tipi diversi anche se sono composti dagli stessi caratteri.

- Un indirizzo esterno è l'indirizzo di memoria di una struttura dati ritornata da funzioni scritte in un linguaggio tipo C o ADA integrata con Clips. Questo tipo di dato può essere creato esclusivamente mediante la chiamata di queste funzioni, cioè non è possibile specificare un indirizzo esterno scrivendone il valore. E' possibile comunque visualizzarne il valore nel qual caso si ottiene :

  <Pointer-XXXXXX>
  dove XXXXXX è il valore del puntatore.

- Un fact-address è l'indirizzo di un fatto presente nella WM. Tale indirizzo è ritornato da funzioni quali la *assert*, la *retract* e la *modify* di cui parleremo più diffusamente in seguito. Per i fact-address

vale un discorso analogo a quello degli indirizzi esterni : sono ritornati da alcune funzioni e non si possono specificare scrivendone il valore ma si possono visualizzare ottenendo :

<Fact-XXX>
dove XXX è l'indice del fatto.

- L'instance-name é il nome simbolico che viene associato ad una istanza all'atto della sua creazione.
- L'instance-address é un indice interno che rappresenta l'indirizzo dell'istanza.

In Clips ci sono diversi modi per rappresentare la conoscenza : ci sono i fatti, le regole di produzione, le variabile globali, gli oggetti e le procedure. Si sceglie tra questi diversi metodi di rappresentazione a seconda dell'uso che si vuole fare della conoscenza a disposizione e a seconda che essa sia di tipo dichiarativo o procedurale.
CLIPS mette a disposizione uno strumento per organizzare la base di conoscenza - il costrutto *defmodule* (Sez. 10) : e' possibile cioè dividere regole, procedure , variabili e oggetti in diversi gruppi detti moduli. In ogni modulo sono visibili solo le regole, le procedure, le variabili globali e gli oggetti definiti al suo interno. Non è possibile dall'interno di un modulo accedere alle strutture di un altro modulo, a meno che non lo si indichi esplicitamente esportando e importando i costrutti: un modulo può esportare ogni costrutto ad esso visibile e non solo i costrutti che esso definisce!

Esempio:

```
(defmodule PIPPO
  (import PLUTO ?ALL)
  (import TOPOLINO deftemplate ?ALL)
  (import PAPERINO defglobal x y z)
  (export defgeneric +)
  (export defclass ?ALL))
```

Qui si definisce il modulo PIPPO che importa tutti i costrutti definiti dal modulo PLUTO; importa tutti i fatti non ordinati (*deftemplate*) del modulo TOPOLINO; importa le variabili grlobali x, y e z dal modulo PAPERINO; esporta la definizione della procedura generica di nome «+» e infine esporta tutte le classe che esso vede.

Un costrutto per essere importato da un modulo deve essere dichiarato esportabile dal modulo che lo definisce.
L'uso dei moduli risulta utile non solo per migliorare l'efficienza (per esempio in un modulo c'è un minor numero di regole da confrontare con i fatti per cui questa fase viene eseguita in minor tempo) ma anche per meglio seguire l'evoluzione dell'esecuzione del programma.

CLIPS per il suo funzionamento usa le seguenti strutture dati :

- La memoria di lavoro o working memory (WM) : vi sono registrati tutti i fatti e tutte le istanze di oggetti. In Clips la WM è anche indicata come lista dei fatti[1].
- L'insieme delle regole o Knowledge Base (KB).
- Lo stack dei moduli e il focus. E' possibile controllare mediante uno stack la sequenza di moduli

[1] In Clips la WM è gestita come un array dinamico. In ogni posizione della lista c'è l'indirizzo di un fatto. Se si aggiunge un fatto lo si inserisce al termine della lista; se lo si cancella si lascia semplicemente la posizione vuota.

usati da CLIPS. Il modulo usato da CLIPS in un determinato momento è dato dal focus. Si può spostare il focus o esplicitamente a programma mediante il costrutto *setfocus* o automaticamente : quando non ci sono più regole da considerare in un modulo, esso viene tolto dallo stack e CLIPS passa a confrontare i fatti con le regole del modulo che si trova in cima allo stack.

- L'agenda. Durante la fase di confronto tutte le regole le cui condizioni sono soddisfatte e che ancora non sono state eseguite sono aggiunte in una lista detta *agenda*. L'ordine in cui tali istanze di regole compaiono nella lista dipende dalla loro *salience* e tra regole con identica *salience* dalla strategia di risoluzione dei conflitti adottata: la risoluzione dei conflitti ordina secondo certi criteri le varie istanze di regole basandosi sulla loro importanza. Tale importanza é determinata dai seguenti fattori :

    1. La recentezza dell'attivazione dell'istanza di regola: una regola attivata in un ciclo successivo a quello in cui è stata attivata l'altra é considerata piú recente;
    2. la specificità della regola : una regola è più specifica di un'altra se nella sua parte sinistra appare un maggiore numero di condizioni rispetto all'altra ;
    3. la recentezza dei fatti: un fatto asserito dopo un altro é considerato piú recente;
    4. il caso : se ancora non si è stabilito un ordinamento si inseriscono le istanze di regole nell'agenda a caso (in verità si inseriscono le regole in base all'ordine in cui esse sono state scritte nel programma).

Vedremo in seguito tra quali strategie si può scegliere in Clips.

CLIPS esegue continuamente il seguente ciclo.

1. Confronta tutti i fatti con le regole del modulo in cima allo stack dei moduli. Se nessuna regola è attivabile, il focus passa al modulo successivo nello stack. Se non ci sono più moduli nello stack, l'esecuzione si ferma.
2. Le istanze delle regole attivate sono messe nell'agenda in ordine di *salience*. Se nell'agenda c'è più di una regola si attiva la conflict resolution secondo la strategia scelta e se ne sceglie una.
3. Vengono eseguite le azioni della regola scelta. Tra le azioni ci può tra l'altro essere un cambiamento esplicito del focus. Se la *salience* di alcune regole è calcolata dinamicamente, essa viene rivalutata.
4. Si ripete il ciclo dal passo 1.

Per eseguire un programma in Clips si devono eseguire 3 diversi passi :

1. Innanzitutto si deve caricare il programma in memoria : al comando di prompt del Clips si digita la parola chiave *load* seguito dal nome del file contenente il programma. Sullo schermo appariranno le definizioni di tutte le regole caricate nella KB e l'indicazione di tutti i fatti definiti ;

Esempio :
CLIPS> (load «c :/Clips/pippo.clp»)
Questo comando carica in memoria il programma pippo.clp

2. Si caricano tutti i fatti iniziali nella WM: al comando di prompt si digita *reset*. Oltre al caricamento dei fatti iniziali, in questa fase avviene anche un primo confronto tra le regole e tali fatti. Tutte le regole attivate vengono inserite nell'agenda. Attenzione : mediante il comando reset sono caricati solo quei fatti esplicitamente indicati dal programmatore nel programma (vedremo in seguito come li si indica all'interno del codice) ; altri fatti possono essere aggiunti dall'utente direttamente dal

comando di prompt mediante la parola chiave *deffacts* oppure mediante il comando *assert*. [2]

3. Si esegue il programma : si digita *run* e l'esecuzione inizia.

Ogni volta che si dà il comando di *reset* si ristabiliscono le condizioni iniziali. Perciò ogni volta che si vuole ricominciare la computazione daccapo, si esegue questo comando.
Per scaricare un programma dalla memoria e cancellare il contenuto della KB e della WM si da il comando di *clear*. Dopo questo comando si è pronti a caricare ed eseguire un altro programma.
L'ambiente Clips è leggermente diverso dagli ambienti tradizionali per un linguaggio imperativo. In tali ambienti, se si interrompe l'esecuzione, si deve normalmente eseguire di nuovo il programma; la stessa cosa se si vogliono cambiare i valori di alcune variabili. Eseguire un programma in Clips non significa necessariamente far partire l'esecuzione e ottenere risultati senza far nulla durante la computazione! Per esempio, in ogni momento si può fermare l'esecuzione e aggiungere o togliere regole, fatti o oggetti, oppure modificare i comportamenti che il sistema deve avere come la strategia di risoluzione dei conflitti e poi riprendere la computazione senza dover rifare il lavoro svolto fino al momento dell'interruzione; non è strettamente necessario caricare un programma: si possono caricare i fatti, oppure definire regole e oggetti direttamente dalla linea di comando. Insomma l'ambiente Clips si presenta come il contenuto della KB, della WM e dalle variabili di sistema che possono essere variati in qualsiasi istante sia per effetto di modifiche fatte direttamente dall'utente tramite la linea di comando, sia per effetto dell'esecuzione delle regole. Si noti perciò che:

- eseguire in Clips significa far partire la fase di confronto tra le regole della KB e il contenuto della WM;
- l'ambiente si riporta alla fase iniziale solo con il comando di *reset*: se era stato caricato un programma, l'ambiente iniziale è composto da tutti i fatti, gli oggetti e le regole definite nel programma stesso, altrimenti è semplicemente vuoto.

Vediamo ora in maggiore dettaglio alcuni costrutti del Clips.

Fatti

Come si è detto i fatti possono essere sia i dati iniziali su cui il sistema lavora, sia le conclusioni cui a mano a mano esso giunge durante la computazione. Essi costituiscono insomma le informazioni su cui lavorano le regole di produzione contenute nella KB.
Esistono in CLIPS due tipi di fatti : ordinati e non ordinati. I primi sono semplicemente una lista composta da un simbolo seguito da zero o più simboli: si possono vedere come relazioni tra i vari simboli o come i predicati in logica; il riferimento a ciascun elemento è puramente posizionale.

Esempio:

```
(assert (zio_di  Paperone Paperino))
```

Questo comando inserisce un fatto ordinato nella WM.

I secondi sono invece strutture molto simili alla struct del C dove ogni campo ha un nome e uno o più valori. Tali valori possono avere vincoli di tipo, valore e cardinalità espressi da un cosiddetto «attributo» associato al campo. I fatti non ordinati sono definiti nel programma mediante il costrutto

[2] Ci si può chiedere a questo punto qual è la differenza tra inserire un fatto usando il comando *deffact* o usando il comando *assert*. Si veda il paragrafo sui fatti per ulteriori informazioni al riguardo.

'deftemplate'. Il vantaggio nell'uso di fatti non ordinati è evidente quando si deve fare riferimento a singoli campi (slot) del fatto stesso[3]. Esistono due tipi di slot: uno slot semplice che può assumere un solo valore e i cosiddetti *multislot* che possono invece assumere una lista di valori

Commentato [TD1]:

Esempio:

```
(deftemplate  animal
                        (slot race)
                        (slot age  (type INTEGER))
                        (slot energy (default 40))
                        (multislot can_be_found_in
                                                (allowed-values Africa SouthAmerica Asia)))
```

Questo esempio definisce il fatto non ordinato «animal» che ha 4 campi (slot) : un campo «race» che è un simbolo, un campo «age» che è un intero; un campo «energy» che è un intero con default e il campo «can_be_found_in» che può assumere uno o più dei valori indicati.

In uno slot è possibile definire un valore di default. Esso è un valore che viene attribuito automaticamente al campo durante la creazione del fatto se non si specifica un valore diverso. Tale valore può naturalmente essere modificato durante la computazione. Il default è solo uno dei possibili attributi che può avere uno slot. Un attributo è una caratteristica che caratterizza uno slot. Nella Tabella 1 viene data una panoramica dei possibili attributi di uno slot (Sez. 3.3 e 11).

Tabella 1

Attributi	Valore	Commento
Default	Espressione	Il valore dello slot se non è indicato assume il valore dell'espressione indicata nell'attributo.

[3] Avvertenza! Nel manuale (Sez. 3) c'è il rischio di fare confusione traducendo dall'inglese all'italiano. L'autore indica con «field» le singole parole usate in un fatto, cioè i singoli simboli che appaiono nella definizione di un fatto (non si traduca perciò tale parola con «campo»!) e con «slot» ciò che noi intendiamo per campo di un fatto non ordinato.
Esempio:

(deftemplate città (slot nome) (slot #_di_abitanti))

Nel manuale «città» è indicato come il primo *field*; per noi è il simbolo che contraddistingue il fatto non ordinato di nome «città». «nome» è il primo *slot* del fatto; per noi invece è il primo campo del fatto.

Esempio:

(padre_di Marco Luca)

Questo fatto è composto da tre 'fields', cioè da tre simboli.

Qui e nel seguito adotteremo la seguente convenzione:

- 'symbol' indica il tipo di dato simbolo;
- 'campo' indica uno dei simboli di un fatto ordinato;
- 'slot' indica uno dei campi di un fatto non ordinato o di un'istanza di un oggetto;
- 'simbolo' indica qualsiasi stringa di caratteri.

	?none	Il valore dello slot deve essere specificato al momento dell'inserimento in WM. Se non lo si fa Clips dà un messaggio di errore e blocca la computazione.
	?derive	Il valore di default dello slot è derivato in base ad una complessa base di regole. Per maggiori informazioni si consulti la Sez. 11.5 del manuale.
Default-dynamic	Espressione	Il valore dello slot se non è indicato assume il valore a run time dell'espressione indicata.
Allowed-values[4]	Lista	Il valore dello slot può assumere solo uno o più valori tra quelli indicati nella lista.
Range	(from to) I due valori from e to possono essere espressi sia con numeri interi che con numeri reali. Si può anche specificare la parola chiave ?VARIABLE: al posto di from sta per -∞; al posto di to sta per +∞	Il valore dello slot deve essere compreso nel range indicato.
Cardinality	(min max) I due valori min e max devono essere espressi con numeri interi. Si può anche specificare la parola chiave ?VARIABLE: al posto di min sta per 0 (zero); al posto di max sta per +∞	Il numero dei valori assunti dallo slot deve essere almeno pari al numero minimo indicato e non maggiore del numero massimo.

[4] Se si usa *allowed-values* il tipo dei valori indicato è generico. Volendo si può specificare il tipo dei valori usando le parole chiave corrispondenti: *allowed-symbols, allowed-strings, allowed-lexemes, allowed-integers , allowed-floats, allowed-numbers, allowed-instance-names, allowed-values.*
Es.:

(slot stringhe)
(allowed-strings ("Pippo", "Pluto", "Minnie")

Indica uno slot di nome 'stringhe' che accetta come valori eclusivamente le stringhe "Pippo", "Pluto" oppure "Minnie".

Type	I tipi ammessi sono: SYMBOL, STRING, LEXEME, INTEGER, FLOAT, NUMBER, INSTANCE-NAME, INSTANCE-ADDRESS, INSTANCE, EXTERNAL-ADDRESS, FACT-ADDRESS Usare NUMBER è equivalente ad usare entrambi INTEGER e FLOAT; usare LEXEME è equivalente ad usare entrambi SYMBOL e STRING; usare INSTANCE è equivalente ad usare entrambi INSTANCE-NAME e INSTANCE-ADDRESS. Se si usa la parola chiave ?VARIABLE si indica che lo slot può assumere qualsiasi valore.	Il tipo del valore assunto dallo slot deve essere dello stesso tipo indicato.

Esempio 1:

```
(deftemplate obj
  (slot name
    (type SYMBOL)
    (default ?DERIVE))
  (slot location
    (type SYMBOL)
    (default ?DERIVE))
  (slot on-top-of
    (type SYMBOL)
    (default floor))
  (slot weight
    (allowed-values light heavy)
    (default light))
  (multislot contents
    (type SYMBOL)
    (default ?DERIVE)))
```

Il fatto non ordinato «obj» ha i seguenti campi:

* «name» e «location» di tipo symbol e il cui valore di default è derivato;
* «on-top-of» di tipo symbol che ha come defautl il valore 'floor';
* «weight» che può assumere solo i valori 'light' e 'heavy' e che ha come default il valore 'light';
* «contents» che può assumere una lista di valori di tipo symbol e il cui default è derivato.

Esempio 2:

```
CLIPS> (clear)
CLIPS>
(deftemplate foo
  (slot w (default ?NONE))
  (slot x (default ?DERIVE))
  (slot y (default (gensym*)))
  (slot z (default-dynamic (gensym*))))
CLIPS> (assert (foo))

[TMPLTRHS1] Slot w requires a value because of its (default ?NONE) attribute.
CLIPS> (assert (foo (w 3)))
<Fact-0>
```

Il fatto non ordinato foo ha uno slot «w» a cui - a causa della presenza della parola chiave ?NONE - deve essere esplicitamente dato un valore al momento della dichiarazione. Si noti infatti che viene generato un

errore quando si fa la prima assert. Agli slot «y» e «z» è invece assegnato il valore di una funzione (in questo caso è una funzione che genera e restituisce un nuovo simbolo ogni volta che è chiamata)5.

I fatti non ordinati (deftemplate facts) si distinguono da quelli ordinati dalla prima stringa di caratteri del fatto stesso: essa - in tutti i fatti - deve necessariamente essere una stringa di tipo symbol e se questo simbolo corrisponde al nome di un *deftemplate*, allora il fatto è considerato non ordinato. In ogni caso il primo simbolo di un fatto può essere visto come il nome del fatto.

Si può agire sui fatti mediante diversi costrutti[6].
La parola chiave *deffacts* permette di specificare un insieme di conoscenze «iniziali»: ogni volta che si effettua un comando di *reset* in Clips, tutti i fatti indicati in un costrutto *deffacts* vengono automaticamente caricati nella lista dei fatti ed essi costituiscono perciò la conoscenza a priori del sistema.
Durante l'esecuzione la lista dei fatti può essere modificata con diversi comandi : *assert* inserisce un nuovo fatto; *retract* cancella un fatto dalla lista; *modify* modifica un fatto della lista. Esiste anche il comando *duplicate* che duplica un fatto della lista. Essi costituiscono le azioni che come prima detto si possono eseguire quando si applica una regola. Si badi bene di comprendere la differenza tra asserire un fatto con la *deffacts* e asserire un fatto con la *assert*: un fatto asserito con la *deffacts* viene inserito nella WM ogni volta che si dà un comando di *reset* al sistema e solo in questo caso. Un fatto indicato nella *assert*, viene invece inserito nella WM ogni volta che il flusso del programma porta ad eseguire la *assert* stessa.

Esempio :

```
(1) CLIPS> (deffacts  my_name (handempty))
(2) CLIPS> (deffacts  our_name (clear) (empty))
(3) CLIPS> (reset)
(4) CLIPS> (assert (off))
```

Il comando (1) dichiara che il fatto «handempty» è uno dei fatti noti a priori dal sistema. Si noti come si debba necessariamente assegnare un nome (my_name) al comando *deffacts*. Il comando (2) dichiara i fatti «clear» e «empty» come noti a priori. Si noti che una volta eseguiti questi comandi la lista dei fatti non cambia: la deffacts è solo una dichiarazione! Se si esegue un comando di *reset* del sistema, si vedrà che i fatti sopra menzionati appariranno nella WM. Il comando (4) inserisce il fatto «off» nella lista dei fatti. Questa volta, il fatto «off» apparirà immediatamente nella lista dei fatti.
Si provi ad eseguire l'esempio e dare un ulteriore comando di *reset* dopo il comando (4): il fatto «off» sparisce mentre i fatti dichiarati nelle *deffacts* vengono automaticamente di nuovo caricati nella WM.

Si faccia attenzione ad una particolarità del Clips: se si asserisce un fatto che esiste già nella WM, il

[5] Differenza tra default e default-dynamic: si noti come entrambi accettino una espressione che restituiscono un valore. Con default si valuta l'espressione al momento della definizione e successivamente si userà sempre quel valore; con default-dynamic invece l'espressione è valutata ad ogni assert.

[6] Per evitare confusione in seguito è bene chiarire la terminologia seguita in CLIPS riguardo i fatti. Si può fare riferimento ad un fatto in più modi :

1. Con un fact-index : esso è un intero e rappresenta la sua posizione nella lista dei fatti; l'indice dei fatti parte da zero e viene incrementato di uno ogni volta che viene inserito un nuovo fatto. Ad ogni *reset* esse viene riportato a zero.
2. Con un fact-address : è il valore ritornato dalle funzioni indicate nel seguito;
3. Con un fact-identifier : è l'identificatore usato nell'interfaccia del Clips che mostra il contenuto della lista dei fatti. Esso è sempre del tipo f-xxx dove xxx è l'indice del fatto.

fatto non viene inserito. Questo comportamento può venire cambiato mediante un apposito comando che permette appunto la duplicazione dei fatti (Sez. 13.4.4 e 13.4.5). In una *assert* si possono specificare non solo costanti ma anche espressioni; il primo campo del fatto comunque deve essere un symbol. Se la *assert* va a buon fine, restituisce l'indirizzo dell'ultimo fatto appena inserito[7], altrimenti restituise FALSE.

[7] Con un unico comando di assert si possono inserire più fatti nella WM.

Esempio:
(1) CLIPS> (clear)
(2) CLIPS> (assert (color red))
<Fact-0>
(3) CLIPS> (assert (color blue) (value (+ 3 4)))
<Fact-2>
(4) CLIPS> (assert (color red))
FALSE
(5) CLIPS> (deftemplate status (slot temp) (slot pressure))
(6) CLIPS> (assert (status (temp high) (pressure low)))
<Fact-3>
Con il comando (3) si usa la stessa assert per inserire due fatti diversi. La assert restituisce l'indirizzo dell'ultimo fatto inserito. Con il comando (4) si cerca di inserire un fatto già esistente. La assert non lo inserisce e restituisce FALSE.
Con la *assert* si possono ovviamente inserire sia fatti ordinati che non ordinati.

Clips offre una serie di funzioni che permettono di ottenere informazioni riguardo ai fatti. Si veda la Tabella 2 per una lista di tali funzioni.

Tabella 2

Funzione	Commento	Esempio
Funzioni per i deftemplate (Sez. 12.8)		
get-deftemplate-list	Restituisce una lista contenenete i nomi di tutti i deftemplate definiti fino a quel momento.	CLIPS> (clear) CLIPS> (get-deftemplate-list) () CLIPS> (deftemplate foo) CLIPS> (deftemplate bar) CLIPS> (get-deftemplate-list) (foo bar) CLIPS>
Deftemplate-module	Questa funzione restituisce il modulo in cui il deftemplate specificato è stato definito.	CLIPS> (clear) CLIPS> (deftemplate foo) CLIPS>(deftemplate-module foo) MAIN
Funzioni per i fatti (Sez. 12.9)		
Fact-index	Ritorna l'indice nella lista dei fatti del fatto dato. Tale fatto è indicato mediante il suo indirizzo.	Sintassi: fact-index <fact-address> Si veda il paragrafo sulle regole per come specificare l'indirizzo di un fatto.
Assert	Inserisce un nuovo fatto nella lista dei fatti. Restituisce l'indice dell'ultimo fatto inserito.	
Retract	Elimina il fatto indicato mediante	Si veda il paragrafo sulle regole

	il suo indirizzo dalla lista dei fatti.	per come specificare l'indirizzo di un fatto.
Modify	Modifica il fatto indicato mediante il suo indirizzo.	Si veda il paragrafo sulle regole per come specificare l'indirizzo di un fatto[8].
Duplicate	Duplica il fatto indicato mediante il suo indirizzo.	Si veda il paragrafo sulle regole per come specificare l'indirizzo di un fatto.
Funzioni per i deffacts (Sez. 12.10)		
get-deffacts-list	La funzione restituisce una lista contenente i nomi di tutti i costrutti deffacts definiti fino a quel momento.	CLIPS> (clear) CLIPS> (getdeffacts-list) (initial-fact) CLIPS> (deffacts foo) CLIPS> (get-deffacts-list) (initial-fact foo) CLIPS>
deffacts-module	Questa funzione restituisce il nome del modulo in cui il deffacts specificato è definito.	CLIPS> (clear) CLIPS> (deffacts foo) CLIPS>(deffacts-module foo) MAIN

In ogni momento è possibile dire al sistema di visualizzare la lista dei fatti mediante il comando *facts*.

Variabili globali

Come si è già accennato, Clips integra al suo interno anche un linguaggio di tipo procedurale. Com'è naturale in tali linguaggi, anche in Clips è possibile definire variabili globali usando il costrutto *defglobal*. All'interno di un programma una variabile globale è indicata con la sintassi ?*<*symbol*>*.

Esempio:

```
(defglobal
  ?*x* = 3
  ?*y* = ?*x*
  ?*z* = (+ ?*x* ?*y*))
```

E' definita la variabile «x» con valore iniziale '3'; si definisce anche la variabile «y» a cui è assegnato lo stesso valore iniziale di «x»; infine viene definita la variabile «z» a cui è associato come valore iniziale la somma delle variabili «x» e «y».

Per legare un particolare valore ad una variabile globale si usa la funzione *bind*[9].

[8] La *modify* non puó essere usata per modificare un fatto ordinato. Per modificare un fatto ordinato si deve prima fare una *retract* del fatto che si vuole modificare e successivamente una *assert* del nuovo fatto.

[9] Attenzione! Con riferimento all'esempio precedente: se si definisce in quel modo il valore iniziale di «y», ogni volta che le si applica la funzione di *bind*, essa assumerà il valore che la variabile «x» ha in quel momento.
Esempio:

```
CLIPS> (bind ?*x* 10)          il valore di x è 10 e il valore di y è 3;
CLIPS> (bind ?*y* 20)          il valore di y è 20;
```

CLIPS> (bind ?*y*) il valore di y è ora 10, cioè il valore di x.

Esempio:

```
(bind ?*count* 0)
(bind ?*count* (+ ?*count* 1))
```

Il primo comando inizializza la variabile globale «count» a 0; successivamente la si incrementa di 1.

Come si vede dall'esempio, in ogni *defglobal* possono essere definite una o più variabili globali. E' possibile indicare nel costrutto anche il modulo a cui le variabili globali appartengono: se non si specifica alcun modulo, la variabile è automaticamente definita all'interno del modulo corrente.

Esempio:

```
CLIPS> (clear)
CLIPS> (defglobal   ?*x* = 0)
CLIPS> (defmodule   A)
CLIPS> (defglobal  ?*y* = 0)
CLIPS> (defglobal  MAIN  ?*z* = 0)
```

La variabile «x» è definita automaticamente all'interno del modulo MAIN che è quello corrente dopo un comando di clear. Cosa analoga per la variabile «y» che è definita nel modulo 'A' appena definito. La variabile «z» è invece espressamente definita in MAIN e perciò appartiene a tale modulo.

Al pari di altre variabili esse possono partecipare al confronto con le regole della KB, con l'unica differenza che cambiare il valore di una di esse non fa scattare automaticamente il processo di confronto: normalmente se si modifica una fatto od un oggetto nella WM, Clips effettua automaticamente un nuovo confronto tra le regole e il contenuto della WM; se si modifica invece una variabile globale (per esempio, si effettua una *bind* con un nuovo valore come azione di una regola) non scatta un nuovo confronto e si prosegue con l'esecuzione della regola successiva nell'agenda.

Ogni volta che viene impartito un ordine di *reset*, le variabili globali assumono nuovamente il valore iniziale, così come quando viene eseguito un comando *bind* senza farlo seguire da un valore. Le variabili globali possono essere eliminate in due modi: o dando un comando di *clear*, oppure eseguendo il comando *undefglobal*.

Esempio:

```
CLIPS> (defglobal A ?*x*= 10)
CLIPS> (undefglobal A x)
```

In questo esempio si definisce una variabile globale «x» nel modulo A; successivamente la si elimina. Si noti che nella undefglobal si indica la variabile semplicemente con il suo nome «x» senza asterischi.

Clips fornisce le funzioni della Tabella 3 per manipolare le variabili globali.

Tabella 3

Funzione	Commento	Esempio
Funzioni delle variabili globali (Sez. 12.13)		
get-defglobal-list	La funzione restituisce una lista contenente i nomi di tutte la variabili con i loro relativi valori definite fino a quel momento.	CLIPS> (clear) CLIPS> (get-defglobal-list) () CLIPS> (defglobal ?*x* = 3 ?*y* = 5) CLIPS> (get-defglobal-list) (x y)
Defglobal-module	Questa funzione restituisce il nome del modulo in cui la variabile globale indicata è definita. Se due o più variabili aventi lo stesso nome sono definite in moduli diversi, la funzione restituisce il nome dell'ultimo modulo in cui la variabile indicata è stata definita.	CLIPS> (defmodule A) CLIPS> (defmodule B) CLIPS> (defglobal B ?*x* = 1) CLIPS> (defglobal A ?*x* = 0) CLIPS>(defglobal-module x) (A)

Regole

Una regola è un insieme di condizioni e di azioni che possono essere eseguite se le condizioni sono soddisfatte. Le regole sono eseguite in base all'esistenza o alla non esistenza di certi fatti specificati nelle condizioni. CLIPS fornisce il motore per confrontare le condizioni specificate in ciascuna regola con i fatti presenti in memoria. Ogni regola in un programma CLIPS è definita mediante il costrutto '*defrule*' (Sez. 5).

In questo costrutto oltre al nome della regola si indicano anche un commento opzionale, un insieme di proprietà della regola, l'insieme dei condition elements (CE) cioè l'insieme delle condizioni necessarie per attivare la regola e l'insieme delle azioni (LHS) da eseguire nel caso la regola venga applicata. Si noti che tutti gli insiemi sopra menzionati possono anche essere vuoti: se una regola non ha CE allora essa è automaticamente attivata; se una regola non ha azioni nella sua LHS ed è eseguita, non accadrà nulla.

Tutti le condizioni indicate nella regola sono legate tra di loro da un *and* logico implicito. Quando una regola è applicata le sue azioni sono eseguite in modo sequenziale.

Esempio :

```
(defrule regola-esempio   "Esempio di regola"
   (frigorifero luce  accesa)
   (frigorifero sportello aperto)
   =>
   (assert (frigorifero cibo avariato)))
```

Qui si è definita una regola di nome «regola-esempio» e con un commento. Essa ha due CE ('frigorifero luce accesa' e 'frigorifero sportello aperto') e una azione (asserire il fatto 'frigorifero cibo avariato').

Nel paradigma a regole di produzione si parla di istanza di regola che é una regola in cui tutti i CE sono soddisfatti e dove tutte le variabili sono state associate ad un particolare valore. Nella lista dei fatti si possono trovare piú combinazioni di fatti che soddisfano la stessa regola: Clips attiva nell'agenda tante istanze di quella regola quante sono le combinazioni di fatti che la soddisfano.

Analizziamo ora la struttura della parte destra (RHS) di una regola Clips.
Essa è composta da una serie di condizioni (CE) che devono essere tutte soddisfatte affinché la regola venga attivata. Esistono 8 diversi tipi di condizioni in Clips: pattern, test, and, or, not, exists, forall e logical.

1. La condizione 'pattern'

Questa condizione è espressa mediante un insieme di vincoli che fatti o istanze di oggetti devono soddisfare. Tali vincoli sono espressi mediante nomi di fatti o istanze di oggetti, variabili e caratteri jolly. Una condizione 'pattern' è soddisfatta se e solo se ciascun vincolo che la compone è soddisfatto.
I vincoli espressi mediante nomi di slot e campi sono usati per testare slot di fatti non ordinati, campi appartenenti a fatti ordinati o ancora elementi di una istanza di oggetto; i caratteri jolly servono a indicare che un campo o uno slot possono essere associati a qualsiasi valore; le variabili sono usate per memorizzare valori di campi o slot così da poter essere usate nel seguito in altre condizioni o nella RHS della regola come argomenti di un'azione.
Il primo simbolo della condizione 'pattern' deve essere obbligatoriamente il nome di un fatto o di un oggetto e deve perciò essere un symbol: esso serve al Clips per stabilire a che cosa la condizione è applicata: ad un fatto ordinato, ad un fatto non ordinato o ancora ad un'istanza di oggetto.

Questo è l'insieme di fatti su cui si baseranno gli esempi nel seguito. Si possono caricare in memoria con un comando di *reset*.

```
(deffacts data-facts                Definisce dei fatti ordinati come fatti iniziali
  (data 1.0 blue "red")
  (data 1 blue)
  (data 1 blue red)
  (data 1 blue RED)
  (data 1 blue red 6.9)
)

(deftemplate person                 Definisce un fatto non ordinato 'person'
  (slot name)
  (slot age)
  (multislot friends)
)

(deffacts people                    Definisce alcuni fatti non ordinati come fatti iniziali.
  (person (name Joe) (age 20))
  (person (name Bob) (age 20))
  (person (name Joe) (age 34))
  (person (name Sue) (age 34))
  (person (name Sue) (age 20))
)
```

Come valore di confronto si possono usare letterali, cioè costanti di diverso tipo, caratteri jolly e variabili.

Esempio 1 :

(defrule find-data (data 1 blue red) => . . .)

Come valori per i vincoli qui si sono usati dei letterali.
Cosa fa Clips nella fase di confronto: vede che c'è un CE e inizia la sua analisi; prende la prima stringa; essa deve essere un symbol e va a cercare nella WM se esiste un fatto o un'istanza che ha questa stringa come primo simbolo. Nel nostro caso la trova e Clips scopre così che si tratta di un fatto ordinato. Prosegue nell'analisi del CE e vede che ci sono dei valori letterali. A questo punto vede se il fatto ordinato contiene esattamente gli stessi valori e in tal caso attiverà la regola 'find-data'.
Perciò, la regola 'find-data' verrà attivata se e solo se:

- esiste un fatto ordinato il cui nome è 'data';
- il secondo campo ha valore '1';
- il terzo campo ha valore 'blue';
- il quarto campo ha valore 'red'.

Esempio 2 :
(defrule Find-Bob
(person (name Bob) (age 20)) => ...)
Anche qui si usano come valori dei letterali. Vale lo stesso discorso fatto per l'esempio precedente; l'unica differenza con l'esempio precedente e che qui si fa il confronto con un fatto non ordinato.

Anziché un valore è possibile esprimere un carattere jolly. Il Clips ne ha due: il carattere '?' che sta per qualsiasi valore singolo e il carattere «$?» che sta invece per una lista di valori.

Esempio 3 :

defrule find-data
(data ? blue red $?)
=>)

Questa regola verrà attivata se esiste almeno un fatto ordinato che ha come valore in terza e quarta posizione rispettivamente le costanti 'blue' e 'red'. Nella seconda posizione ci può essere qualunque valore singolo di qualunque tipo; in quinta posizione ci può essere una lista (cioè zero, uno o più elementi) di qualsiasi valore e tipo.
Vale qui la pena di vedere quali fatti tra quelli d'esempio hanno successo nel confronto:

f-1 (data 1.0 blue "red")
Non va bene per attivare la regola perché red non è di tipo symbol come specificato nell'esempio, bensì di tipo stringa;
f-2 (data 1 blue)
Non va bene perché non ha il simbolo 'red';
f-3 (data 1 blue red)
Va bene. In questo caso la lista che sta al posto di $? è vuota;
f-4 (data 1 blue RED)
Non va bene: il simbolo 'RED' è diverso dal simbolo 'red';

f-5 (data 1 blue red 6.9)
Va bene.

Esempio 4 :

```
(defrule match-all-persons
  (person)
  =>)
```

Affinchè questa regola venga attivata basta che esista almeno un fatto non ordinato di nome 'persona'.

E' possibile utilizzare anche una variabile: al posto di un letterale o di un carattere jolly si inserisce il nome di una variabile. Essa non ha inizialmente né tipo, né valore; durante la fase di confronto le verrà attribuito un valore e un tipo e lo manterrà fintanto che l'istanza della regola è attiva. E' possibile ripetere la variabile in altri CE della regola, si noti però che essa avrà sempre lo stesso valore a cui è stata legata inizialmente.

Esempio 5 :

```
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
```

Questa regola è attivata da ogni fatto di nome 'data' che abbia esattamente 3 valori. Tali valori sono memorizzati nelle variabili x, y e z e sono usati nella LHS per la stampa.

Esempio 6 :

```
CLIPS> (reset)
CLIPS> (assert (data 1 blue)
           (data 1 blue red)
           (data 1 blue red 6.9))
<Fact-3>
CLIPS>
```

```
(defrule find-data-1
  (data ?x $?y ?z)
  =>
  (printout t "?x = " ?x crlf
            "?y = " ?y crlf
            "?z = " ?z crlf
            "------" crlf))
CLIPS> (run)
?x = 1
?y = (blue red)
?z = 6.9
------
?x = 1
?y = (blue)
?z = red
------
?x = 1
?y = ()
?z = blue
------
```

Questa regola è attivata da ogni fatto che presenta un valore (memorizzato in x), seguito da una lista di valori(memorizzata in y), seguita da un altro valore(memorizzato in z). Si noti che esistono diversi modi di istanziare queste variabili. Clips li prende in considerazione tutti e infatti la regola è eseguita 3 volte, tanti quanti sono i modi di istanziare queste variabili sui fatti presenti in WM.

Esempio 7 :

```
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> (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.
```

Si noti come la variabile nelle LHS delle due regole occorra due volte. Esaminiamo la prima regola: nel confronto con i fatti, la variabile singola x è associato al valore 'green' del primo fatto. Successivamente Clips prende in considerazione il secondo CE e cerca un fatto di nome 'data' che abbia come primo simbolo 'purple' e come secondo simbolo il valore di x: trova così che il terzo fatto soddisfa questa condizione e attiva la regola. Analogo discorso per la seconda regola definita, solo che qui si ha una variabile che può assumere una lista di valori. Le possibili istanziazioni sono due, perciò sono attivate due istanze di regola.

I letterali, le variabili e i caratteri jolly che compaiono in una condizione di tipo 'pattern' possono anche essere uniti dai connettivi logici & (and), | (or) e ~ (not).
Una condizione & è soddisfatta se sono soddisfatte tutte le sottocondizioni della formula; la condizione | è soddisfatta se è soddisfatta almeno una delle sottocondizioni e infine la condizione ~ è soddisfatta se non è soddisfatta la sottocondizione.

Esempio 8 :

```
CLIPS> (clear)
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     example1-2: f-4
```

```
0     example1-3: f-3
0     example1-1: f-1
For a total of 3 activations.
CLIPS>
```

La condizione della regola ‘example1-1’ è soddisfatta dal fatto ordinato f-1: infatti si richiede che esista un fatto ordinato di nome ‘data-A’ che non abbia come primo simbolo il valore ‘blue’. La condizione della regola ‘example1-2’ è soddisfatto se esiste un fatto non ordinato di nome ‘data-B’ il cui slot ‘value’ abbia un valore che non è né ‘red’ né ‘green’. Infine la condizione dell’ultima regola è soddisfatta se esiste un fatto di nome ‘data-B’ che abbia nel suo slot ‘value’ un valore ‘green’ o ‘red’.

Esempio 9 :

```
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
CLIPS>
```

In questo esempio appaiono anche delle variabili. La regola ‘example2-1’ è attivata nel caso esista un fatto di nome ‘data-B’ che ha un valore e inoltre esso non è né ‘red’ e né ‘green’. In pratica qui la variabile è utile per memorizzare il valore dello slot per poi stamparlo. Commenti analoghi si possono fare per la condizione della seconda regola.

Esempio 10 :

```
CLIPS> (clear)
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 example3-1
  (data-A ?x&~green)
  (data-B (value ?y&~?x))
  =>)
CLIPS>
(defrule example3-2
  (data-A ?x)
  (data-B (value ?x&green|blue))
  =>)
CLIPS>
(defrule example3-3
  (data-A ?x)
  (data-B (value ?y&blue|?x))
  =>)
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      example3-3: f-1,f-4
0      example3-3: f-2,f-4
0      example3-2: f-2,f-4
0      example3-1: f-2,f-3
For a total of 4 activations.
```

In questo esempio nelle condizioni delle regole definite, le stesse variabili compaiono in più CE diversi. Valgono naturalmente gli stessi commenti fatti nell'esempio 7.

Leggendo l'esempio 10 potrebbe essere sorto il dubbio su come si devono leggere le formule logiche che compaiono nei CE: ?x&green|blue viene inteso come (?x&green)|blue oppure come ?x&(green|blue) ? Clips valuta questa espressione come nell'ultimo caso: prima si lega la variabile ad un valore e poi si controlla che tale valore rispetti le condizioni che seguono. Seguendo questa indicazione si comprende anche il significato di ?y&blue|?x che si intende come (?y&blue)|(?x) cioè si lega 'y' ad un valore e si controlla che esso sia 'blue' oppure si controlla se nel fatto è presente un valore che è identico a quello contenuto nella variabile 'x'.

In una regola talvolta è utile vedere se su un valore è verificato un predicato, cioè una funzione che ritorna un valore booleano. E' possibile perciò inserire una chiamata a una funzione booleana in un CE.

Esempio 11 :

```
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>
```

In questo caso la condizione della regola è soddisfatta se esiste un fatto di nome 'data' avente come primo campo un numero. Nel CE il valore di tale campo è associato ad una variabile. Viene poi chiamata la funzione 'numberp' con tale variabile come parametro che verifica se il valore passato è un numero. Se la funzione restituisce TRUE, allora la condizione è soddisfatta. Nell'esempio solo i primi due fatti dei tre inseriti soddisfano la condizione: infatti nell'agenda come si vede compaiono due istanziazioni della regola.

Esempio 12 :

```
CLIPS>
(defrule example-2
  (data ?x&~:(symbolp ?x))
  =>)
CLIPS> (assert (data 1) (data 2) (data red))
<Fact-2>
CLIPS> (agenda)
0     example-2: f-1
0     example-2: f-0
For a total of 2 activations.
CLIPS>
```

Questo esempio richiede nella condizione che il campo del fatto 'data' non sia un simbolo: richiama la funzione 'symbolp' che restituisce TRUE se il valore passatole è di tipo symbol. Il risultato ovviamente è lo stesso dell'esempio precedente.

Esempio 13 :

```
CLIPS> (clear)
CLIPS>
(defrule example-3
  (data ?x&:(numberp ?x)&:(oddp ?x))
  =>)
CLIPS> (assert (data 1) (data 2) (data red))
<Fact-2>
CLIPS> (agenda)
0     example-3: f-0
For a total of 1 activation.
```

```
CLIPS>
```

Nella condizione si richiede che il fatto 'data' contenga nel suo primo campo un numero dispari.

Esempio 14 :

```
CLIPS> (clear)
CLIPS>
(defrule example-4
  (data ?y)
  (data ?x&:(> ?x ?y))
  =>)
CLIPS> (assert (data 3)
               (data 5)
               (data 9))
<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>
```

Qui per attivare la regola si devono trovare due valori, uno maggiore dell'altro. Dati i fatti dell'esempio, si hanno 3 attivazioni della regola.

Esempio 15 :

```
CLIPS> (clear)
CLIPS>
(defrule example-5
  (data $?x&:(> (length$ ?x) 2))
  =>)
CLIPS> (assert (data 1
               (data 1 2)
               (data 1 2 3))
<Fact-2>
CLIPS> (agenda)
0     example-5: f-2
For a total of 1 activation.
CLIPS>
```

La regola è attivata se si trova un fatto di nome 'data' avente un numero di campi maggiore di 2. Nella variabile multipla x, infatti, viene inserita la lista di valori che seguono la stringa «data» del fatto e viene poi valutata la lunghezza di tale lista.

In Clips è anche possibile effettuare una chiamata ad una funzione e usare il valore ritornato come se fosse un letterale per il confronto tra le regole e i fatti. Per fare questo si usa il simbolo '=' seguito dalla funzione desiderata

Esempio 16 :

```
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))
            (data (x 3) (y 9)))
<Fact-1>
CLIPS> (agenda)
0      twice: f-0
For a total of 1 activation.
CLIPS>
```

Qui la regola è attivata da un fatto di nome 'data' il cui slot y presenta un valore doppio di quello dello slot x. Nel CE infatti si richiede che il valore dello slot y sia uguale al valore della funzione moltiplicazione chiamata con due parametri: 2 e il valore della variabile x che era stata legata in precedenza al valore contenuto nello slot x.

Molto spesso è utile sapere quale fatto o oggetto ha soddisfatto il CE. Clips permette di associare l'indirizzo di tale fatto o oggetto ad una variabile mediante il carattere '<-' . In questo modo tale variabile può essere usata come parametro per operazione successive sul fatto o sull'oggetto che ha soddisfatto le condizione, come per esempio, modifiche o eliminazioni. Non è possibile legare una variabile ad un CE con il 'not' davanti[10].

Esempio 17 :

```
(defrule pippo
   (data 1)
   ?f <- (topolino pluto)
   =>
   (retract ?f))
```

Questa regola viene attivata se esistono in WM i fatti ordinati '(data 1)' e '(topolino pluto)'. L'indirizzo di quest'ultimo fatto è inserito nella variabile 'f' e successivamente usato nella parte sinistra della regola per eliminare il fatto dalla WM.

```
(defrule compare-facts-1
   ?f1 <- (color ~red)
   ?f2 <- (color ~green)
   (test (neq ?f1 ?f2))
```

[10] Si veda più avanti la condizione 'not'.

```
  =>
  (printout t "Rule fires from different facts" crlf))
```

Per confrontare questa regola con i fatti, Clips prende un fatto color che non abbia valore ‘red’ e inserisce il suo indirizzo nella variabile f1, un fatto color che non abbia valore ‘green’ e inserisce il suo indirizzo nella variabile f2; effettua un controllo per verificare che i due indirizzi non siano uguali, cioè si tratti di due fatti diversi, e quindi la attiva. Quando la esegue stampa la stringa indicata.

```
(defrule compare-facts-2
  ?f1 <- (color ~red)
  ?f2 <- (color ~green&:(neq ?f1 ?f2))
  =>
  (printout t "Rule fires from different facts" crlf))
```

Questa regola si comporta esattamente come quello precedente. La verifica per determinare che i due fatti scelti siano distinti viene effettuata però tramite una chiamata di una funzione.

2. La condizione ‘test’

La condizione di un CE di tipo ‘test’ è soddisfatta se la funzione booleana chiamata al suo interno restituisce TRUE. Anche qui, si possono usare variabili che sono state legate in precedenza nelle CE che precedono il test e inoltre è anche possibile chiamare funzioni esterne definite dall’utente. Una lista di funzioni che possono essere usate si trova nella Sez. 12.1 del manuale.

Il CE ‘test’ è valutato se e solo se tutti i CE che lo precedono all’interno di una regola sono soddisfatti.

Esempio :

```
(defrule example-1
 (data ?x)
 (value ?y)
 (test (>= (abs (- ?y ?x)) 3))
 =>)
```

Questo esempio controlla se la differenza in valore assoluto tra due numeri è maggiore o uguale a 3.

3. La condizione ‘or’

Il CE ‘or’ permette di legare tra di loro mediante un ‘or logico’ più condizioni: la condizione perciò è soddisfatta se una o più delle sottocondizioni è soddisfatta. Le sottocondizioni possono essere a loro volta composte da qualsiasi combinazione di CE.

Esempio :

```
(defrule system-fault
  (error-status unknown)
  (or (temp high)
     (valve broken)
     (pump (status off)))
  =>
  (printout t "The system has a fault." crlf))
```

Qui si definisce una regola ‘system-fault’ che è attivata se nella WM esiste un fatto ordinato ‘(error-status unknown)’ ed esiste uno o più dei fatti ‘(temp high)’, ‘(valve broken)’ e ‘(pump (status off)’. Si noti che questo esempio è equivalente a scrivere le tre seguenti seguenti regole separate:

```
(defrule system-fault
  (error-status unknown)
  (pump (status off))
  =>
  (printout t "The system has a fault." crlf))

(defrule system-fault
  (error-status unknown)
  (valve broken)
  =>
  (printout t "The system has a fault." crlf))

(defrule system-fault
  (error-status unknown)
  (temp high)
  =>
  (printout t "The system has a fault." crlf))
```

4. La condizione ‘and’

Clips considera tutti i CE di una regola associati in un ‘and logico’ implicito. Difatti affinché la regola sia attivata, tutti i suoi CE devono essere soddisfatti. E’ fornito comunque un ‘and’ esplicito per poter esprimere liberamente ogni combinazione di CE. La condizione ‘and’ è soddisfatta se tutte le sue sottocondizioni sono soddisfatte. Di nuovo, le sottocondizioni possono essere a loro volta composte da qualsiasi combinazione di CE.

Esempio :

```
(defrule system-flow
  (error-status confirmed)
  (or (and (temp high)
           (valve closed))
      (and (temp low)
           (valve open)))
  =>
  (printout t "The system is having a flow problem." crlf))
```

Affinchè questa regola sia soddisfatta, in WM ci deve essere un fatto ordinato ‘(error-status confirmed)’; inoltre devono essere vere una o entrambe delle seguenti condizioni: in WM sono presenti i fatti ‘(temp high)’ e ‘(valve closed)’ oppure sono presenti ‘(temp low)’ e ‘(valve open)’.

5. La condizione ‘not’

Talvolta la mancanza di informazione può essere utile, cioè si vuole attivare una regola se non esiste un certo fatto o un certo oggetto nella WM. Si usa per questo il CE ‘not’. Esso è soddisfatto se e solo se la sul sottocondizione non è soddisfatta. La sottocondizione può essere a sua volta composta da qualsiasi combinazione di CE. Come è naturale, si può negare un solo CE alla volta, cioè il ‘not’ è unario. Si faccia molta attenzione nell’usare variabili all’interno di una condizione ‘not’: si possono usare liberamente variabili legate in CE precedenti, ma le variabili legate per la prima volta all’interno del ‘not’ possono essere usate solo all’interno di tale condizione.

Esempio 1 :

```
(defrule high-flow-rate
  (temp high)
  (valve open)
  (not (error-status confirmed))
  =>
  (printout t "Recommend closing of valve due to high temp"  crlf))
```

La regola è attivata se oltre ad assere soddisfatti i primi due CE, in WM non c’è il fatto ‘(error-status confirmed)’.

Esempio 2 :

```
  (defrule check-valve
```

```
  (check-status ?valve)
  (not (valve-broken ?valve))
  =>
  (printout t "Device " ?valve " is OK" crlf))
```

Clips controlla se in WM esiste un fatto ordinato di nome 'check-status'; se c'è, lega il valore del suo primo campo alla variabile 'valve'. Successivamente controlla se non esiste in WM un fatto ordinato di nome 'valve-broken' che abbia come valore del suo primo campo lo stesso valore contenuto nella variabile. Se tutte queste condizioni sono soddisfatte, allora Clips attiva la regola.

Esempio 3 :

```
(defrule double-pattern
  (data red)
  (not (data red ?x ?x))
  =>
  (printout t "No patterns with red  a_color  a_color!" crlf ))
```

Qui si effettua un controllo, per verificare che in WM non esistano fatti ordinati aventi nome 'data' con il valore 'red' nel primo campo e un valore uguale nel secondo e terzo. Si noti che la variabile 'x' è legata per la prima volta nel CE 'not', pertanto non può essere usata all'esterno di tale CE.

6. La condizione 'exists'

Questa condizione fornisce un controllo per verificare se in WM esiste almeno una occorrenza di fatti o oggetti che la soddisfano.Questa condizione in Clips è implementata sostituendo al momento della definizione la parola 'exists' con due 'not' nidificati tra di loro.

Esempio :

La regola

```
(defrule example
  (exists (a ?x) (b ?x))
  =>)
```

viene convertita nella seguente:

```
(defrule example
  (not (not (and (a ?x) (b ?x)))) =>)
```

7. La condizione 'forall'

La condizione 'forall' controlla se la sua sottocondizione è verificata da tutte le occorrenze di fatti e/o oggetti individuate da un'altra sottocondizione. Molto spesso la stessa sottocondizione individua sia i CE da controllare, sia le occorrenze di fatti e oggetti da controllare.

Questo CE in Clips è implementato sostituendo la parola chiave 'forall' con una combinazione di CE 'not' e 'and'.

Esempio 1 :

La regola
```
(defrule example
  (forall (a ?x) (b ?x) (c ?x))
  =>)
```

viene convertita in
```
(defrule example
  (not (and (a ?x)
         (not (and (b ?x) (c ?x)))))
  =>)
```

Esempio 2 :

```
CLIPS> (clear)
CLIPS>
(defrule all-students-passed
  (forall (student ?name)
        (reading ?name)
        (writing ?name)
        (arithmetic ?name))
  =>
  (printout t "All students passed." crlf))
CLIPS>
```

Questa regola controlla che tutti gli studenti abbiano passato l'esame di lettura, scrittura e di matematica. Per vedere se la regola è attivabile, Clips cerca se esiste un fatto di nome 'reading' nella WM. Se lo trova, memorizza il valore del secondo campo di tale fatto nella variabile 'name'. Quindi va alla ricerca di un fatto che abbia come nome 'writing' e come secondo campo il valore contenuto in 'name'. Cosa analoga per il fatto 'arithmetic'. Se tutti questi fatti esistono, Clips procede nella ricerca per trovare se c'è un altro fatto 'student'. Se c'è ricomincia i suoi controlli in modo analogo a prima fino ad esaurimento dei fatti 'student'.

Si noti che se in WM non esiste nessun fatto, le condizioni della regola sono automaticamente soddisfatte e perciò la regola è attivata.

8. La condizione 'logical'

Come si sa, alle stesse conclusioni si può arrivare in più modi diversi. Per esempio, le stesse conclusioni possono essere tratte da regole diverse. Clips permette di valutare la concretezza di una conclusione con il CE di tipo 'logical'. Ogni volta che Clips vede questa parola chiave, definisce un cosiddetto 'supporto logico' per il fatto o l'oggetto indicati nella condizione. Un esempio aiuta a capire meglio il concetto. Si considerino le seguenti regole

```
(defrule rule1
  (logical (a))
  (logical (b))
  (c)
  =>
(assert (g) (h)))
```

```
(defrule rule2
  (logical (d))
  (logical (e))
  (f)
  =>
(assert (g) (h)))
```

e si diano al Clips i comandi per visualizzare a schermo tutti i fatti asseriti (watch facts), le attivazioni di regole (watch activations) e le regole eseguite (watch rules).
Dati i fatti 'a', 'b', 'c', 'd', 'e' ed 'f', si ha subito l'attivazione di entrambe le regole.

```
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS. (watch rules)
CLIPS> (assert (a) (b) (c) (d) (e) (f))
==> f-0    (a)
==> f-1    (b)
==> f-2    (c)
==> Activation 0      rule1: f-0,f-1,f-2
==> f-3    (d)
==> f-4    (e)
==> f-5    (f)
==> Activation 0      rule2: f-3,f-3,f-5
<Fact-5>
```

Se si lancia l'esecuzione, il Clips esegue le azioni della seconda regola, cioè inserisce nella WM due nuovi fatti 'g' e 'h' e ci informa che essi dipendono logicamente dalle condizioni che hanno soddisfatto la regola. Successivamente Clips esegue la prima regola che come si vede asserisce gli stessi fatti: Clips ci informa che questi fatti hanno maggiore validità cioè sono conclusioni che ricevono maggiore supporto dai fatti presenti in WM in quanto ribaditi da più parti e perciò maggiormente validi.

```
CLIPS> (run)
FIRE    1 rule2: f-3,f-4,f-5 ; 1st rule adds logical support
==> f-6    (g)
==> f-7    (h)
FIRE    2 rule1: f-0,f-1,f-2 ; 2nd rule adds further support
```

Si elimini ora il fatto 'a': Clips ci informa che una delle condizioni che supportavano le conclusioni 'g' e 'h' non è più valida.

```
CLIPS> (retract 1)
<== f-0    (a)          ; Removes 1st support for (g) and (h)
```

Se adesso si asserisce il fatto 'h' direttamente dal prompt del Clips e non lo si indica all'interno di una condizione 'logic', si dice che 'h' è supportato incondizionatamente. Ciò fa sì che Clips ci dica che il fatto è incondizionato e inoltre cancelli ogni supporto logico per il fatto 'h', cioè se una delle due regole sopra asserisse di nuovo 'h', Clips non ci informerebbe più che esso ha maggiore validità.

```
CLIPS> (assert (h))      ; (h) is unconditionally supported
FALSE
```

Si elimini ora il fatto 'd': nessuna delle due regole è più attivabile: Clips ci informa che anche il secondo supporto al fatto 'g' viene a mancare e perciò lo elimina dalla lista dei fatti. Si noti che il fatto 'h' rimane.

```
CLIPS> (retract 3)
<== f-3    (d)          ; Removes 2nd support for (g)
<== f-6    (g)          ; (g) has no more support
CLIPS> (unwatch all)
CLIPS>
```

Da questo esempio si possono trarre alcune note. Dopo questo esempio si capisce meglio l'utilizzo della condizione 'logic': se non la si fosse usata e si fossero eliminati i fatti 'a' e 'd', il fatto 'g' asserito sarebbe rimasto in WM. Invece con 'logical' il fatto 'g' è stato eliminato quando sono venute a mancare le sue condizioni per esistere. Con la condizione 'logical', insomma, si crea una dipendenza logica tra i vari fatti: alcuni fatti esistono in WM, perché lì ce ne sono di altri. Se questi vengono a mancare, anche i primi non hanno più ragione di essere. La condizione 'logical', perciò crea un legame molto forte tra i fatti e/o gli oggetti in WM; si può dire che con questa condizione si ha un mantenimento dei vincoli di integrità nella WM o in altre parole un mantenimento della consistenza dei fatti contenuti nella WM.
Le sottocondizioni di 'logic' possono essere una qualsiasi combinazione di altre condizioni. Un particolare sulla sintassi: le regole che presentano una condizione 'logic', la devono avere nelle prime N condizioni.

Esempio:

La seguente regola è sintatticamente giusta

```
(defrule ok
  (logical (a))
  (logical (b))
  (c)
  =>
  (assert (d)))
```

mentre queste sono errate.

```
(defrule not-ok-1
  (logical (a))
  (b)
  (logical (c))
  =>
(assert (d)))

(defrule not-ok-2
  (a)
  (logical (b))
  (logical (c))
  =>
  (assert (d)))

 (defrule not-ok-3
  (or (a)
```

```
    (logical (b)))
(logical (c))
=>  (assert (d)))
```

Alle regole è possibile associare delle proprietà; ce ne sono due: la salience e l'auto-focus.
Ad ogni regola può essere assegnata un valore detto *salience* che rappresenta l'importanza della regola rispetto alle altre (o se si vuole una priorità). Una regola con *salience* maggiore di un'altra viene considerata più importante e perciò eseguita per prima. Si noti che la *salience* può essere determinata sia staticamente a programma che dinamicamente durante l'esecuzione. La *salience* di una regola può perciò variare in fase di esecuzione. Essa deve essere determinata da una espressione che viene valutata in un numero di tipo intero compreso tra (-10000 e +10000). All'interno dell'espressione è possibile fare riferimento alle variabili globali. Per *default* la salience di una regola è 0.

Esempio :

```
(defrule test-1
  (declare (salience 99))
  (fire test-1)
  =>
  (printout t "Rule test-1 firing." crlf))

(defrule test-2
  (declare (salience (+ ?*constraint-salience* 10)))
  (fire test-2)
  =>
  (printout t "Rule test-2 firing." crlf))
```

La regola 'test-1' ha come salience il valore 99 espresso da una costante. La regola 'test-2' ha una salience valutata dinamicamente mediante un espressione. Nell'espressione si fa riferimento ad una variabile globale.

La salience può essere valutata in tre diverse occasioni:
1. al momento della definizione della regola;
2. al momento dell'attivazione della regola;
3. alla fine dell'esecuzione di una regola.

Per default, la *salience* di una regola è valutata solo al momento della definizione e rimane costante durante l'intera esecuzione. Con il comando *set-salience-evaluation* si può modificare questo comportamento scegliendo tra una delle tre possibilità sopra elencate.

Esempio:

```
  (set-salience-evaluation when-activated)
```

Dice a Clips di valutare la salience delle regole nel momento in cui esse sono attivate.

Clips fronisce un'altra proprietà per le regole: l'auto-focus. Con questa proprietà impostata a TRUE, ogni volta che una regola diventa attiva, Clips inserisce nello stack dei moduli il modulo a cui la regola appartiene. Per default, la proprietà è impostata a FALSE.

Esempio :

```
(defrule VIOLATIONS::bad-age
  (declare (auto-focus TRUE))
  (person (name ?name) (age ?x&:(< ?x 0)))
  =>
  (printout t ?name " has a bad age value." crlf))
```

Se si scopre che una persona ha un'età negativa, si deve avvertire l'utente dell'errore. Qui si è scelto di mettere tutte le regole relative ai controlli nel modulo VIOLATION. Le regole di questo modulo, grazie

all'auto-focus, sono inserite nell'agenda in modo automatico, appena ne viene la necessità. Se non ci fosse questa proprietà, il programmatore dovrebbe esplicitamente inserire questo modulo ogni volta che necessita di controlli.

Clips offre una serie di funzioni che permettono di ottenere informazioni riguardo alle regole. Si veda la Tabella 4 per una lista di tali funzioni.

Tabella 4

Funzione	Commento	Esempio
Funzioni della defrule (Sez. 12.11)		
get-defrule-list	La funzione ritorna una lista contenente i nomi di tutte le regole definite fino a quel momento.	CLIPS> (clear) CLIPS> (get-defrule-list) () CLIPS> (defrule foo =>) CLIPS> (defrule bar =>) CLIPS> (get-defrule-list) (foo bar)
Defrule-module	La funzione restituisce il nome del modulo in cui la regola indicata è stata definita, cioè il modulo in cui si trova la relativa defrule.	CLIPS>(clear) CLIPS>(defrule foo =>) CLIPS>(defrule-module foo) (MAIN)

Strategie di risoluzione dei conflitti

CLIPS può adottare diverse strategie di risoluzione dei conflitti[11] :

- Profondità : quando si aggiungono nuove istanze di regole nell'agenda, essa sono inserite in ordine di importanza sopra quelle aventi la stessa salience. Per considerare l'importanza delle istanze di regole si considerano nell'ordine la recentezza della regole, la specificitá , la recentezza dei fatti e poi il caso;
- Ampiezza : quando si aggiungono nuove istanze di regole nell'agenda, esse sono inserite in ordine di importanza sotto quelle aventi la stessa salience. Per considerare l'importanza delle istanze di regole si considerano nell'ordine la recentezza della regole, la specificitá , la recentezza dei fatti e poi il caso;
- Semplicitá : quando si aggiungono nuove istanze di regole nell'agenda, le istanze di regole in essa contenute sono ordinate nell'ordine in base al minor numero di test contenuti in ciascuna regola e alla recentezza dei fatti: istanze di regole che usano fatti meno recenti sono condiderate piu'importanti;
- Complessità : quando si aggiungono nuove istanze di regole nell'agenda, le istanze di regole in essa contenute sono ordinate nell'ordine in base al maggior numero di test contenuti in ciascuna regola e alla recentezza dei fatti: istanze di regole che usano fatti meno recenti sono condiderate piu'importanti; é il comportamento inverso della strategia basata sulla semplicitá;
- LEX : si attiva le regola che coinvolge fatti piu` recenti; ad ogni fatto ed istanza viene associato un '*time tag'* che indica la sua 'recentezza' rispetto agli altri fatti. Per determinare la regola da applicare – tra quelle in conflitto – si confrontano i '*time tag'* uno ad uno cominciando da quelli piu` grandi (e… quindi associati a fatti più recenti) fino a trovare un '*time tag*' più grande rispetto al suo

[11] Si ricordi quanto detto sull'importanza delle regole quando si è introdotto il concetto di agenda in Clips a pag. 6.

corrispondente. La regola con '*time tag*' maggiore è messa prima delle altre nell'agenda.

- MEA : anche in questo caso si usano i 'time tag' ; si confrontano quelli associati al primo fatto di ogni regola coinvolta nel conflitto. La regola con 'time tag' maggiore è messa nell'agenda prima delle altre. Se si hanno regole con stesso 'time tag' allora si applica la strategia LEX ai 'time tag' successivi.
- A caso.

Esempio:

Siano dati le seguenti fatti e regole.

```
(defrule a1
                (declare (salience 100))
                (pippo)
                =>(printout t a1 crlf))

(defrule b1
                (declare (salience 100))
                (pippo)
                (pluto)
                =>(printout t b1 crlf))

(defrule c1
                (declare (salience 100))
                (pippo)
                (pluto)
                (minnie)
                =>(printout t c1 crlf))

(defrule a
                (declare (salience 50))
                (pippo)
                =>(printout t a crlf))

(defrule b
                (declare (salience 50))
                (pippo)
                =>(printout t b crlf))

(defrule c
                (declare (salience 50))
                (pippo)
                =>(printout t c crlf))
(defrule d
                (declare (salience 100))
                (paperino)
                => (printout t d crlf))
(defrule e
```

```
                    (declare (salience 100))
                    (paperino)
                    (pippo)
                    => (printout t e crlf))


(deffacts fatti
                    (pippo)
                    (pluto)
                    (minnie))
```

Dopo il comando di reset si asserisca un nuovo fatto:
CLIPS> (assert (paperino))

Variando la strategia di risoluzione dei conflitti, varia anche l'ordine delle istanze di regola nell'agenda.
L'agenda nel caso di strategia in profonditá risulta: e, d, c1, b1, a1, a, b, c.
L'agenda nel caso di strategia in ampiezza risulta: a1, b1, c1, d, e, c, b, a.
L'agenda nel caso di strategia basata sulla semplicitá risulta: a1, d, b1, e, c1, c, b, a.
L'agenda nel caso di strategia basata sulla complessitá risulta: c1, b1, e, a1, d, c, b, a.
L'agenda nel caso di strategia LEX risulta: e, d, c1, b1, a1, c, b, a.
In questo esempio, nel caso si adotti la strategia MEA si otterrá lo stesso ordinamento.

Funzioni e Procedure

Funzioni e procedure sono pezzi di codice eseguibile identificati da un nome che o restituiscono un valore oppure eseguono delle azioni. In generale se il codice restituisce un valore, si parla di funzione, altrimenti si parla di procedure.
In Clips ci sono diversi tipi di funzione: ci sono funzioni definite dall'utente e funzioni definite da sistema che sono in genere scritte in un linguaggio esterno tipo C, Fortran o Ada e integrate nell'ambiente Clips. Nelle precedenti versioni di Clips queste erano le uniche funzioni che si potevano aggiungere all'ambiente. Nella versione più recente sono stati introdotti dei costrutti che permettono di definire funzioni scritte direttamente in Clips. Questi sono la *deffunction* che permette di definire funzioni in linguaggio Clips usando la sintassi Clips e le *defgeneric* e *defmethod* che permettono di creare funzioni 'overloaded', che si comportano cioè in modo diversi a seconda del numero e del tipo di parametri passati.
Tutte le funzioni in Clips utilizzano la notazione prefissa, cioè gli argomenti passati appaiono dopo il nome della funzione. Gli argomenti possono essere a loro volta delle chiamate a funzione.

Esempio :

(+ 3 4 5)	➔ 3 + 4 + 5
(* 5 6.0 2)	➔ 5 * 6.0 * 2
(+ 3 (* 8 9) 4)	➔ 3 + (8 * 9) + 4
(* 8 (+ 3 (* 2 3 4) 9) (* 3 4))	➔ 8 * (3 + (2 * 3 * 4) + 9) * (3 * 4))

Esempi di funzioni matematiche in Clips con a fianco la loro notazione infissa.

E' possibile definire funzioni con un numero di parametri variabili o con parametri di default. CLIPS supporta la ricorsione.

Una definizione di funzione deffunction è composta da 5 elementi:
1. il nome;
2. un commento opzionale;
3. una lista di zero o più parametri;
4. un eventuale carattere jolly per indicare un numero di parametri variabile;
5. il corpo della procedura composto da una serie di azioni.

Una deffunction deve avere un nome diverso da tutte le altre funzioni definite nel sistema e non può essere 'overloaded'. Inoltre deve essere definita prima di essere usata a meno che non sia ricorsiva. La funzione generica accetta esattamente il numero di parametri con i quali essa è stata definita a meno di non aver usato un carattere jolly, nel qual caso il numero minimo di parametri è indicato dal numero di parametri regolari indicati prima del carattere jolly. Nel caso di funzioni con numero di parametri variabili, tali parametri sono inseriti in una lista (multifield).

Esempio :

```
CLIPS> (clear)
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)
```

La funzione è definita con tre argomenti: i primi due accettano valori singoli, il terzo è invece multivalore.

Quando si chiama una funzione, le azioni del suo corpo sono eseguite nell'ordine in cui esse sono state scritte. Il valore della funzione è il valore dell'ultima azione eseguita. Se una funzione non ha azioni oppure si verifica un errore durante la sua esecuzione, Clips ritorna FALSE.
Come già detto le funzioni possono essere ricorsive e anche mutualmente ricorsive. In quest'ultimo caso, visto che tutte le funzioni prima di essere chiamate devono essere definite, si usa una dichiarazione di funzione 'preventiva' - cioè senza azioni - a cui seguirà la definizione vera e propria.

Esempio :

```
(deffunction factorial (?a)
   (if (or (not (integerp ?a)) (< ?a 0)) then
                        (printout t "Factorial Error!" crlf)
   else
                        (if (= ?a 0) then 1
                        else
                        (* ?a (factorial (- ?a 1))))))
```

Questo esempio mostra la definizione della funzione fattoriale in Clips.

Esempio :

```
(deffunction foo ())
(deffunction bar ()
                        (foo))
(deffunction foo ()
                        (bar))
```

Qui si vede come definire due funzioni mutualmente ricorsive. La funzione 'foo' è definita una prima volta senza azioni; si definisce poi completamente la funzione 'bar' che al suo interno chiama la funzione 'foo' e infine si ridefinisce la funzione 'foo' completamente indicando nel suo corpo la chiamata alla funzione 'bar'.

Anche con il costrutto *defgeneric* si possono definire funzioni; esso però è molto più potente in quanto le funzioni definite con tale costrutto possono essere 'overloaded' come già si è accennato prima. Per esempio, si può definire la funzione '+' che concatena due stringhe se i suoi parametri sono stringhe, ma esegue una addizione se i suoi parametri sono numeri. E' chiaro che si debbano fornire codici diversi per eseguire le giuste azione in accordo al tipo di parametri passati: entra in gioco il costrutto *defmethod*[12] con il quale si definiscono le diverse componenti della funzione.

Esempio

```
(defgeneric +
    (defmethod stringhe(string string)
                        (str-cat ?a ?b)
    )
    (defmethod numeri (x y)
                        (+ x y)
    )
)
```

Proprio questo esempio permette di introdurre i 'metodi impliciti': essi sono l'implementazione di funzioni già forniti dal sistema. Per esempio, la somma di due numeri è già definita dal sistema, è già implicita. Basta definire il metodo per la concatenazione di stringhe e la funzione '+' sarà già 'overloaded'.

La chiamata ad una funzione *defgeneric* risulta in Clips computazionalmente oneroso in quanto Clips deve prima esaminare i parametri per sapere quale metodo deve eseguire (ci può essere un rallentamento del 15% - 20% nell'esecuzione). E' raccomandabile perciò non usare le funzioni *defgeneric* quando il fattore tempo è critico. Inutile poi definire una funzione generica con un solo metodo: per questa si usi invece una *deffunction*, che è più veloce durante l'esecuzione. Questo giustifica il perché in Clips ci siano due costrutti per definire le funzioni.

[12] A proposito delle funzioni è importante fare un appunto sulla terminologia. In CLIPS come si è detto una stessa funzione può essere chiamata con diversi tipi di parametri ; per fare ciò è necessario fornire un'implementazione diversa della funzione per ogni combinazione di tipi di parametri prevista. Si dice in CLIPS che si forniscono diversi *metodi* per una funzione. Questa terminologia può provocare confusione quando pensando al paradigma object-oriented si pensa ai metodi di una classe. Per evitare equivoci perciò in CLIPS si preferisce chiamare i metodi di una classe *message-handler*. Quando più metodi di una funzione sono applicabili, CLIPS sceglie uno di essi in base alle precedenze loro assegnate.

Si presti attenzione nell'uso di metodi impliciti: per esempio, si definisca una regola che usa la funzione di sistema '+'; si definisca poi una funzione generica che renda '+' overloaded. Quando la regola è eseguita, essa farà riferimento esclusivamente alla funzione di sistema e non alla funzione generica definita dall'utente. Se la regola venisse invece definita dopo la funzione generica, allora farebbe riferimento a quest'ultima.

Esempio:

```
(defrule pippo
  =>
        ( + «pippo» « e « «pluto»))
(defgeneric +
   (defmethod stringhe (?a STRING ?b STRING)
            (str-cat ?a ?b)
   ))
```

Con la *defgeneric* si è definita la funzione overloaded '+' che ha due metodi: uno implicito che è la funzione somma fornita dal sistema e l'altro esplicito definito dall'utente che è la concatenazione di stringhe. Così scrivendo, la regola 'pippo' fa riferimento non alla defgeneric definita, ma alla funzione '+' di sistema. Ci sarà perciò un errore perché quest'ultima si aspetta degli integer e non delle stringhe. Per far funzionare le cosse, si dovrebbe invertire l'ordine delle due definizioni.

Per definire una funzione si deve indicare il nome della funzione seguito da zero o più metodi. Ogni metodo è composto da sei elementi:

1. un nome;
2. un indice opzionale;
3. un commento opzionale;
4. l'indicazione del tipo dei parametri;
5. un eventuale carattere jolly per indicare un numero di parametri variabile;
6. una sequenza di azione e/o espressioni che saranno eseguiti in ordine quando il metodo è chiamato.

La restrizione di tipo sui parametri è la chiave usata da Clips per determinare quale metodo di una funzione generica deve applicare. Ogni metodo deve differire dagli altri o per il nome o per le restrizioni sui parametri.
E' interessante notare che in Clips, non solo si possono dare restrizioni di tipo sui parametri, ma si può anche effettuare una chiamata (query) ad una funzione che esamina le variabili indicate e restituisce un valore booleano: se esso è TRUE, il metodo è applicato. Risulta chiaro, che il tempo di esecuzione di una chiamata dipende dalla complessità del test effettuato sulle variabili.

Esempio:

```
(defgeneric confronta
   (defmethod positivo ( ?x  INTEGER ( > ?x  0))
      (printout t «Il numero è positivo)
   )
  (defmethod negativo (  ?x  INTEGER (< ?x  0))
     (printout t «Il numero è positivo)
  ))
```

La funzione ‘confronta’ è una funzione generica con un parametro intero. Se esso è maggiore di zero allora usa il metodo ‘positivo’, altrimenti usa il metodo ‘negativo.

Se un parametro non ha alcuna restrizione, significa che quel metodo può accettare qualsiasi argomento in quella posizione.
Può accadere che più metodi di una stessa funzione siano applicabili. E’ necessario perciò che si fornisca un meccanismo per permettere la scelta di uno solo tra di essi. Durante la definizione si può definire una precedenza tra i metodi: essa è determinata da Clips in base alle restrizioni sui parametri. Il metodo con le restrizioni maggiori ha una precedenza maggiore. Per esempio, un metodo che richiede un parametro di tipo integer, ha maggiore precedenza di un metodo che richiede come parametro un numero.
Nel seguito è descritto l’algoritmo che Clips usa per determinare la precedenza. I metodi sono confrontati due a due.

1. Le restrizioni sui parametri dei due metodi sono confrontati uno a uno da sinistra verso destra. Il primo parametro del primo metodo è confrontato con il primo parametro del secondo metodo; e così via per i parametri successivi. Si determina se un parametro è più specifico di un altro in base alle seguenti regole:
   a) un parametro regolare ha precedenza maggiore di un parametro espresso mediante un carattere jolly;
   b) Un parametro di tipo più specifico ha maggiore precedenza di uno meno specifico. Es.: INTEGER è più specifico di NUMBER.
   c) Un parametro con una query ha precedenza su uno che non ce l’ha.
2. Il metodo con il maggior numero di parametri regolari ha precedenza;
3. Un metodo senza caratteri jolly ha maggiore precedenza di uno che ne ha;
4. Un metodo definito prima di un altro ha maggiore precedenza.

Vediamo alcuni esempi:

Esempio 1 :

```
; The system operator '+' is an implicit method                    ; #1
; Its definition provided by the system is:
; (defmethod + ((?a NUMBER) (?b NUMBER) ($?rest NUMBER)))

(defmethod + ((?a NUMBER) (?b INTEGER)))                          ; #2
(defmethod + ((?a INTEGER) (?b INTEGER)))                         ; #3
(defmethod + ((?a INTEGER) (?b NUMBER)))                          ; #4
(defmethod + ((?a NUMBER) (?b NUMBER)
                ($?rest PRIMITIVE)))                              ; #5
(defmethod + ((?a NUMBER)
                (?b INTEGER (> ?b 2))))                           ; #6
(defmethod + ((?a INTEGER (> ?a 2))
                (?b INTEGER (> ?b 3))))                           ; #7
(defmethod + ((?a INTEGER (> ?a 2))
                (?b NUMBER)))                                     ; #8
```

Le precedenze sarebbero: #7,#8,#3,#4,#6,#2,#1,#5. I metodi possono essere divisi in tre gruppi di precedenza

via via decrescente secondo le regole sopra elencate considerando le restrizioni sul primo parametro: A) metodi che hanno una query e una restrizione a tipo INTEGER (#7,#8), B) metodi che hanno una restrizione di tipo a INTEGER (#3,#4), e C) metodi che hanno una restrizione di tipo NUMBER (#1,#2,#5,#6). Il gruppo A ha precedenza sul gruppo B perché i parametri con query hanno maggiore priorità su quelli che non ce l'hanno. Il gruppo B ha precedenza sul gruppo C perché INTEGER è una sottoclasse di NUMBER. L'ordine risultante è: (#7,#8),(#3,#4),(#1,#2,#5,#6). L'ordine all'interno di ogni gruppo ancora non è stato determinato. Per farlo si considera il secondo parametro. #7 ha priorità su #8 perchè INTEGER è sottoclasse di NUMBER. #3 ha priorità su #4 per la stessa ragione. #6 e #2 hanno priorità su #1 e #5 perchè INTEGER è sottoclasse di NUMBER. #6 ha priorità su #2 perchè ha una query e #2 non ce l'ha. L'ordine risulta: #7,#8,#3,#4,#6,#2,(#1,#5). Le restrizioni sul carattere jolly fanno sì che #1 (il metodo implicito) abbia priorità su #5 poichè NUMBER è una sottoclasse di PRIMITIVE. Questo porta al risultato finale: #7,#8,#3,#4,#6,#2,#1,#5.

Esempio 2 :

```
(defmethod foo ((?a NUMBER STRING)))      ; #1
(defmethod foo ((?a INTEGER LEXEME)))     ; #2
```

La precedenza è #2,#1. Sebbene STRING sia sottoclasse di LEXEME, l'ordine è ancora #2,#1 perché INTEGER è sottoclasse di NUMBER, e NUMBER/INTEGER è la coppia di parametri più a sinistra della lista e perciò valutata per prima.

Esempio 3 :

```
(defmethod foo ((?a MULTIFIELD STRING))) ; #1
(defmethod foo ((?a LEXEME)))                                    ; #2
```

La precedenza è #2,#1 perché i tipi della prima coppia di parametri non sono tra loro correlati e #2 ha meno classi nella sua lista di tipi.

Esempio 4 :

```
(defmethod foo ((?a INTEGER LEXEME)))     ; #1
(defmethod foo ((?a STRING NUMBER)))      ; #2
```

Entrambe le coppie di tipi (INTEGER/STRING and LEXEME/NUMBER)no sono tra loro correlati e le liste di tipi sono della stessa lunghezza. L'ordine è determinato perciò l'ordine è determinato dall'ordine in cui i metodi sono stati definiti: #1,#2.

Clips fornisce alcune funzioni per operare sulle procedure. Si veda la Tabella 5 per una lista di tali funzioni.

Tabella 5

Funzione	Commento	Esempio
Funzioni della deffunction (Sez 12.14)		
get-deffunction-list	Ritorna la lista dei nomi di funzioni definite fino a quel momento	CLIPS> (clear) CLIPS> (get-deffunction-list) () CLIPS> (deffunction foo ()) CLIPS> (deffunction bar ()) CLIPS> (get-deffunction-list) (foo bar)
Deffunction-module	Ritorna il nome del modulo in cui è definita la funzione indicata .	CLIPS> (clear) CLIPS> (deffunction foo ()) CLIPS>(deffunction-module foo) MAIN
Funzioni della defgeneric (Sez. 12.15)		
get-defgeneric-list	Restituisce la lista contenente i nomi di tutte le funzioni	CLIPS> (clear) CLIPS> (get-defgeneric-list)

	generiche definite.	() CLIPS> (defgeneric foo) CLIPS> (defgeneric bar) CLIPS> (get-defgeneric-list) (foo bar)
Defgeneric-module	Ritorna il nome del modulo in cui è definita la funzione generica indicata .	CLIPS> (clear) CLIPS> (defgeneric foo) CLIPS> (defgeneric-module foo) MAIN
get-defmethod-list	Restituisce la lista dei nomi con relativo indice di tutti i metodi definiti fino a quel momento. Se si fornisce anche il nome di una funzione generica (opzionale), restituisce la lista dei nomi con l'indice di tutti i metodi definiti per la funzione indicata.	CLIPS> (clear) CLIPS> (get-defmethod-list) () CLIPS> (defmethod foo ((?x STRING))) CLIPS> (defmethod foo ((?x INTEGER))) CLIPS> (defmethod bar ((?x STRING))) CLIPS> (defmethod bar ((?x INTEGER))) CLIPS> (get-defmethod-list) (foo 1 foo 2 bar 1 bar 2) CLIPS> (get-defmethod-list foo) (foo 1 foo 2)
call-specific-method	Questa funzione permette di chiamare un particolare metodo di una funzione generica, evitando così tutto il calcolo delle precedenze.	CLIPS> (defmethod + ((?a INTEGER) (?b INTEGER)) (* (- ?a ?b) (- ?b ?a))) CLIPS> (list-defmethods +) + #2 (INTEGER) (INTEGER) + #SYS1 (NUMBER) (NUMBER) ($? NUMBER) For a total of 2 methods. CLIPS> (call-specific-method + 1 1 2) MTH >> +:#SYS1 ED:1 (1 2) MTH << +:#SYS1 ED:1 (1 2) 3
get-method-restrictions	Restituisce una lista contenente informazioni relative alle restrizioni applicate ai parametri del metodo indicato. Tali informazioni rigurdano il numero minimo e massimo di argomenti, numero di restrizioni, presenza di caratteri jolly e query, etc.	CLIPS> (clear) CLIPS> (defmethod foo 50 ((?a INTEGER SYMBOL) (?b (= 1 1)) $?c)) CLIPS> (get-method-restrictions foo 50) (2 -1 3 7 11 13 FALSE 2 INTEGER SYMBOL TRUE 0

		FALSE 0) CLIPS>

Ancora sui moduli

Ora che si ha una panoramica più completa del linguaggio si possono aggiungere ancora alcune informazioni riguardo ai moduli.

Un modulo può importare i costrutti in 3 modi diversi:

1. Il modulo può importare tutti i costrutti che sono visibili al modulo specificato.

   Esempio :
   (defmodule A (import D ?ALL))

   Il modulo A importa tutti i costrutti definiti e visti dal modulo D.

2. Il modulo può importare ogni costrutto di un determinato tipo che è visibile al modulo specificato.

   Esempio:
   (defmodule B (import D deftemplate ?ALL))

   Il modulo B importa tutti i deftemplate, cioè tutti i fatti non ordinati definiti e visti dal modulo D.

3. Il modulo può importare particolari costrutti visibili al modulo specificato.

   Esempio :
   (defmodule C (import D defglobal vb1 vb2 vb3))

   Il modulo C importa dal modulo D le variabili globali di nome «vb1», «vb2» e «vb3». Per importare i singoli costrutti è necessario specificare anche il tipo di tali costrutti.

La parola chiave ?NONE serve a indicare esplicitamente che il modulo non importa o non esporta alcun costrutto da altri moduli.

Si faccia attenzione al fatto che:

1. I metodi di una procedura e i message-handlers non possono essere singolarmente importati: per farlo si devono importare i loro rispettivi costrutti *defgeneric* e *defclass* e in questo caso tutti i metodi e tutti i message-handlers definiti da tali costrutti sono automaticamente importati.
2. I costrutti *deffacts*, *definstances* e *defrules* non possono essere importati: ogni modulo deve poter vedere solo ciò che è di suo effettivo interesse. Se si potessero esportare o importare fatti, istanze e regole a piacere, il concetto di modulo non avrebbe senso[13].

[13] Il fatto che le *deffacts* non siano esportabili implica che non è possibile esportare o importare fatti ordinati: essi non hanno infatti un costrutto attraverso il quale fare loro riferimento e perciò risulta impossibile renderli visibili in altri moduli al di fuori del loro.

3. Un modulo non può essere né ridefinito, né cancellato una volta che è stato definito. Unica eccezione a tale regola è il modulo MAIN che può essere ridefinito una volta. L'unico modo per cancellare un modulo è dare un comando di *clear* al sistema.

I fatti e le istanze di oggetti sono visibili solo ed esclusivamente ai moduli che importano i rispettivi *deftemplate* e *defclass*. Questo permette di suddividere una base di conoscenza in modo tale che le regole e altri costrutti possano vedere solo i fatti e le istanze che loro interessano. Quando si esegue un programma si eseguono le regole contenute nell'agenda del modulo corrente (si noti che i comandi di *reset* e *clear* rendono il modulo MAIN il modulo corrente). L'esecuzione delle regole di un modulo continua fino a quando o un altro modulo diventa il modulo corrente, oppure fino a quando non ci sono più regole da eseguire nell'agenda, oppure fino a quando non viene eseguita la funzione di *return*. Ogni volta che il modulo corrente esaurisce le regole nella sua agenda, il focus corrente viene rimosso dallo stack dei focus e il modulo in cima allo stack diventa quello corrente. E' possibile cambiare il modulo corrente anche mediante il comando di *focus* (Sez. 5.2, 5.4.10.2, 12.12 e 13.12).

Per operare con i moduli, Clips mette a disposizione le funzioni contenute nella Tabella 6.

Tabella 6

Funzione	**Commento**	**Esempio**
Funzioni per lo stack (Sez 12.12)		
get-focus	La funzione restituisce il nome del modulo in cima allo stack del focus. Se lo stack è vuoto, restituisce FALSE.	CLIPS> (clear) CLIPS> (get-focus) MAIN CLIPS> (defmodule A) CLIPS> (defmodule B) CLIPS> (focus A B) TRUE CLIPS> (get-focus) A CLIPS>
get-focus-stack	La funzione restituisce il nome di tutti i moduli contenuti nello stack in una lista. Tale lista è vuota se lo stack è vuoto. Tale lista è un multifield.	CLIPS> (clear) CLIPS> (get-focus-stack) (MAIN) CLIPS> (clear-focus-stack) CLIPS> (get-focus-stack) () CLIPS> (defmodule A) CLIPS> (defmodule B) CLIPS> (focus A B) TRUE CLIPS> (get-focus-stack) (A B) CLIPS>
pop-focus	La funzione effettua una pop dello stack del focus. Il valore ritornato è ol nome del modulo che era sulla cima. Se lo stack era vuoto, restituisce FALSE.	CLIPS> (clear) CLIPS> (list-focus-stack) MAIN CLIPS> (pop-focus) MAIN

		CLIPS> (defmodule A) CLIPS> (defmodule B) CLIPS> (focus A B) TRUE CLIPS> (list-focus-stack) A B MAIN CLIPS> (pop-focus) A CLIPS> (list-focus-stack) B CLIPS>
Funzioni per i moduli (Sez 12.17)		
get-defmodule-list	La funzione restituisce un multifield contenente i nomi di tutti i moduli definiti fino a quel momento.	CLIPS> (clear) CLIPS> (get-defmodule-list) (MAIN) CLIPS> (defmodule A) CLIPS> (defmodule B) CLIPS> (get-defmodule-list) (MAIN A B) CLIPS>
set-current-module	Questa funzione fa sì che il modulo indicato diventi il modulo corrente. Restituisce il nome del modulo corrente precedente. Se il nome di modulo indicato non è corretto, la funzione non cambia il modulo corrente e restituisce il suo nome.	CLIPS>(reset) CLIPS>(defmodule A) CLIPS>(set-current-module A) MAIN CLIPS>
get-current-module	La funzione restituisce il nome del modulo corrente.	CLIPS>(get-current-module) MAIN CLIPS>

Ancora sulle regole

Finora nulla si è detto sulla struttura della parte destra (LHS) delle regole. Risulterà chiaro comunque che in pratica si può fare di tutto nella LHS: si possono modificare, aggiungere o rimuovere fatti; si possono chiamare funzioni o procedure; si possono variare le strategie di conflitto; si può cambiare il modulo corrente; si possono definire nuovi fatti o nuovi oggetti. Non è posto alcun limite al tipo di operazioni che si possono eseguire nella parte destra di una regola.
Nel seguito si forniscono brevi esempi di regole.

Esempio 1

```
CLIPS> (deftemplate pippo (slot abitazione))
CLIPS> (deffacts mondo (pippo (abitazione topolinia)))
```

```
CLIPS> (defrule myrule
                    ?n <- (pippo (abitazione topolinia))
                    =>
                    (modify ?n (abitazione paperop)))
```

La regola 'myrule' viene attivata se esiste un fatto non ordinato 'pippo' avente nello slot il valore 'topolinia'; inserisce l'indirizzo di tale fatto nella variabile 'n' e nella parte destra usa questa variabile per fare riferimento al fatto e modificarlo.

Esempio 2

```
(defrule watch-it
                    ?n <- (pippo (abitazione topolinia))
                    =>
                    (modify ?n (abitazione paperop))
                    (watch facts)
                    (assert (pippo))
(defrule show-facts
                    ?n <- (pippo (abitazione topolinia))
                    =>
                    (modify ?n (abitazione paperop))
                    (facts)
                    (assert (pippo))
)
```

La regola 'watch-it' se eseguita modifica il fatto che l'ha attivata, mostra tutti i fatti presenti nella WM e asserisce un nuovo fatto 'pippo'.

Esempio 3

```
(defmodule A)
(defmodule B)

(deffacts MAIN::io
                    (pippo))

(deffacts A::tu
                    (topolino))
(deffacts B::egli
                    (minnie))

(defrule MAIN::push
                    (pippo)
                    =>
                    (focus A B)
                    (list-focus-stack)
)
(defrule A::pop
                    (topolino)
                    =>
```

```
                    (pop-focus)
                    (list-focus-stack)
)
(defrule B::go
                    (minnie)
                    =>
                    (assert (pluto))
)
```

Questo programma definisce 2 nuovi moduli A e B. All'inizio dell'esecuzione in ciascuno dei moduli MAIN, A e B è presente un fatto come si vede dai costrutti *deffacts*; si definiscono inoltre tre regole – una per ciascun modulo – che vengono attivate dal fatto presente nel rispettivo modulo. Se si esegue questo programma e si guarda il focus, si noterà come all'inizio nello stack del focus c'è solo MAIN. Dopo l'esecuzione della regola 'push', compariranno anche A e B e dopo l'esecuzione della regola 'pop' compariranno solo B e MAIN.

Esempio 4

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

Questa regola viene attivata se esiste un fatto 'valves-open-through' a cui è associato un valore che è memorizzato nella variabile 'v'. Quando è eseguito, la regola effettua un ciclo che scrive a video una stringa e decrementa il valore contenuto in 'v'. Il ciclo si ferma quando 'v' contiene il valore 0.

Esempio

```
(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)))
```

In questa regola si esegue un costrutto 'if..then..else..'. Se il valore contenuto nella variabile 'v' è pari a 6 allora si asserisce un nuovo fatto, altrimenti si manda un messaggio a video.

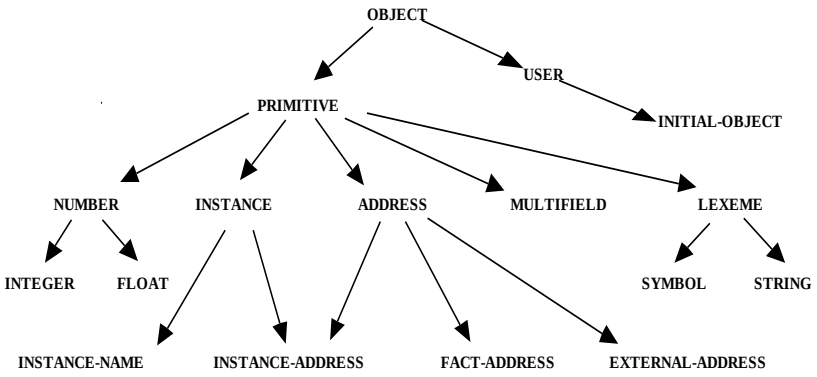
Esempio 5

```
CLIPS> (assert pippo)
CLIPS> (defrule loop 1
                        (pippo)
                        =>
                        (loop-for-count 2 (printout t "Hello world" crlf)))
CLIPS>(run)
Hello world
Hello world
FALSE
CLIPS> (defrule loop 2
                        (pippo)
                        =>
                        (loop-for-count (?cnt1 2 4) do
                                                (loop-for-count (?cnt2 1 3) do
                                                                        (printout t ?cnt1 " " ?cnt2 crlf))))
CLIPS> (run)
2 1
2 2
2 3
3 1
3 2
3 3
4 1
4 2
4 3
FALSE
CLIPS>
```

La regola 'loop 1' esegue un ciclo semplice, mentre la regola 'loop 2' esegue due cicli nidificati. Entrambe stampano a video una stringa.

Clips Object Oriented Language (COOL) (Sez. 9)

L'estensione al formalismo di base del CLIPS (regole di produzione), con programmazione ad oggetti (COOL), permette di esprimere la conoscenza statica sul dominio tramite un formalismo object oriented che aumenta la potenza espressiva del *deftemplate* con l'introduzione del meccanismo di ereditarietà e quello dei metodi (in COOL si parla di messaggi) associati ad ogni classe.
Il controllo sul tipo dei valori inseriti negli slot associati alle classi (cioè le variabili di istanza) è rimasto quello dei *deftemplate*, quindi ogni classe non può essere vista come un nuovo tipo a tutti gli effetti e tali restrizioni possono essere fatte solo sui tipi predefiniti visti precedentemente, ne consegue che molte delle potenzialità del formalismo object oriented come classificatore vengono a mancare.
Confonde quindi il fatto che sia fornita una tassonomia di base tra cui sono presenti i tipi predefiniti, come ad intendere che l'omonimia tra i nomi delle classi ed il nome dei tipi sia un effettivo legame, ciò è sostanzialmente falso.
La tassonomia fornita dall'interprete è la seguente:



Tutte le classi fornite in questa tassonomia, ad esclusione di initial-object, sono «astratte», da cui in pratica, non si possono creare istanze, ma non solo, ad esclusione di USER, tutte queste classi non hanno messaggi ad essi associati e nessuna di esse possiedono slot, servono, in pratica, solamente per sfruttare il meccanismo di ereditarietà.
Il paradigma OO fornito dal COOL non implementa il concetto di metaclasse come nello Smalltalk, i messaggi sono quindi spediti agli oggetti e solo ad essi.
Non essendovi dei metodi di classe, la new dei paradigmi object oriented classici è sostituita da un comando dell'interprete, il make-instance (Sez. 9.6.1).

Classi astratte e concrete.
È possibile, in COOL, la definizione di classi astratte la cui sola utilità è quella di poter ereditare metodi e slot nelle sottoclassi.
Da tali classi non si potranno creare istanze ed ogni tentativo in tal senso provocherà la visualizzazione di un messaggio di errore.
Il tipo della classe (astratta o reale) si dovrà dichiarare tramite la parola chiave *role* dopo la dichiarazione delle sopracclassi , se la classe che si vuole definire è astratta si avrà *(role abstract)*, se invece si vorrà definire una classe concreta (da cui si possono cioè creare delle istanze) si scriverà *(role concrete)*.

Ereditarietà
Il tipo di ereditarietà è multiplo (Sez. 9.3.1); si possono quindi creare dei DAG (Direct Acyclic Graph) per esprimere la conoscenza sul dominio dove eventuali conflitti sui metodi e sull'identità degli slot sono risolti con una strategia fissa.
La strategia usata per risolvere conflitti è la stessa usata nel C++, ossia, se una classe non ha un metodo per gestire un messaggio che gli è stato inviato va a cercare tra quelli delle sue sopracclassi, iniziando dalla prima di sinistra, e se questa non ha il metodo cercato la ricerca prosegue con le sopracclassi della sopracclasse che si è presa in considerazione, con lo stesso ordine, a meno che una di esse sussuma delle classi non ancora considerate nella lista di precedenza.
Per permettere agli utenti di non incorrere in facili errori nella costruzione della tassonomia, il COOL dispone di un comando *(describe-class <class>)* (Sez. 13.11.1.4) che data una classe, ne dà una descrizione esaustiva, fornendo anche la lista ordinata di sopracclassi da cui eredita slot e messaggi (metodi).

Il comando *describe-class* fornisce inoltre una descrizione esaustiva degli slot propri della classe, del loro tipo (se tipati), se vi sono restrizioni di range o di cardinalità, se lo slot in questione è un multislot; fornisce anche una lista dei messaggi che essa può gestire, il loro tipo e la classe in cui sono definiti.

Slots (Sez. 9.3.3)
Il paradigma OO fornito dal COOL consente di dichiarare delle variabili di istanza chiamate "slot", il nome con cui s'identificano le variabili non deve ingannare però, sono molto più simili a delle variabili di istanza del paradigma OO classico che non agli slot del formalismo a frames.
Infatti, eventuali restrizioni sul tipo dei valori contenuti negli slot (filler) sono effettuate solamente con i tipi predefiniti dal COOL e non con le classi (Sez. 9.3.3.1).
La possibilità di compiere del calcolo simbolico fornita dal CLIPS dà all'utente una libertà quasi completa, infatti, se si vuole, si può non definire alcuna restrizione di tipo sullo slot.
La distinzione principale che si effettua è sulla molteplicità di valori associati allo slot, infatti, come per i fatti ordinati del CLIPS è possibile avere dei multislot oppure degli slot semplici; nel caso si decida di avere dei multislot si potrà dare delle limitazioni sulla sua cardinalità (Sez. 9.3.3.11).
Ad ogni slot sono inoltre associati dei *facet* (Sez. dalla 9.3.3.2 alla 9.3.3.11), cioè delle caratteristiche quali presenza di valori di default, tipo di memorizzazione, accessi consentiti e visibilità.
Per ogni slot è possibile scegliere anche se permetterne l'ereditarietà o meno (Sez. 9.3.3.5); è possibile indicare se si vogliono creare automaticamente i metodi di scrittura e/o lettura dello slot (Sez. 9.3.3.9) e se si vuole permettere che partecipi al meccanismo di confronto all'interno della LHS di una regola (Sez. 9.3.3.7).
Come detto precedentemente il COOL non implementa il concetto di metaclasse, è possibile però, mediante il facet di memorizzazione *storage shared* (Sez. 9.3.3.3), definire degli slot il cui contenuto è condiviso da tutte le istanze di una classe e da tutte le sottoclassi che ereditano tale slot, avendo così un qualcosa di simile alle variabili di classe.

Creazione di istanze
All'atto della creazione di un oggetto si devono fornire un insieme di parametri quali:

- Un nome da associargli, tramite cui l'utente potrà identificare l'oggetto per l'invio di messaggi ed eventuali altri comandi (se non sarà fornito un nome, ne verrà dato uno automaticamente usando il generatore automatico di simboli, *gensym**)
- Una classe di appartenenza
- Una lista di coppie nome-slot, valore per inizializzare gli slot associati a tutti gli oggetti appartenenti alla classe di appartenenza.

Per permettere la creazione di oggetti si può operare in due modi differenti, il primo modo è quello che permette la definizione di un insieme di oggetti da caricare nella memoria di lavoro ad un comando di *reset* tramite il costrutto *definstances* (Sez. 9.6.6.1), l'altro è quello che permette di creare dinamicamente delle istanze dal prompt del CLIPS, il *make-instances*.

Esempio:

```
CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
  (role concrete)
                        (slot x (default 34)
      (create-accessor write))
```

```
                        (slot y (default abc)))
CLIPS>
(defmessage-handler A put-x before (?value)
                        (printout t "Slot x set with message." crlf))
CLIPS>
(defmessage-handler A delete after ()
                        (printout t "Old instance deleted." crlf))
CLIPS> (make-instance a of A)
[a]
CLIPS> (send [a] print)
[a] of A
(x 34)
(y abc)
CLIPS> (make-instance [a] of A (x 65))
Old instance deleted.
Slot x set with message.
[a]
CLIPS> (send [a] print)
[a]of A
(x 65)
(y abc)
CLIPS> (send [a] delete)
Old instance deleted.
TRUE
CLIPS>
```

Si noti che per riferirsi ad una istanza si usa sì il suo nome di istanza, ma tra parentesi quadre.

Meccanismo di default (Sez. 9.3.3.2)
Il meccanismo di default è quello dei template precedentemente descritto; ereditando gli slot si ereditano anche tutti i suoi facet, compreso quello dei valori di default; volendo, si possono ridefinire i facets di uno slot ereditato.
Si noti che la definizione di un facet associato ad uno slot ereditato è a tutti gli effetti una ridefinizione del facet stesso e non un suo raffinamento, quindi, la precedente definizione data del facet di default sarà riscritta (ciò vale in generale per tutti i facet).
Il facet di default specifica un valore da assegnare allo slot se all'atto della creazione dell'istanza non viene fornito alcun valore, tale valore è quindi da considerarsi puramente opzionale e modificabile in ogni momento, è quindi un meccanismo di default debole visto da questo punto di vista; se si vuole forzare lo slot ad avere un solo valore, od un insieme di valori, si dovrà ricorrere ad un altro facet, quello della definizione di valori ammissibili per uno slot visto in precedenza per i template (Sez. 11.2).

Esempio:

```
CLIPS> (clear)
CLIPS> (setgen 1)
1
CLIPS>
(defclass A (is-a USER)
```

```
                    (role concrete)
                    (slot foo (default-dynamic (gensym))
                                                    (create-accessor read)))
CLIPS> (make-instance a1 of A)
[a1]
CLIPS> (make-instance a2 of A)
[a2]
CLIPS> (send [a1] get-foo)
gen1
CLIPS> (send (a2) get-foo)
gen2
CLIPS>
```

Visibilità ed accesso agli slot (Sez. 9.3.3.8 e Sez. 9.3.3.4)
Si può limitare sia la visibilità dello slot, suddividendo lo stesso in *public* e *private* come per i formalismi object oriented classici, che l'accesso allo stesso e questa si presenta come una caratteristica deviante rispetto ai formalismi OO più comuni, infatti, si può definire per ogni slot, se esso può venire letto, scritto, entrambe le operazioni, o solo inizializzato tramite il comando *make-instance.*
Gli slot aventi facet di visibilità *private* possono essere letti e modificati solamente da *message-handler* appartenenti alla classe in cui è definito lo slot, quelli che invece hanno il facet di visibilità *public* possono essere letti e modificati da *message-handler* delle sopracclassi in cui è definito lo slot e dalle sottoclassi che lo ereditano.
È anche possibile, come detto precedentemente, la creazione automatica dei *message-handler* per la lettura e la scrittura degli slot tramite il facet *create-accessor*, il *message-handler* automatico per la scrittura avrà come nome quello dello slot preceduto da *put-*, quello automatico di lettura invece avrà come nome quello dello slot preceduto da *get-*.

```
Esempio:

CLIPS> (clear)
CLIPS>
(defclass A (is-a USER)
                    (role concrete)
                    (slot foo (create-accessor write)
                                                    (access read-write))
                    (slot bar (access read-only)
                                                    (default abc))
                    (slot woz (create-accessor write)
                                                    (access initialize-only)))
CLIPS>
(defmessage-handler A put-bar (?value)
                    (dynamic-put (sym-cat bar) ?value))
CLIPS> (make-instance a of A (bar 34))
(MSGFUN3) bar slot in [a] of A: write access denied.
(PRCCODE4) Execution halted during the actions of message-handler put-bar primary in class A
FALSE
CLIPS> (make-instance a of A (foo 34) (woz 65))
[a]
```

```
CLIPS> (send [a] put-bar 1)
(MSGFUN3) bar slot in [a] of A: write access denied.
(PRCCODE4) Execution halted during the actions of message-handler put-bar primary in class A
FALSE
CLIPS> (send [a] put-woz 1)
(MSGFUN3) woz slot in [a] of A: write access denied.
(PRCCODE4) Execution halted during the actions of message-handler put-bar primary in class A
FALSE
CLIPS> (send [a] print)
[a] of A
(foo 34)
(bar abc)
(woz 65)
CLIPS>
```

Nota: una lista dei messaggi di errori viene fornita nel manuale di programmazione di base nell'appendice G, inoltre, grazie ai comandi di *watch* si possono monitorare durante l'esecuzione le eventuali modifiche fatte al contenuto degli slot.

Messaggi e loro gestori (message-handler) (Sez. 9.4)

I *message-handler* sono delle procedure di gestione di messaggi che un oggetto può ricevere e gestire.
Ogni gestore di messaggi (*message-handler*) può eseguire una serie di azioni tra le quali l'invio di altri messaggi ad altri oggetti, tra cui anche se stesso mediante l'identificatore *?self* (Sez. 9.4.2).
La definizione dei *message-handler* propri di una classe si può fare all'atto della definizione della classe oppure più tardi, a run-time. Inoltre i *message-handler* di una classe si possono anche cancellare,quindi il comportamento di una classe può variare nel tempo.
E' comunque possibile avere una lista dei metodi definiti in una classe tramite il comando (*get-defmessage-handler-list*) (Sez. 12.16.1.17).
Ogni *message-handler* ha associato un tipo, (brevemente: *primary*, *before*, *after*, *around*) (Sez. 9.4), che definisce quando questo metodo deve essere eseguito a fronte dell'invio di un messaggio; il default è *primary*, se manca il metodo *primary*, il messaggio non è gestito e viene inoltrato un messaggio di errore, questo anche se sono presenti uno o più metodi degli altri tipi per lo stesso messaggio.
Una volta definito il metodo *primary* per un messaggio, si possono definire *message-handler* degli altri tipi: un gestore di messaggi di tipo *before* viene eseguito prima del corrispondente di tipo *primary*, quello di tipo *after*, invece, viene eseguito dopo.
Grazie a metodi quali *before* ed *after* è possibile eseguire delle azioni di default e di controllo prima e dopo l'esecuzione del metodo *primary*, in particolare è pensabile di inserire dei controlli sui filler degli slot che non sono forniti di default dal COOL nel caso in cui essi non siano appartenenti ad uno dei tipi predefinito.
I *message-handler* accettano anche dei parametri in input, ma non è permesso l'overloading dei gestori, due definizioni di uno stesso gestore di messaggi aventi liste di parametri differenti saranno viste come due definizioni dello stesso *message-handler*, in cui la seconda riscrive la prima.
I gestori di messaggi vengono ereditati nelle sottoclassi in cui sono definiti, quando un oggetto riceve un messaggio cerca prima i corrispondenti *message-handler* di tipo *before* nella lista di precedenza e li esegue in ordine, da quello con priorità maggiore fino a quello con priorità minore, successivamente viene eseguito il corpo del gestore di tipo *primary* (nel caso dei gestori di tipo *primary* il comportamento è diverso, infatti, un nuovo gestore riscrive quello ereditato) e dopo vengono eseguiti in ordine inverso i gestori di tipo *after*.

Per inviare un messaggio ad un oggetto si deve usare il comando *send* fornendo il nome dell'istanza cui si vuole spedire il messaggio, il nome del messaggio ed una lista di parametri, se richiesta.

Esempio:

```
CLIPS> (defclass A
                        (is-a USER)
                        (role concrete)
                        (message-handler message))
CLIPS> (defmessage-handler A message primary ()
                                                (printout t "primary A" crlf))
CLIPS> (defmessage-handler A message before ()
                                                (printout t "before A" crlf))
CLIPS> (defmessage-handler A message after ()
                                                (printout t "after A" crlf))
CLIPS> (defclass B
                        (is-a A)
                        (role concrete))
CLIPS> (defmessage-handler B message primary ()
                                                (printout t "primary B" crlf))
CLIPS> (defmessage-handler B message before ()
                                                (printout t "before B" crlf))
CLIPS> (defmessage-handler B message after ()
                                                (printout t "after B" crlf))
CLIPS> (make-instance b of B)
[b]
CLIPS> (send [b] message)
before B
before A
primary B
after A
after B
CLIPS>
```

Un *message-handler* di tipo *around* si sostituisce a tutti gli altri gestori di massaggi (*before*, *primary* e *after*) per un dato messaggio.

Esempio (riprende il precedente) :

```
CLIPS> (defmessage-handler B message around ()
                                                (printout t "around B" crlf))
CLIPS> (send [b] message)
around B
CLIPS>
```

Se si vuole cambiare questo comportamento si deve usare il comando *call-next-handler*, che richiama il gestore con priorità immediatamente inferiore dopo quello che si sta eseguendo.

```
Esempio (riprende il precedente) :

CLIPS> (defmessage-handler B message around ()
                                        (printout t "around B" crlf)
(call-next-handler))
CLIPS> (send [b] message)
around B
before B
before A
primary B
after A
after B
CLIPS>
```

Il meccanismo di ereditarietà per i *message-handler* è lo stesso di quello degli slot; insieme alla descrizione della classe si può leggere la lista di tutti i gestori ad essa associati, ereditati e non, con la relativa classe di appartenenza.
È possibile, grazie ai comandi di *watch*, di monitorare durante l'esecuzione, la ricezione di messaggi da parte delle istanze e l'esecuzione degli handler.
Grazie alla possibilità offerta dal calcolo simbolico del CLIPS è possibile creare dei simboli, dinamicamente, che sono dei messaggi ed inoltrarli ad un oggetto affinché li gestisca.
Si noti che i messaggi di distruzione di un'istanza (*delete*) e di inizializzazione degli slot di un'istanza sono gestiti come tutti gli altri messaggi, in questo modo, tramite dei *daemons* si possono implementare dei meccanismi di controllo che altrimenti nel formalismo COOL di base non sono forniti.

Daemons
I daemons sono porzioni di codice che vengono eseguite implicitamente quando alcune azioni vengono eseguite su di un'istanza, quali possono essere inizializzazioni di istanze, distruzioni delle stesse, letture o scritture di slot.
Tali azioni, infatti, sono implementate tramite *message-handler* di tipo *primary* associati alla classe di appartenenza dell'istanza, i daemons possono essere facilmente implementati definendo *message-handler* di altro tipo *before* o *after* che riconoscano gli stessi messaggi.
L'utilizzo di daemons puó risultare utile nell'implementazione di controlli di tipo, non forniti nel formalismo di base del COOL.

```
Esempio
CLIPS> (clear)
CLIPS> (defclass A (is-a USER) (role concrete))
CLIPS>
(defmessage-handler A init before ()
                        (printout t "È stata creata un'istanza della classe A..."
        crlf))
CLIPS> (make-instance a of A)
È stata creata un'istanza della classe A...
[a]
CLIPS>
```

Query su insiemi di oggetti ed azioni distribuite (Sez. 9.7)

Il formalismo COOL permette la dichiarazione di query su insiemi di istanze, si possono quindi avere informazioni sugli oggetti presenti ad un dato momento della computazione nella memoria di lavoro, sapere se esiste un sottoinsieme di esse che soddisfa un predicato, ottenere tale sottoinsieme ed infine eseguire delle azioni per ogni oggetto di questo sottoinsieme.
I comandi sono i seguenti:

Funzione	Commento
Any-instancep	Determina se uno o più oggetti soddisfano il predicato
Find-instance	Fornisce come risultato il primo oggetto che soddisfa il predicato
Find-all-instances	Restituisce un multislot contenente tutti gli oggetti che soddisfano un dato predicato
Do-for-instance	Esegue un'azione per la prima istanza che soddisfa un dato predicato
Do-for-all-instances	Esegue un'azione per tutte le istanze che soddisfano un dato predicato
Delayed-do-for-all-instances	Raggruppa tutti gli oggetti che soddisfano un dato predicato ed itera un'azione su tale insieme

Si riportano in seguito alcuni esempi presi dal manuale di programmazione di base:

```
Esempio :
CLIPS>
(defclass PERSON (is-a USER)
  (role abstract)
  (slot sex (access read-only)
        (storage shared))
  (slot age (type NUMBER)
        (visibility public)))
CLIPS>
(defmessage-handler PERSON put-age (?value)
  (dynamic-put age ?value))
CLIPS>
(defclass FEMALE (is-a PERSON)
  (role abstract)
  (slot sex (source composite)
        (default female)))
CLIPS>
(defclass MALE (is-a PERSON)
  (role abstract)
  (slot sex (source composite)
        (default male)))
CLIPS>
(defclass GIRL (is-a FEMALE)
  (role concrete)
  (slot age (source composite)
        (default 4)
```

```
        (range 0.0 17.9)))
CLIPS>
(defclass WOMAN (is-a FEMALE)
  (role concrete)
  (slot age (source composite)
        (default 25)
        (range 18.0 100.0)))
CLIPS>
(defclass BOY (is-a MALE)
  (role concrete)
  (slot age (source composite)
        (default 4)
        (range 0.0 17.9)))
CLIPS>
(defclass MAN (is-a MALE)
  (role concrete)
  (slot age (source composite)
        (default 25)
        (range 18.0 100.0)))
CLIPS>
(definstances PEOPLE
  (Man-1 of MAN (age 18))
  (Man-2 of MAN (age 60))
  (Woman-1 of WOMAN (age 18))
  (Woman-2 of WOMAN (age 60))
  (Woman-3 of WOMAN)
  (Boy-1 of BOY (age 8))
  (Boy-2 of BOY)
  (Boy-3 of BOY)
  (Boy-4 of BOY)
  (Girl-1 of GIRL (age 8))
  (Girl-2 of GIRL))
CLIPS> (reset)
CLIPS>
```

Per l'esecuzione di azioni su di un'insieme di oggetti:

```
Esempio :

CLIPS>
(deffunction count-instances (?class)
  (bind ?count 0)
  (do-for-all-instances ((?ins ?class)) TRUE
    (bind ?count (+ ?count 1)))
 ?count)
CLIPS>
(deffunction count-instances-2 (?class)
  (length (find-all-instances ((?ins ?class)) TRUE)))
```

```
CLIPS> (count-instances WOMAN)
3
CLIPS> (count-instances-2 BOY)
4
CLIPS>
```

In questo esempio, come predicato, è usato il valore booleano TRUE, è ovvio che ad esso si può sostituire un qualsiasi predicato booleano sui valori degli slot dell'oggetto in questione per restringere l'insieme di oggetti su cui eseguire le azioni; si noti che la visibilità della variabile ?ins si estende a tutte le azioni definite nel *do-for-all-instances*, si possono quindi modificare gli oggetti tramite l'invio di messaggi oppure, tramite un'omonimia di variabili, forzare l'uguaglianza di valori, sia di valori di slot che di nomi di istanza.

Esempio :

```
CLIPS>
(find-instance ((?m MAN) (?w WOMAN)) (= ?m:age ?w:age))
((Man-1) (Woman-1))
CLIPS>
```

Si ricordi che una volta definita una tassonomia vale la regola di sussunzione, perciò un oggetto sarà visto come appartenente, non solo alla classe in cui è definito, ma a tutte le sue sopracclassi.

Esempio :

```
CLIPS>
(find-all-instances ((?m MAN) (?w WOMAN)) (= ?m:age ?w:age))
((Man-1) (Woman-1) (Man-2) (Woman-2))
CLIPS>
```

Qui, a differenza dell'esempio precedente, non ci si ferma alla prima coppia di oggetti che soddisfano il predicato, ma si cercano tutte le coppie.

Predicati e funzioni sulla tassonomia (Sez. 12.16)

Il formalismo COOL mette a disposizione tutta una serie di predicati sulla tassonomia, elencati nella sezione 12.16 del manuale di programmazione di base.

I predicati permettono di sapere se una data classe esiste (*class-existp*), oppure se una classe sussume un'altra (*superclassp*), o ancora se una classe è sussunta da un'altra (*subclassp*); questi danno informazioni sulle relazioni che intercorrono tra le varie classi, ma vi è tutta una serie di altri predicati che indagano sulla struttura interna delle classi, se esse sono astratte o meno, se presentano degli slot con un dato nome, se tali slot hanno determinate caratteristiche, oltre che predicati sui *message-handler*.

Vi sono poi funzioni che restituiscono in output i dati riguardanti le classi e le loro strutture interne, come ad esempio la funzione *class-superclasses* che, data una classe, restituisce la lista di tutte le sue sopracclassi:

Esempio :

```
CLIPS> (class-superclasses INTEGER)
(NUMBER)
CLIPS> (class-superclasses INTEGER inherit)
(NUMBER PRIMITIVE OBJECT)
CLIPS>
```

Interfaccia con il formalismo a regole di produzione

Senza un'adeguata interfaccia con il formalismo a regole di produzione, da cui è nato il CLIPS, l'aggiunta della possibilità di gestire gli oggetti sembrerebbe poco sensata.

Una prima possibilità di interfacciare i due formalismi si ha tramite le funzioni sulla tassonomia, la cui struttura si ricorda è dinamica, permettendo la navigazione al suo interno, permettendo l'implementazione di un classificatore, non fornito dal formalismo COOL.

La presenza di predicati riguardanti la tassonomia permette di usare dei test nelle LHS di una regola che coinvolgono gli oggetti ed i loro slot.

Per consentire l'uso degli oggetti nella LHS di una regola, si ricorda, è necessario specificare nella definizione della classe, tramite il ruolo *pattern-match*, che gli oggetti appartenenti a tale classe possono apparire nella parte sinistra di una regola, settandolo a *reactive*, tale ruolo dovrà essere settato per ogni singolo slot che verrà a far parte di un match nella LHS di una regola.

Esempio :

```
(defrule eat-it
                (declare (salience 10))
                ?predator <- (bestia (race ?r)
                                     (posx ?x)
                                     (posy ?y)
                                     (age ?hours)
                                     (timer ?time)
                                     (energy ?level))
                ?prey <- (bestia     (race ?s)
                                     (posx ?x1)
                                     (posy ?y1)
                                     (energy ?add))
                (test (and (<= (abs (- ?x ?x1)) 1)
                                       (<= (abs (- ?y ?y1)) 1)))
                (time ?time)
                (object (is-a ANIMALE) (race ?r) (food $?f))
                (test (member ?s $?f))
                ?counter <- (number-of ?s ?n)
                => (modify ?predator (posx ?x1)
                                     (posy ?y1)
                                     (age (+ ?*prossima-caccia* ?hours))
                                     (timer (+ ?*prossima-caccia* ?time))
                                     (energy (+ ?level ?add)))
```

```
                                        (retract ?prey)
                                         (retract ?counter)
                                        (assert (number-of ?s (- ?n 1))))
```

Nell'esempio, tramite la parola chiave *object*, si ricerca nella tassonomia un oggetto rappresentante di una classe sussunta da ANIMALE la cui razza (race) sia la stessa del *template* bestia (si noti l'omonimia della variabile «r») e tra i vari animali di cui si ciba, la cui lista è nel multislot «food», sia presente anche la razza «s» del secondo *template* (predicato *member*). Il significato di questo test è semplice, rappresentando con i *template* bestia gli animali di un ipotetico territorio e nella tassonomia le proprietà della singola razza e le relazioni preda-predatore, quello che si chiede è se sono presenti nella working-memory due animali, adiacenti, che siano preda e predatore; in questo caso il predatore mangia la preda.
Quello visto nell'esempio è un altro modo di interfacciare il formalismo object oriented con il formalismo a regole di produzione.

Definizione dei template e della tassonomia per l'esempio precedente:

```
(deftemplate bestia
                        (slot race)
                        (slot posx (type INTEGER))
                        (slot posy (type INTEGER))
                        (slot age  (type INTEGER))
                        (slot timer )
                        (slot energy (default 40)))


(defclass ANIMALE
                        (is-a USER)
                        (role concrete)
                        (pattern-match reactive)
                        (slot race (propagation inherit)
                                        (access read-only)
                                        (create-accessor read)
                                        (pattern-match reactive))
                        (multislot food (propagation inherit)
                                        (access read-only)
                                        (create-accessor read)
                                        (pattern-match reactive))
                        (multislot prolific (propagation inherit)
                                        (access read-only)
                                        (create-accessor read)
                                        (pattern-match reactive)
                                        (cardinality 2 2))
                        (slot life-expectance (propagation inherit)
                                        (access read-only)
                                        (create-accessor read)
                                        (pattern-match reactive))
                        (message-handler primary get-prolific-from)
                        (message-handler primary get-prolific-till)
```

```
                    )

(defmessage-handler ANIMALE get-prolific-from primary ()
                    (nth$ 1 (send ?self get-prolific)))

(defmessage-handler ANIMALE get-prolific-till primary ()
                    (nth$ 2 (send ?self get-prolific)))


(defclass ERBIVORO
                    (is-a ANIMALE)
                    (role concrete)
                    (pattern-match reactive)
                    (multislot food (source composite)
                                        (default erba)))

(defclass CARNIVORO
                    (is-a ANIMALE)
                    (role concrete)
                    (pattern-match reactive))

(defclass GAZZELLA
                    (is-a ERBIVORO)
                    (role concrete)
                    (slot race (source composite)
                                        (default gazzella))
                    (multislot food (source composite)
                                        (default erba))
                    (multislot prolific         (source composite)
                                        (default 10 20))
                    (slot life-expectance (source composite)
                                        (default 20)))

(defclass LEONE
                    (is-a CARNIVORO)
                    (role concrete)
                    (slot race (source composite)
                                        (default leone))
                    (multislot food (source composite)
                                        (default gazzella))
                    (multislot prolific         (source composite)
                                        (default 10 20))
                    (slot life-expectance (source composite)
                                        (default 20)))


(defclass TIGRE
                    (is-a CARNIVORO)
                    (role concrete)
```

```
                    (slot race (source composite)
                                        (default tigre))
                    (multislot food     (source composite)
                                        (default gazzella leone))
                    (multislot prolific (source composite)
                                        (default 10 15))
                    (slot life-expectance (source composite)
                                        (default 50)))

(definstances mondo-animale
                    (gazzella of GAZZELLA)
                    (leone of LEONE)
                    (tigre of TIGRE))
```

Si rimanda al manuale per una lista completa di funzioni utili per la programmazione (Sez. 12) e per alcuni argomenti avanzati quali i metodi 'shadowed' (Sez. 8.5.3) e il riordino e l'aggiunta automatici di CE nelle regole (Sez. 5.4.9).

Appendice

Viene fornito un esempio di programmazione in Clips in cui si usano i principali costrutti del linguaggio.
Si definisce infatti una tassonomia che descrive le principali caratteristiche (quali durata della vita, razza, prolificitá, fonti di cibo) di alcuni animali quali tigri, leoni e gazzelle.

```
(defclass ANIMALE (is-a USER)
                  (role concrete)
                  (pattern-match reactive)
                  (slot race  (propagation inherit)
                              (access read-only)
                              (create-accessor read)
                              (pattern-match reactive))
                  (multislot food   (propagation inherit)
                                    (access read-only)
                                    (create-accessor read)
                                    (pattern-match reactive))
                  (multislot prolific     (propagation inherit)
                                          (access read-only)
                                          (create-accessor read)
                                          (pattern-match reactive)
                                          (cardinality 2 2))
                  (slot life-expectance   (propagation inherit)
                                          (access read-only)
                                          (create-accessor read)
                                          (pattern-match reactive))
                  (message-handler primary get-prolific-from)
                  (message-handler primary get-prolific-till)
)

(defmessage-handler ANIMALE get-prolific-from primary ()
      (nth$ 1 (send ?self get-prolific)))

(defmessage-handler ANIMALE get-prolific-till primary ()
```

```
      (nth$ 2 (send ?self get-prolific)))


(defclass ERBIVORO
      (is-a ANIMALE)
      (role concrete)
      (pattern-match reactive)
      (multislot food (source composite)
                  (default erba)))

(defclass CARNIVORO
      (is-a ANIMALE)
      (role concrete)
      (pattern-match reactive))

(defclass GAZZELLA
      (is-a ERBIVORO)
      (role concrete)
      (slot race  (source composite)
                  (default gazzella))
      (multislot food (source composite)
                  (default erba))
      (multislot prolific     (source composite)
                  (default 10 20))
      (slot life-expectance (source composite)
                  (default 20)))

(defclass LEONE
      (is-a CARNIVORO)
      (role concrete)
      (slot race  (source composite)
                  (default leone))
      (multislot food (source composite)
                  (default gazzella))
      (multislot prolific     (source composite)
                  (default 10 20))
      (slot life-expectance (source composite)
                  (default 20)))


(defclass TIGRE
      (is-a CARNIVORO)
      (role concrete)
      (slot race  (source composite)
                  (default tigre))
      (multislot food (source composite)
                  (default gazzella leone))
      (multislot prolific     (source composite)
                  (default 10 15))
      (slot life-expectance (source composite)
                  (default 50)))


(definstances mondo-animale
      (gazzella of GAZZELLA)
      (leone of LEONE)
      (tigre of TIGRE))
```

Oltre a ció sono state definite le regole che descrivono il comportamento dei carnivori e degli erbivori, ad esempio: un carnivoro si sposta nella direzione della sua preda e spende energia nel farlo; un carnivoro che non ha mangiato da un certo lasso di tempo vá a caccia; gli erbivori si muovono sul territorio brucando quindi acquistando energia nel farlo sottraendola dal livello trofico del territorio che si rinnova ogni quanto di tempo.

```
(defrule eat-it
      (declare (salience 10))
      ?predator <- (bestia    (race ?r)
                              (posx ?x)
                              (posy ?y)
                              (age ?hours)
                              (timer ?time)
                              (energy ?level))
      ?prey   <- (bestia      (race ?s)
                              (posx ?x1)
                              (posy ?y1)
                              (energy ?add))
      (test (and (<= (abs (- ?x ?x1)) 1)
                 (<= (abs (- ?y ?y1)) 1)))
      (time ?time)
      (object (is-a ANIMALE) (race ?r) (food $?f))
      (test (member ?s $?f))
      ?counter <- (number-of ?s ?n)
      => (modify ?predator    (posx ?x1)
                              (posy ?y1)
                              (age (+ ?*prossima-caccia* ?hours))
                              (timer (+ ?*prossima-caccia* ?time))
                              (energy (+ ?level ?add)))
         (retract ?prey)
         (retract ?counter)
         (assert (number-of ?s (- ?n 1))))

(defrule move-to-eat
      (declare (salience 5))
      ?predator <- (bestia    (race ?r)
                              (posx ?x)
                              (posy ?y)
                              (age ?hours)
                              (timer ?time)
                              (energy ?level))
      ?prey   <- (bestia      (race ?s)
                              (posx ?x1)
                              (posy ?y1))
      (object (is-a ANIMALE) (race ?r) (food $?f))
      (test (member ?s $?f))
      (time ?time)
      => (if (> ?x ?x1) then (bind ?x-incr -1))
         (if (< ?x ?x1) then (bind ?x-incr 1))
         (if (= ?x ?x1) then (bind ?x-incr 0))
         (if (> ?y ?y1) then (bind ?y-incr -1))
         (if (< ?y ?y1) then (bind ?y-incr 1))
         (if (= ?y ?y1) then (bind ?y-incr 0))
         (modify ?predator (posx (+ ?x ?x-incr))
                           (posy (+ ?y ?y-incr))
                           (age (+ 1 ?hours))
```

```
                    (timer (+ 1 ?time))
                    (energy (- ?level ?*fatica*))))
```

Vengono fornite anche regole generali per ogni animale, che ne regolano l'esistenza, morte naturale, generazione della prole, morte per fame.

```
(defrule am-i-too-much-old-?
      (declare (salience 20))
      ?animal <- (bestia (race ?r) (age ?hours))
      (test (>= ?hours (send (symbol-to-instance-name ?r) get-life-expectance)))
      ?counter <- (number-of ?r ?n)
      => (retract ?animal)
         (retract ?counter)
         (assert (number-of ?r (- ?n 1))))

(defrule i-am-starving
      (declare (salience 20))
      ?animal <- (bestia (race ?r) (energy ?level))
      (test (<= ?level 0))
      ?counter <- (number-of ?r ?n)
      => (retract ?animal)
         (retract ?counter)
         (assert (number-of ?r (- ?n 1))))


(defrule ooooh-my-son
      (declare (salience 10))
      ?animal <- (bestia      (race ?r)
                        (posx ?x)
                        (posy ?y)
                        (age ?hours)
                        (timer ?time)
                        (energy ?level))
      (not (bestia (posx ?x1&:(= ?x1 (+ 1 ?x))) (posy ?y)))
      (test (< ?x 20))
      (time ?time)
      (test (and (>= ?hours (send (symbol-to-instance-name ?r) get-prolific-from))
                 (<= ?hours (send (symbol-to-instance-name ?r) get-prolific-till))))
      ?counter <- (number-of ?r ?n)
      => (modify ?animal (age (+ 1 ?hours))
                     (timer (+ 1 ?time))
                     (energy (div ?level 2)))
         (assert (bestia (race ?r)
                     (posx (+ 1 ?x))
                     (posy ?y)
                     (age 0)
                     (timer (+ 1 ?time))
                     (energy (div ?level 2))))
         (retract ?counter)
         (assert (number-of ?r (+ 1 ?n))))


;; regole per gli erbivori


(defrule move-to-the-right-grasseater
```

```
      ?animal <- (bestia      (race ?r)
                        (posx ?x)
                        (posy ?y)
                        (age ?hours)
                        (timer ?time)
                        (energy ?level))
      (not (bestia (posx ?x1&:(= ?x1 (+ 1 ?x))) (posy ?y)))
      (time ?time)
      (test (< ?x 20))
      (object (is-a ERBIVORO) (race ?r))
      ?trophic <- (livello-trofico ?grass)
      => (if (> ?grass 0) then (modify ?animal (posx (+ 1 ?x))
                         (age (+ 1 ?hours))
                         (timer (+ 1 ?time))
                         (energy (+ ?level 5)))
                         (retract ?trophic)
                         (assert (livello-trofico (- ?grass 5)))
                      else (modify ?animal (posx (+ 1 ?x))
                         (age (+ 1 ?hours))
                         (timer (+ 1 ?time))
                         (energy (- ?level 5)))))

(defrule move-to-the-left-grasseater
      ?animal <- (bestia      (race ?r)
                        (posx ?x)
                        (posy ?y)
                        (age ?hours)
                        (timer ?time)
                        (energy ?level))
      (not (bestia (posx ?x1&:(= ?x1 (- ?x 1))) (posy ?y)))
      (test (> ?x 0))
      (time ?time)
      (object (is-a ERBIVORO) (race ?r))
      ?trophic <- (livello-trofico ?grass)
      => (if (> ?grass 0) then (modify ?animal (posx (- ?x 1))
                         (age (+ 1 ?hours))
                         (timer (+ 1 ?time))
                         (energy (+ ?level 5)))
                         (retract ?trophic)
                         (assert (livello-trofico (- ?grass 5)))
                      else (modify ?animal (posx (- ?x 1))
                         (age (+ 1 ?hours))
                         (timer (+ 1 ?time))
                         (energy (- ?level 5)))))

(defrule move-up-grasseater
      ?animal <- (bestia      (race ?r)
                        (posx ?x)
                        (posy ?y)
                        (age ?hours)
                        (timer ?time)
                        (energy ?level))
      (not (bestia (posx ?x) (posy ?y1&:(= ?y1 (+ ?y 1)))))
      (time ?time)
      (test (< ?y 20))
      (object (is-a ERBIVORO) (race ?r))
      ?trophic <- (livello-trofico ?grass)
      => (if (> ?grass 0) then (modify ?animal (posy (+ 1 ?y))
```

```
                        (age (+ 1 ?hours))
                        (timer (+ 1 ?time))
                        (energy (+ ?level 5)))
                        (retract ?trophic)
                        (assert (livello-trofico (- ?grass 5)))
                     else (modify ?animal (posy (+ 1 ?y))
                        (age (+ 1 ?hours))
                        (timer (+ 1 ?time))
                        (energy (- ?level 5)))))

(defrule move-down-grasseater
      ?animal <- (bestia      (race ?r)
                        (posx ?x)
                        (posy ?y)
                        (age ?hours)
                        (timer ?time)
                        (energy ?level))
            (not (bestia (posx ?x) (posy ?y1&:(= ?y1 (- ?y 1)))))
      (test (> ?y 0))
      (time ?time)
      (object (is-a ERBIVORO) (race ?r))
      ?trophic <- (livello-trofico ?grass)
      => (if (> ?grass 0) then (modify ?animal (posy (- ?y 1))
                        (age (+ 1 ?hours))
                        (timer (+ 1 ?time))
                        (energy (+ ?level 5)))
                        (retract ?trophic)
                        (assert (livello-trofico (- ?grass 5)))
                     else (modify ?animal (posy (- ?y 1))
                        (age (+ 1 ?hours))
                        (timer (+ 1 ?time))
                        (energy (- ?level 5)))))


;; regole per i carnivori


(defrule move-to-the-right-meateater
      ?animal <- (bestia      (race ?r)
                        (posx ?x)
                        (posy ?y)
                        (age ?hours)
                        (timer ?time)
                        (energy ?level))
      (not (bestia (posx ?x1&:(= ?x1 (+ 1 ?x))) (posy ?y)))
      (time ?time)
      (test (< ?x 20))
      (object (is-a CARNIVORO) (race ?r))
      => (modify ?animal (posx (+ 1 ?x))
                     (age (+ 1 ?hours))
                     (timer (+ 1 ?time))
                     (energy (- ?level 5))))

(defrule move-to-the-left-meateater
      ?animal <- (bestia      (race ?r)
                        (posx ?x)
                        (posy ?y)
                        (age ?hours)
```

```
                        (timer ?time)
                        (energy ?level))
      (not (bestia (posx ?x1&:(= ?x1 (- ?x 1))) (posy ?y)))
      (test (> ?x 0))
      (time ?time)
      (object (is-a CARNIVORO) (race ?r))
      =>  (modify ?animal (posx (- ?x 1))
                      (age (+ 1 ?hours))
                      (timer (+ 1 ?time))
                      (energy (- ?level 5))))
(defrule move-up-meateater
      ?animal <- (bestia      (race ?r)
                        (posx ?x)
                        (posy ?y)
                        (age ?hours)
                        (timer ?time)
                        (energy ?level))
      (not (bestia (posx ?x) (posy ?y1&:(= ?y1 (+ ?y 1)))))
      (time ?time)
      (test (< ?y 20))
      (object (is-a CARNIVORO) (race ?r))
      => (modify ?animal (posy (+ 1 ?y))
                    (age (+ 1 ?hours))
                    (timer (+ 1 ?time))
                    (energy (- ?level 5))))

(defrule move-down-meateater
      ?animal <- (bestia      (race ?r)
                        (posx ?x)
                        (posy ?y)
                        (age ?hours)
                        (timer ?time)
                        (energy ?level))
      (not (bestia (posx ?x) (posy ?y1&:(= ?y1 (- ?y 1)))))
      (test (> ?y 0))
      (time ?time)
      (object (is-a CARNIVORO) (race ?r))
      =>(modify ?animal (posy (- ?y 1))
                    (age (+ 1 ?hours))
                    (timer (+ 1 ?time))
                    (energy (- ?level 5))))

(defrule i-stay-here
      (declare (salience -5))
      ?animal <- (bestia      (timer ?time)
                        (energy ?level))
      (time ?time)
      => (modify ?animal (timer (+ 1 ?time)) (energy (- ?level 10))))
```

Come ultime regole vengono scritte quelle che regolano la simulazione, scansione del tempo, stampa dei dati relativi alla popolazione e terminazione della simulazione quando non vi sono piú animali.

```
(defrule tempo
      (declare (salience -10))
      ?t<-(time ?time)
      (number-of leone ?leone)
      (number-of gazzella ?gazzella)
      (number-of tigre ?tigre)
```

```
      ?trophic <- (livello-trofico ?)
      =>
      (printout t "leoni: " ?leone "  gazzelle: " ?gazzella  "  tigre: " ?tigre
crlf )
      (retract ?t)
      (retract ?trophic)
      (assert (livello-trofico ?*produzione-vegetale*))
      (assert (time (+ 1 ?time))))

(defrule stop
      (declare (salience -9))
      (number-of leone 0)
      (number-of gazzella 0)
      (number-of tigre 0)
      => (halt))
```

Come ultimi dati, viene fornita una popolazione di base e vengono definiti i fatti non ordinati usati per rappresentare gli animali, vengono definite le variabili globali usate nella computazione e vengono inseriti i fatti iniziali per poter partire con la simulazione.

```
(deftemplate bestia
      (slot race)
      (slot posx (type INTEGER))
      (slot posy (type INTEGER))
      (slot age  (type INTEGER))
      (slot timer )
      (slot energy (default 40)))

(defglobal
      ?*prossima-caccia* = 3
      ?*fatica* = 3
      ?*produzione-vegetale* = 80)

(deffacts world
      (bestia(race leone)(posx 10)(posy 10)(age 0)(timer 0)
            (energy 100))
      (bestia (race leone) (posx 1) (posy 1) (age 0) (timer 0)
            (energy 100))
      (bestia (race leone) (posx 11) (posy 10) (age 0)
            (timer 0)(energy 100))
      (bestia (race tigre) (posx 15) (posy 15) (age 0)
            (timer 0)(energy 100))
      (bestia (race tigre) (posx 7) (posy 7) (age 0)
            (timer 0)(energy 100))
      (bestia (race gazzella)(posx 10) (posy 5) (age 0) (timer 0))
      (bestia (race gazzella)(posx 4) (posy 8) (age 0) (timer 0))
      (bestia (race gazzella)(posx 5) (posy 3) (age 0) (timer 0))
      (bestia (race gazzella)(posx 3) (posy 4) (age 0) (timer 0))
      (bestia (race gazzella)(posx 3) (posy 5) (age 0) (timer 0))
      (bestia (race gazzella)(posx 3) (posy 7) (age 0) (timer 0))
      (time 0)
      (livello-trofico 80)
      (number-of gazzella 6)
      (number-of leone    3)
      (number-of tigre    2))
```

Prima di impostare il comando di *run* si ricordi di settare il flag di fact-duplication a *true* se si vuole evitare la spiacevole esperienza di vedere identificare due animali appartenenti alla stessa razza, nella stessa posizione come se fossero lo stesso animale.