

CLIPS: un linguaggio per sistemi esperti



SOLUTA.NET

Versione:	1.0
Data di rilascio:	20 Dicembre 2000
Stato:	final
Distribuzione:	
Nome del file:	Clips.pdf

AUTORI

Name	Role	Date	Signature
Pierfranco Ferronato	Chief Architect	2000	



PREMESSA

Le informazioni contenute in questo documento sono state verificate con attenzione e si ritengono esatte. Tuttavia, Soluta.Net non si assume la responsabilità per qualsiasi inesattezza tecnica o per errori tipografici che possono essere contenuti nella presente. In nessun caso Soluta.Net sarà ritenuta responsabile per eventuali perdite dirette, indirette, particolari, accidentali o per qualsiasi altro danno causato da errori, omissioni, errori di stampa od interpretazioni che si trovano in questa pubblicazione, anche se esplicitamente avvertita della presenza di errori od omissioni. Soluta.Net è esente da ogni responsabilità nei confronti di chiunque per fatti, omissioni ed eventuali conseguenze in relazione al contenuto di questa pubblicazione.

CONTATTI

Soluta.net Via Vignola, 9 31037 Loria (TV) Italy
tel: + 39 0423-915547
info@soluta.net www.soluta.net

LICENZA

Questo lavoro è autorizzato con la "Creative Commons Attribution-ShareAlike License". Per visionare una copia di questa licenza, visitate <http://creativecommons.org/licenses/by-sa/1.0/> oppure inviate una lettera a "Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA".

E' possibile:

- copiare, distribuire, modificare e condividere il presente documento
- estrarne delle parti
- farne uso commerciale

Alle seguenti condizioni:

- Attribuzione. Deve essere citato l'autore del documento originale.
- Stessa condivisione. Se questo lavoro viene modificato, trasformato o utilizzato come base, il lavoro risultante deve sottostare a un Licenza simile a questa.
- Per ogni riutilizzo o distribuzione, devono essere indicati agli utenti i termini della licenza di questo documento.
- Si può rinunciare a ciascuna di queste limitazioni solo con il permesso esplicito dell'Autore.



Contenuto

1.SOMMARIO ESECUTIVO.....	4
2.INTRODUZIONE.....	5
3.UN SISTEMA ESPERTO.....	6
4.FATTI, REGOLE, MOTORE INFERENZIALE.....	7
5.I FATTI.....	8
6.LE REGOLE.....	10
7.VARIABILI.....	14
8.FATTI DI CONTROLLO.....	16
9.ELIMINIAMO I FATTI.....	17
10.PIÙ REGOLE.....	18
11.CONCLUSIONI.....	21
12.BIBLIOGRAFIA E RIFERIMENTI.....	22



1. Sommario Esecutivo

Un algoritmo espresso con un linguaggio procedurale non esprime limpidamente la sua funzionalità (che è spesso addirittura criptica), ma solo il modo con cui si realizza: la logica è criptata, nascosta. Perché è difficile leggere il codice scritto da altri? Nei linguaggi procedurali la logica è implicita non esplicita. Il CLIPS ed i sistemi esperti in genere, sono più vicini al dominio del problema e rendono le cose più semplici quando è necessario trattare problemi di logica e di analisi dei dati.



2. Introduzione

Il CLIPS (C Language Integrated Production System) è un linguaggio per sistemi esperti messo a punto Software Technology Branch (STB), presso la NASA/Lyndon B. Johnson Space Center. L'esigenza della NASA era di realizzare un linguaggio che fosse adeguato alla manipolazione della conoscenza umana. I linguaggi di programmazione procedurali, compresi quelli ad oggetti, forniscono principalmente strutture per la logica ed in parte per la manipolazione dei dati; i linguaggi per sistemi esperti invece sono disegnati per trattare una grossa mole di dati. Si tratta di elaborazioni su una base di conoscenza (knowledge base). Le applicazioni più tipiche riguardano simulazioni, ottimizzazioni, ricerca di errori e anomalie, miglioramento di processi, controllo ed analisi.

In questo articolo cercherò di non proporre una sfilza di regole sintattiche e predicati ma per quanto possibile farò in modo di essere pragmatico e di illustrare gli usi concreti che potrete fare di CLIPS nelle vostre realtà lavorative.

3. Un sistema esperto

Un sistema esperto estrae e deduce nuova conoscenza dai dati iniziali. Un sistema esperto, il CLIPS in particolare, si inserisce in parallelo ad un'applicazione preesistente e con un puro apporto processivo, elabora ed astrae nuovi dati ed informazioni. Non si presta certamente a formare un'applicazione a sé stante, tanto che non ha primitive GUI a sua disposizione. Necessita per questo anche di un'adeguata interfaccia utente e certamente un procedimento di interfacciamento verso l'applicazione ospite.

La teoria dei sistemi esperti è nota da tempo, specialmente a livello accademico, e diversi linguaggi sono già stati definiti. I sorgenti, l'interprete, le DLL, gli help file ed il motore inferenziale di Clips sono di pubblico dominio e disponibili presso l'URL <http://www.ghg.net/clips/clips.html>. Tutto quello che vedremo in queste pagine potrà essere facilmente riprodotto da voi. Nel riquadro 1 ci sono chiarimenti su download e installazione.

Prima di poter fare degli esempi utili oppure dare esempi di applicazioni pratiche è necessario un excursus sulle sue caratteristiche tecniche. Non è mia intenzione, nelle prossime sezioni, descrivere tutti gli aspetti del linguaggio ma solo fornire gli elementi necessari per capire la sua applicabilità. Gli esempi che fornirò nel seguito sono presi, con liberi adattamenti, dalla mia esperienza diretta in vari progetti.

4. Fatti, regole, motore inferenziale

I componenti principali di un sistema esperto sono:

- Fatti, Knowledge base: i dati che vengono sottoposti al processo di analisi
- Regole: le regole che deducono la conoscenza dai dati
- Rules Engine: il sistema di controllo dell'esecuzione delle regole.

Seppure la versione 6.0 di CLIPS supporta classi ed oggetti io mi soffermerò solamente sulla parte puramente Expert System. Per descrivere le funzionalità object-oriented del CLIPS sarebbe, infatti, necessario un altro articolo. Lo schema di interazione è presentato in Figura 1. CLIPS è fornito di una shell che permette di interagire con il sistema:

<CLIPS>

Per uscire dall'interprete si dà il comando:

<CLIPS> (Exit)

Tutti i comandi CLIPS sono racchiusi tra parentesi rotonde come in LISP. Forse qualcuno alla NASA aveva dei rimpianti...

Un sistema esperto scritto in CLIPS è pilotato dai fatti che sono lo starting-point del programma. Senza fatti non si può procedere; questi sono i mattoni costitutivi che rappresentano il sistema stesso in esame. Le regole li manipolano per dedurre altri fatti e conseguentemente altra conoscenza. Il controllo del sistema è realizzato dal Rules Engine.

5. I fatti

I fatti sono un elenco di stringhe racchiuse tra parentesi, che vengono caricate nel *Fact List* con il comando *assert*.

```
<CLIPS>(assert (mario))
<Fact-0>
<CLIPS>(assert (pino))
<Fact-1>
<CLIPS>(assert (tony))
<Fact-2>
<CLIPS>_
```

Il comando *Facts* elenca i fatti caricati nel Fact List:

```
<CLIPS>(facts)
f-0 mario
f-1 pino
f-2 tony
For a total of 3 facts
```

Il numero (Fact Index) che compare nella parte sinistra identifica la posizione del fatto; questo valore ne è anche l'identificatore univoco. È necessario tenere in mente che i fatti non sono propriamente dei dati; proviamo ad inserire uno stesso fatto per la seconda volta:

```
<CLIPS> (assert (pino))
FALSE
```

L'interprete ritorna FALSE ad indicare che il fatto già esiste. Non si tratta quindi di un puro elenco ordinato di stringhe, ma di un'informazione pura, nel senso proprio del termine. Come nella realtà, la stessa informazione recepita due volte non è significativa. Inseriamo altri fatti tenendo conto che è buona norma far precedere i fatti da un tag di identificazione del loro significato tramite *assert*.

```
<CLIPS> (reset)
<CLIPS> (assert (parla Paolo con Mario))
<FACT-1>
<CLIPS> (assert (parla Tony con Tony))
<FACT-2>
<CLIPS> (facts)
f-0 (initial-fact)
f-1 (parla Paolo con Mario)
f-2 (parla Tony con Tony)
f-3 (parla Mario con Paolo)
For a total of 2 facts
```

Un fatto può essere ORDERED oppure UNORDERED. Fino ad ora abbiamo visto quelli del primo tipo: (parla Paolo con Mario) è diverso da (parla Mario con Paolo). Questa è una differenza importante. I fatti UNORDERED estendono il significato verso concetti di database relazionali. Fate esperienza con l'inserimento di fatti e noterete come spazi ed a-capo non sono significativi a meno che non siano racchiusi tra doppi apici «».

```
(parla «Mario» con «Tony»)
```

è diverso da

```
(parla «Mario » con «Tony»)
```

Allo stesso modo potete sperimentare come i fatti siano case-sensitive. I lettori pignoli avranno già notato come il fatto 0 non sia stato esplicitamente caricato. È una conseguenza del comando (*reset*) che reinizializza l'ambiente CLIPS. (*initial-fact*) viene sempre caricato dal sistema ed è l'unico fatto sempre presente per default. Il suo utilizzo è necessario per poter scatenare delle regole anche senza la presenza di alcun fatto. (**Riquadro 2**) Prima di procedere ulteriormente con l'approfondimento del CLIPS, salviamo in un file i fatti e le regole inseriti finora:

```
<CLIPS> (save-facts facts)
```

```
<CLIPS> (save rules)
```

Potremo recuperarli in ogni momento dopo aver ripristinato l'ambiente con (*reset*). I comandi necessari sono:

```
<CLIPS> (load-facts facts)
```

```
<CLIPS> (load rules)
```

Per semplificare l'operazione si può creare un file batch con tutti e tre i comandi e caricarlo con:

```
<CLIPS> (batch <nome file>)
```

6. Le regole

Affinché il nostro si possa definire un sistema «esperto», è necessario fornirlo di regole che possano produrre conoscenza a partire dai fatti. Le regole implementano nuclei più o meno elementari di elaborazione. Una regola a sua volta consiste in una o più azioni che vengono scatenate se tutte le condizioni (dette anche pattern) sono verificate. Una regola tipo è la seguente:

```
(defrule nome «commento»  
  (pattern 1)  
  (pattern 2)  
  ..  
  (pattern n)  
=>  
  (rule1)  
  (rule2)  
  ..  
  (rule n)  
)
```

La parte sinistra della freccia (\Rightarrow) si chiama LHS (Left Hand Side) della regola mentre, manco a dirlo, la parte destra RHS. LHS descrive un pattern (anche composto) di condizioni che debbono essere tutte verificate in logica AND sui fatti. RHS, invece, descrive una o più azioni da scatenare. L'avvenuto matching del pattern attiva la regola stessa che porta all'esecuzione (fire) delle azioni presenti nella RHS. È il Rule Engine del sistema ad eseguire e controllare questo procedimento. Creiamo una regola che deduca, dai fatti utilizzati precedentemente, se Paolo e Mario comunicano tra di loro. Poiché non vi sono solamente regole ma anche funzioni, consiglio di usare un prefisso identificativo.

In generale è consigliabile commentare ogni singola riga delle regole ed in particolare lo scopo che ci si prefigge con la regola stessa. Non sono rari i casi di applicazioni con centinaia di regole. Diverse regole collaborano assieme per ottenere lo scopo voluto ed è opportuno farle precedere da una comune stringa di identificazione. La manutenzione del codice CLIPS non è difficoltosa, a patto però che sia chiaro il procedimento utilizzato. Se adeguatamente strutturato, documentato e se si seguono degli accurati e meticolosi standard, la manutenzione del codice è più immediata, questo sia nel caso di interventi evolutivi che correttivi.

```
<CLIPS>(defrule Rcomunicare  
  (parla Paolo con Mario)  
=>  
  (assert (comunica Paolo a Mario))  
)
```

Quando viene mandato in esecuzione il Rule Engine con il comando (run), i pattern vengono confrontati con i fatti. Nel caso venga trovata una coincidenza, viene scatenata l'azione che inserisce un nuovo fatto.

Prima di eseguire la regola, al fine di avere una maggiore visione di cosa accadrà nel sistema, attiviamo un (watch all) e con un (unwatch statistics) eliminiamo le informazioni statistiche che non ci sono di alcuna utilità.

```
<CLIPS> (watch all)
<CLIPS> (unwatch statistics)
<CLIPS> (run)
FIRE 1 Rcomunica: f-0
==> f-3 (comunica Paolo con Mario)
<== Focus MAIN
<CLIPS>(facts)
f-0 (initial-fact)
f-1 (parla Paolo con Mario)
f-2 (parla Tony con Tony)
f-3 (parla Mario con Paolo)
f-4 (comunica Paolo a Mario)
For a total of 5 facts.
```

Ci viene mostrato come il Rule Engine abbia fatto il FIRE della regola Rcomunica avendo trovato un matching con il fatto f-0. L'azione della regola ha prodotto il fatto f-3. In seguito il controllo passa al Rule Engine e la sessione si conclude. Entriamo più in dettaglio.

Il Rule Engine mette in un elenco temporaneo (l'Agenda) la RHS delle regole attivate, cioè di quelle il cui pattern matching è stato verificato con successo. Quando tutte le regole sono state controllate, vengono eseguite le azioni presenti nell'Agenda, che poi vengono rimosse.

Se non vi sono più nuovi fatti l'elaborazione termina altrimenti il ciclo riprende verificando nuovamente le regole ed eseguendo le azioni dall'Agenda, e così via. I fatti che hanno determinato un'attivazione vengono marcati e non sono più utilizzati per la stessa regola.

Questo procedere spiega come mai la regola RComunica non venga eseguita ripetutamente sullo stesso fatto. Provate ad eseguire nuovamente il (run) e vedrete come la regola non verrà più attivata.

Il sistema esperto entrerebbe sempre in loop banali. Può sembrare strano, ma questo ha anche un preciso parallelismo nel mondo reale: uno stimolo che ha attivato un evento sparisce.

Prima di procedere con esempi più interessanti è necessario mettere ulteriormente in luce il procedimento del Rule Engine. Facciamo un (reset), richiamiamo dal file i nostri fatti ed aggiungiamo una nuova regola:

```
(defrule Rparlano
  (comunica Paolo con Mario)
=>
  (assert (Paolo parla con Mario))
)
```

Questa verrà attivata dalla presenza nel Fact List del fatto (comunica Paolo con Mario). Guardiamo l'agenda:

```
<CLIPS>(agenda)
```

```
0 Rcomunica: f-1
```

La regola Rcomunica è attivata ma non eseguita, attende il (run). Lo zero è la priorità dell'attivazione della regola (ci torneremo tra poco), segue il suo nome ed il fatto che ne determina

l'attivazione. Eseguiamo il programma.

```
<CLIPS> (run)
```

```
FIRE 1 Rcomunica: f-1
```

```
==> f-4 (comunica Paolo con Mario)
```

```
==> Activation 0 Rparlano: f-4
```

```
FIRE 2 Rparlano: f-4
```

```
==> f-5 (Paolo parla a Mario)
```

```
<== Focus MAIN
```

- Viene eseguita la regola 1 che fa l'assert di un fatto: f-4.
- Viene poi attivata a priorità 0 la regola Rparlano attraverso il fatto f-4 che era stato aggiunto al passo precedente.
- Viene eseguita l'agenda in cui si trova ora la RHS della regola Rparlano che a sua volta crea il fatto (Paolo parla a Mario).
- Il programma termina.

Non è lecito considerare che l'ordine di esecuzione delle regole sia quello della loro definizione. L'ordine di esecuzione è dato dalla loro priorità che, se non indicata in modo specifico, deve essere considerata casuale. Ora con gli esempi che stiamo trattando non sono immediatamente chiare le conseguenze, ma hanno un grosso impatto. Il Rule Engine, prima di ogni esecuzione dall'Agenda, attiva il Conflict Resolutor che ordina le attivazioni per priorità. Questo procedimento non è tipico del CLIPS ma dei sistemi esperti in genere.

Creiamo una regola che ci permetta di sapere se Paolo e Mario parlano tra loro, se la conversazione è bidirezionale. Per questioni di comodità, usiamo il comando printout per inviare delle informazioni testuali a video anziché creare dei fatti.

```
(defrule Rcomunicano
```

```
(parla Paolo con Mario)
```

```
(parla Mario con Paolo)
```

```
=>
```

```
(printout t «Paolo e Mario parlano tra loro» crlf )
```

```
)
```

```
<CLIPS>(run)
```

```
==> Focus MAIN
```

```
FIRE 1 comunicano: f-2, f-1
```

```
Paolo e Mario parlano tra loro
```

```
<== Focus MAIN
```



Vorremo però creare una regola veramente intelligente che ci dicesse chi comunica in modo bidirezionale: possiamo non sapere come si chiamano le persone!

7. Variabili

Anche in CLIPS esistono variabili globali e locali. Le prime hanno visibilità in tutte le regole, mentre le seconde hanno lo scope della regola in cui sono state introdotte. La loro sintassi è:

```
?<name>
```

I nomi sono case-sensitive. Le variabili non hanno un valore di default e devono essere inizializzate dell'uso. Creiamo una regola che ci dica semplicemente chi comunica:

```
(defrule Rchiparla
  (parla ?mittente con ?destinatario)
  ==>
  (printout t "'parla'" ?mittente crlf)
)
<CLIPS>(unwatch all)
<CLIPS>(agenda)
==> activation 0 Rchiparla: f-1
==> activation 0 Rchiparla: f-2
==> activation 0 Rchiparla: f-3
<CLIPS>(run)
parla Mario
parla Paolo
parla Tony
```

Notiamo che ora le attivazioni e le successive esecuzioni sono tre. Le variabili sono considerate dei segnaposto per il Rule Engine: se non diversamente indicato, rappresentano qualunque stringa. Per ogni fatto che soddisfa un pattern, la variabile assume il valore della stringa. La Tabella 1 riassume quali fatti attivano la Rchiparla e quali valori assumono le variabili. Nell'agenda vi sono tre attivazioni pronte per l'esecuzione ed ognuna di loro ha diversi valori per le variabili ?mittente e ?destinatario. Attenzione che il valore non è disponibile solo alla RHS ma anche alle LHS. Vediamo ora se Mario parla bidirezionalmente con qualcuno:

```
(defrule RparlaconMario
  (parla ?mitt con Mario)
  (parla Mario con ?mitt)
  =>
  (printout t ?mitt « e « ?dest «si parlano tra loro» crlf)
)
<CLIPS>(run)
==> Focus MAIN
FIRE 1 PparlaconMario: f-1, f-2
```

Paolo e Mario si parlano tra loro

Ogni singolo pattern che trova un matching assume il relativo valore. Quando (parla ?mitt con Mario) trova un match in (parla Paolo con Mario) allora ?mitt assume il valore «Paolo» e il secondo pattern diventa (parla Mario con Paolo). La RHS della regola è attivata e posta nell'Agenda per la successiva esecuzione. Creiamo una regola per individuare chi parla con sé stesso:

```
(defrule Rmatti
  (parla ?x con ?x)
=>
  (printout t ?x « parla tra se» crlf)
)
```

Spunto per nuove considerazioni viene dalla regola che potrebbe, in linea di massima, stabilire quali persone comunicano in modo bidirezionale tra loro.

```
(defrule Rchicomunica
  (parla ?x con ?y)
  (parla ?y con ?x)
=>
  (printout t ?x « e « ?y «parlano tra loro» crlf)
<CLIPS>(agenda)
==> Activation 0 Rchicomunica: f-3, f-3
==> Activation 0 Rchicomunica: f-1, f-2
==> Activation 0 Rchicomunica: f-2, f-1
```

Anche senza eseguire il programma vediamo già che le attivazioni pronte in agenda sono tre e non una sola: non è stato infatti indicato infatti che ?y debba essere diverso da ?x.

Nella prima attivazione, il Rule Engine trova per il primo pattern il fatto f-3 cioè (*parla Tony con Tony*) e sia ?x che ?y assumono il valore Tony. Il secondo pattern diventa (*parla Tony con Tony*). Non viene saltato perché era sì stato usato, ma per il primo pattern.

Nella seconda attivazione viene trovato il fatto f-1 e ?x diventa Paolo e ?y Mario. Il secondo pattern diventa perciò (*parla Mario con Paolo*) che viene trovato nel fatto f-2. Il terzo caso è lo stesso del secondo, per gli stessi fatti, ma in ordine opposto. (**Tabella 2**)

8. Fatti di controllo

Possiamo correggere il programma precedente evitando di «sparare» due volte la stessa regola: creiamo un fatto che univocamente identifichi la conoscenza acquisita e lo utilizziamo come semaforo.

```
(defrule Rchicomunica
  (parla ?x con ?y)
  (parla ?y con ?x)
  (not (comunicano ?y ?x))
=>
  (printout t ?x « e « ?y « parlano tra loro» crlf)
  (assert (comunicano ?x ?y))
)
```

Il pattern (*NOT (pattern)*) determina una condizione TRUE quando non esiste un fatto specifico. La prima regola che viene eseguita asserisce che Paolo e Mario comunicano e può essere attivata poiché non esiste ancora il fatto di controllo. La stessa regola non viene più attivata una seconda volta poiché il fatto di controllo adesso è presente.

La tecnica dei fatti di controllo viene spesso usata per garantire di non eseguire più di una volta la stessa regola che verrebbe altrimenti eseguita su fatti diversi. Questa condizione è utile per quei casi in cui l'evento scatenato ha valenza simmetrica e conseguentemente avremmo informazioni non errate di per sé ma solamente ridondanti.

La regola precedente può essere anche scritta usando i constrains. I simboli

| OR

& AND

~ NOT

definiscono, nell'LHS della regola, un vincolo per la variabile che li precede. A questo punto possiamo rivedere in modo più elegante la regola Rchicomunica:

```
(defrule Rchicomunica
  (parla ?x con ?y&~?x) ; parla ?x con ?y che sia diverso da ?x
  (parla ?y con ?x)
=>
  (printout t ?x « e « ?y « parlano tra loro» crlf)
)
```

9. Eliminiamo i fatti

Sì certamente, possiamo anche essere distruttivi con i fatti: (retract <fact index>). Supponendo di avere fatto precedere, nel Fact List, i fatti di controllo dalla stringa «CNT», allora li possiamo eliminare con la regola.

```
(defrule Rdelete_facts
  ?fact<-(CNT $?)
=>
  (retract ?fact)
)
```

La prima ricorrenza della variabile ?fact assume il valore dell'indice del fatto che attiva il pattern matching, il suo valore viene usato poi come parametro per il retract.

Il simbolo \$? ha la valenza di una wildcard, rappresenta qualunque successioni di stringhe, può essere una comodità in questi casi in cui non interessa come sia ulteriormente specificato un fatto.

10. Più regole

La cosa diventa ora più interessante se ci chiediamo: in questo mondo di persone che parlano tutte tra di loro come possiamo sapere se c'è una catena di persone che comunicano in circolo?

Non si può affrontare un problema di questo tipo con una logica procedurale. È necessario trovare un cammino chiuso. Si parte dalle singole persone e si procede lungo il percorso della comunicazione verificando ad ogni passo se si è arrivati al punto iniziale. Tre sono i passi:

- Iniziare
- Cammino verso una direzione
- Verificare di essere arrivati al punto iniziale

Creiamo allora le tre regole per ognuno dei punti.

```
(defrule Rinizio
  (parla ?start con ?y)
=>
  (assert (Follow ?start ?y))
)
(defrule Rcammino
  (follow ?x ?y)
  (parla ?y con ?z)
=>
  (assert (Follow ?x ?z))
)
(defrule Rverifica
  (follow ?x ?x)
  (not (esiste cammino chiuso))
=>
  (assert (esiste cammino chiuso))
  (printout t «Cammino chiuso» crlf)
)
```

Tenete presente che le regole vengono scatenate per ogni nodo del nostro grafo: non c'è un nodo di partenza. Per questo motivo, se esiste un cammino chiuso di quattro nodi, allora la nostra regola Rverifica viene «sparata» per quattro volte. Provate con dei fatti di test e vedrete la quantità di regole che vengono attivate. Non cercate di seguire, con un tipico approccio procedurale, tutta sequenza di fire delle regole; rischiate di perdervi. Trovate la logica che è il fattore guida.

Notate come non abbiamo definito un algoritmo, non abbiamo usato delle strutture dati per la memorizzazione dei dati intermedi, non abbiamo dei cicli while, delle condizioni di uscita: abbiamo formalizzato una logica.

La soluzione che ho dato ha però un paio di debolezze:

- lo stesso cammino chiuso viene individuato n volte, una per ogni nodo che forma il cammino chiuso
- si confonde se vi sono dei cammini che si biforcano

Posso lanciare una piccola sfida, se qui in redazione sono d'accordo, chi fornisce una soluzione efficace per questo ultimo caso descritto, sarà meritevole di pubblicazione.

Mi perdonerete se mi sono permesso di prendere un esempio serio per condurvi a scoprire un poco cos'è un sistema esperto. Mi faccio perdonare proponendo un altro esempio:

La seguente struttura di fatti esprime una struttura familiare in genere:

```
(genitore <genitore> <figlio/a>) ; <genitore> ha
generato <figlio/a>
```

```
(sposi <marito> <moglie>) ; banale!
```

```
(sesso <persona> [marchio]/[femmina]) ; banale !
```

Creiamo una regola che ci identifichi i nonni

```
(defrule Rnonni
```

```
(genitore ?nonno ?figlioa) ; ?nonno ha generato
?figlioa
```

```
(sesso ?nonno maschio) ; ?nonno è un maschio
```

```
(OR (sposi ?figlioa ?meta) ; ?figlioa si è sposato/a
```

```
(sposi ?meta ?figlioa))
```

```
(genitore ?meta ?nipote) ; il marito o la moglie di meta
ha generato ?nipote
```

```
(genitore ?figlioa ?nipote) ; ?figlioa ha generato
?nipote
```

```
=>
```

```
(assert (nonno ?nonno ?nipote)
```

```
)
```

Utilizziamo un fatto come input per una regola che ci dica se due persone sono fratelli:

```
(domanda fratelli Dario Gino)
```

```
(defrule Rfratelli
```

```
?f1<-(domanda fratelli ?primo ?secondo)
```

```
(genitore ?genitore1 ?primo) ; ?primo ha un genitore
?genitore1
```

```
(genitore ?genitore2 ?primo) ; ?primo ha l'altro
genitore ?genitore2
```

```
(OR (sposi ?genitore1 ?genitore2) ; sono sposati
```

```
(sposi ?genitore2 ?genitore1)
(genitore ?genitore1 ?secondo) ; hanno un altro figlio
(genitore ?genitore2 ?secondo) ;
=>
(retract ?f1)
(printout t ?primo « e « ?secondo « sono fratelli» crlf)
)
```

Abbiamo fatto un regola severa per trovare i fratelli naturali, con un pattern meno stretto avremmo trovato anche i fratellastri.

11. Conclusioni

Abbiamo visto, soprattutto nell'ultimo esempio come il CLIPS si presta ad essere letto rapidamente nel linguaggio umano; con i sistemi esperti non vengono stesi degli algoritmi ma delle regole logiche. Un algoritmo non esprime limpidamente la sua funzionalità (che è spesso addirittura criptica), ma solo il modo con cui si esplica: la logica è criptata, nascosta. Uno ulteriore sforzo di strutturazione è spesso necessario per mantenere leggibile un algoritmo scritto con i linguaggi procedurali; logica e metodo non viaggiano affiancati. Perché è così difficile leggere il codice scritto da altri? Perché nei linguaggi procedurali la logica è implicita non esplicita. Il CLIPS ed i sistemi esperti in genere, sono più vicini al dominio del problema e rendono le cose più semplici quando si tratta di problemi di logica e di analisi dei dati: non abbiamo mai scritto nei nostri esempi una clausola IF:THEN:ELSE..

Non cerchiamo di sviluppare però un nuovo gestionale in CLIPS. Su tutto deve valere sempre il buon senso. Tengo a precisare, concludendo, che in generale è fondamentale individuare per ogni problema il giusto strumento. Non si ottengono dei buoni risultati cercando di prendere una mosca con uno stuzzicadente ma neppure con un cannone. Il CLIPS è adeguato solamente in certi domini e questo vale tanto per la Fuzzy Logic quanto per l'OO.



12. Bibliografia e riferimenti

CLIPS User's Guide - Version 6.05 - November 1st 1997 by Joseph C. Giarratano, Ph.D. Intersolv
IRE Rule Engine - Ver 4.0 - Agosto '97

--- Fine documento ---