**Un Tour su CLIPS**

<https://ryjo.codes/tour-of-clips.html#>

# 0. Introduzione

Nell'ultimo anno e mezzo circa ho imparato a conoscere il linguaggio di programmazione [CLIPS](https://www.clipsrules.net/). [Un team di sviluppatori della NASA](https://clipsrules.net/AboutCLIPS.html) ha creato CLIPS a metà degli anni '80. È stato reso di dominio pubblico a metà degli anni '90 e da allora è stato sviluppato attivamente da uno dei membri originali del team, [Gary Riley](http://www.clipsrules.net/AboutTheDeveloper.html).

CLIPS mantiene una certa aria di mistero. Non esiste un servizio simile a npm che centralizzi le "librerie" CLIPS comuni. Ci sono molti scritti sul linguaggio, superficiali, di livello accademico o superiore. Per finire, l'[algoritmo Rete](https://en.wikipedia.org/wiki/Rete_algorithm) centrale in CLIPS, può essere piuttosto intimidatorio a pensarci bene.

Non farsi scoraggiare da queste cose; vale la pena imparare CLIPS. Il suo approccio basato su regole è una buona introduzione all'apprendimento delle reti neurali e dell'intelligenza artificiale. Con questo tutorial si muoveranno i primi passi nei Rules Engines e nei Sistemi Esperti.

Innanzitutto, parleremo del motivo per cui si dovrebbe considerare l'utilizzo di CLIPS. Il cuore di CLIPS è l'[algoritmo Rete](https://en.wikipedia.org/wiki/Rete_algorithm). Questo algoritmo fa alcune cose molto interessanti. Per prima cosa, determina l'ordine in cui le regole dovrebbero essere eseguite. Questo consente di concentrarci su **qual è la logica del dominio** del nostro programma piuttosto che su **come funziona il software**. In secondo luogo, la rete Rete "memorizza nella cache" i calcoli comuni che il software effettuerà. Coloro che hanno scritto una cache in-app comprendono i vantaggi di questa funzionalità.

Un altro grande vantaggio per CLIPS è l'eccellente documentazione e supporto forniti dallo sviluppatore principale [Gary Riley](http://www.clipsrules.net/AboutTheDeveloper.html). Consultare le varie guide di tutti i livelli linkate nella [sezione documentazione](https://clipsrules.net/#documentation) della homepage di CLIPS. Inoltre, il signor Riley è molto attivo in vari forum fornendo supporto per CLIPS. Ancora una volta, visitare il sito web principale di CLIPS per trovare i link alle [varie modalità di supporto](https://clipsrules.net/#support).

Infine, CLIPS è ancora attivamente sviluppato; la sua ultima versione 6.40 è stata pubblicata nel maggio del 2021. Davvero incredibile!

## 0.a Questo Tutorial

Questo tutorial è fortemente ispirato al fantastico [A Tour of Go](https://go.dev/tour/welcome/1). Eseguire gli esempi presentati in questo tutorial cliccando sul pulsante (batch\*) sotto il codice. Il codice nella casella è modificabile, quindi modificare il codice mettersi alla prova. L'output del codice verrà visualizzato nella casella sotto i pulsanti. Naviga tra i capitoli previous [precedente] e next [successivo] cliccando sui link nella parte inferiore dello schermo. Il link a sinistra (←) porterà al capitolo precedente mentre il link destro (→) avanzerà al capitolo successivo. Tra questi due link verrà visualizzato il nome del capitolo corrente. Cliccare su quel link per visualizzare un elenco di tutti i capitoli di questo libro. Cliccando su uno di questi capitoli si passa al capitolo appropriato.

; Benvenuti nel tutorial! Questo è noto come un "commento."

; Ha lo scopo di aiutare gli sviluppatori

; a comprendere meglio una particolare riga di codice.

;

; Cliccare sul pulsante (batch\*) per eseguire questo esempio di codice.

; Si vedrà il codice CLIPS scritto in quest'area della pagina web

; durante questo tour.

;

; Questa è una compilazione WASM di CLIPS eseguita completamente nel browser;

; nessun server è coinvolto!

(println "Hello, world")

### Output:

Hello, world

# 1. Il primo saluto

Il classico "primo programma" è lo "Hello, World". Questo mostrerà come generare del testo dal programma. Cliccare sul pulsante (batch\*) sotto il codice sul lato destro. Si vedrà il testo apparire nella casella sotto i pulsanti.

Con CLIPS, ci sono alcuni modi per stampare il testo sullo schermo. Con (print) verranno stampate le parole, mentre con (println) verranno stampate le parole **con un a-capo alla fine**. Nell'esempio di codice a lato, si usa (print) per stampare la parola "Hello". Usiamo poi (println) per stampare ", World!" e un nuovo carattere di a-capo. Le cose stampate dopo questo testo appariranno nella riga successiva.

(printout t) è come (print) e (println), ma consente di inviare testo a un determinato "router". In questo caso, inviamo il testo al router t che è ciò a cui (print) e (println) inviano il testo per default. Usiamo anche il testo crlf come ultimo argomento di (printout. Lo facciamo in modo che il nostro testo "si spezzi" alla fine della riga; la prossima cosa che stamperemo apparirà sotto questa riga corrente. crlf sta per "carriage return line feed", che significa "spostare il cursore all'inizio della riga successiva".

(format t) consente di stampare una struttura di frase che contiene dei "flag di formato". %s e %n sono flag di formato che specificano rispettivamente una stringa e un ritorno a capo. L'argomento successivo (in questo caso "fun") verrà sostituito al posto di %s. %n verrà sostituito con un carattere di a-capo. Per vedere altri "flag di formato", consultare la [Basic Programming Guide](https://clipsrules.sourceforge.io/documentation/v640/bpg.pdf) -- in particolare la sezione intitolata **12.4.6 Formatted Printing** -- per saperne di più.

(print Hello)

(println ", World!")

(printout t "Welcome to this tour of CLIPS" crlf)

(format t "I hope that you have %s!%n" fun)

### Output:

Hello, World!  
Welcome to this tour of CLIPS  
I hope that you have fun!

# 2. Imparare le Regole

Le Regole sono una parte importante per comprendere il funzionamento di CLIPS. Le Regole sono costituite da un "antecedente" e da un "conseguente". L'antecedente è noto anche come "porzione if" o "parte a sinistra" (LHS). Questa parte è costituita dalle condizioni che devono essere soddisfatte affinché la Regola possa essere eseguita.

Il conseguente è noto anche come "porzione then" o "parte destra" (RHS). Questa parte descrive cosa accadrà quando verrà eseguita la regola.

Questi due pezzi sono separati da un =>. L'antecedente sta prima di questo simbolo mentre il conseguente lo segue.

In questo esempio, definiamo una Regola chiamata will-run-no-matter-what. Questa particolare regola non ha nulla sul lato sinistro, quindi verrà sempre eseguita. Il lato destro specifica che verrà eseguito un (println).

Infine, dopo aver definito la regola, c'è un (run). Affinché funzioni, cliccare sul pulsante (batch\*).

(defrule will-run-no-matter-what

=>

(println "Hello from a rule, world!"))

(run)

### Output:

Hello from a rule, world!

# 3. Solo Fatti

Oltre alle Regole in CLIPS, abbiamo i "Fatti". Un Fatto rappresenta i "dati" in CLIPS. Per inserire i dati in CLIPS, si usa (assert). Una volta inseriti i Fatti con (assert), si dice che stanno "nella memoria di lavoro".

Nel codice a lato, si usa (assert) per asserire un Fatto (name con ryjo nel suo secondo campo.

Stampiamo poi una singola riga di testo con il nostro vecchio amico (println).

(facts) è nuovo. Lo usiamo per stampare i Fatti memorizzati nella Memoria di Lavoro. In questo caso, vediamo stampato 1 fatto con un indice 1 come indicato da f-1.

Infine, con (retract si ritratta questo Fatto appena asserito. Possiamo fare riferimento a questo Fatto tramite il suo indice 1.

Nota: Cliccando più volte di seguito su (batch\*), verrà visualizzato un errore. Questo perché dobbiamo eseguire un (reset) o ripulire l'environment con (clear) per resettare gli ID assegnati ai Fatti nella Memoria di Lavoro. In caso contrario, il Fatto che asseriamo avrà un ID di 2, non 1. Cliccando su (reset) o su (clear) e poi ancora su (batch\*), il codice si comporterà come previsto.

**Bonus**: se si ritorna su questo capitolo dopo il completamento del [Capitolo 5](https://ryjo.codes/tour-of-clips.html#5), si possono vedere più di 1 Fatto. Questo perché non è stato eseguito un (clear) all'inizio del codice per "ripulire" prima l'environment. Si approfondirà (clear) nel [Capitolo 18](https://ryjo.codes/tour-of-clips.html#18).

(assert (name ryjo))

(println "The current facts: ")

(facts)

(retract 1)

(println "After retracting: ")

(facts)

### Output:

The current facts:  
f-1 (name ryjo)  
For a total of 1 fact.  
After retracting:

# 4. Avviare il Motore

I "Fatti" memorizzati nella "Memoria di Lavoro" sono utilizzabili per soddisfare la LHS (left-hand side=la parte sinistra) di una regola. Ricordare: quando il lato sinistro di una Regola è soddisfatto, verrà eseguito il lato destro della Regola.

Nel codice a lato, creiamo una Regola utilizzando (defrule) chiamata greet [saluto]. Questa Regola si "attiverà" quando un Fatto che corrisponde [match] alla forma (name ?) viene inserito nella Memoria di Lavoro. Avviando il motore delle regole con (run), il Lato-Destro di questa Regola verrà eseguita se esiste il Fatto name nel sistema con un secondo campo.

Successivamente, si asserisce con (assert) di un Fatto name col valore ryjo nel suo secondo campo. Una volta fatto questo, la regola greet viene "attivata"; quando si avvia con (run) il Motore delle Regole [Rules Engine] successivamente, verrà eseguita la parte destra, la RHS, di questa regola.

**Bonus**: se si ritorna su questo capitolo dopo il completamento del [Capitolo 5](https://ryjo.codes/tour-of-clips.html#5), si possono vedere più di 1 greeting [saluto]. Questo perché non è stato eseguito un (clear) all'inizio del codice per "ripulire" prima l'environment. Si approfondirà (clear) nel [Capitolo 18](https://ryjo.codes/tour-of-clips.html#18).

(defrule greet

(name ?name)

=>

(println "Hello " ?name "!"))

(assert (name ryjo))

(run)

### Output:

Hello ryjo!

# 5. Negativo, Ghostrider

A volte è necessario attivare una Regola quando nella Memoria di Lavoro è presente un Fatto che non corrisponde a un determinato pattern. Il carattere tilde (~) viene usato per negare un determinato pattern. È come dire "un Fatto che non deve avvenire qui".

Notare che non siamo in grado di utilizzare i nomi non,corrispondenti [non-match] nella parte destra di questa regola come abbiamo fatto nel [Capitolo 4](https://ryjo.codes/tour-of-clips.html#4). Tratteremo come utilizzare questi nomi nel [Capitolo 6](https://ryjo.codes/tour-of-clips.html#6).

Innanzitutto, si esegue (clear) per ripulire l'environment da tutte le Regole e i Fatti precedentemente definiti dall'Environment CLIPS. Quindi si asseriscono con (assert due nomi (name: Zach e Zethus. Infine, la nostra Regola non-ryjos-only si "attiverà" quando un Fatto corrispondente [match] al pattern (name ~ryjo) viene immesso nella memoria di lavoro. Questo pattern potrebbe essere letto come "un Fatto che inizia con name e non ha ryjo nel suo secondo campo". Infine, la parte destra, RHS, di questa Regola stampa il testo "Hello, not ryjo!"

**Bonus:** Dopo l'esecuzione con (batch\*) di questo codice, provare a navigare nel [Capitolo 4](https://ryjo.codes/tour-of-clips.html#4); si vedrà che verrà eseguita la Regola greet definita precedentemente per questi due nuovi nomi. Questo perché gli esempi in questo tutorial utilizzano lo stesso Environment CLIPS. In questo esempio, utilizziamo (clear) all'inizio per rimuovere tutto ciò che è stato definito negli esempi precedenti. Si vedrà meglio questa convenzione nei prossimi capitoli.

(clear)

(assert (name Zach) (name Zethus))

(defrule non-ryjos-only

(name ~ryjo)

=>

(println "Hello, not ryjo!"))

(run)

### Output:

Hello, not ryjo!  
Hello, not ryjo!

# 6. Catturare il nome

Ora combineremo due concetti presentati in precedenza utilizzando una &. Innanzitutto, cattureremo il valore nel primo campo con ?n ([Capitolo 4](https://ryjo.codes/tour-of-clips.html#4)). Poi, specificheremo "not ryjo" con ~ ([Capitolo 5](https://ryjo.codes/tour-of-clips.html#5)). Il simbolo & agisce come un "and". Possiamo leggere il pattern (name ?n&~ryjo) come "un valore AND non è ryjo."

Inoltre, con (assert) si asseriscono tre fatti. Solo due di essi "attiveranno" la regola non-ryjos-only che definiamo subito sotto. Qualcosa di carino da prendere nota: dobbiamo solo chiamare (assert) una sola volta per inserire tre nuovi Fatti nella Memoria di Lavoro.

Definiamo qui una seconda regola ryjo per dimostrare "l'opposto" della regola non-ryjos-only. Questa si attiverà quando un Fatto entra nella Memoria di Lavoro che assomiglia esattamente a (name ryjo).

Notare come le Regole in questo esempio non vengono eseguite nell'ordine in cui sono state scritte. La ragione di ciò è che **l'ordine con cui le regole vengono eseguite è determinato algoritmicamente**. Maggiori informazioni su come CLIPS determina l'ordine in cui le regole vengono eseguite nella sezione **5.2 Basic Cycle Of Rule Execution** del [Basic Programming Guide](https://clipsrules.sourceforge.io/documentation/v640/bpg.pdf). Possiamo usare (salience per dichiarare esplicitamente l'ordine in cui vengono eseguite le regole. Ne discuteremo nel [Capitolo 15](https://ryjo.codes/tour-of-clips.html#15).

(clear)

(assert

(name Gabe)

(name ryjo)

(name John))

(defrule non-ryjos-only

(name ?n&~ryjo)

=>

(println "Hello, " ?n "!"))

(defrule ryjo

(name ryjo)

=>

(println "Oh hey there"))

(run)

### Output:

Oh hey there  
Hello, John!  
Hello, Gabe!

# 7. And/Or

Possiamo avvalerci dei “Vincoli Connettivi” per affinare meglio la parte sinistra, LHS, delle nostre Regole. Ad esempio, nella Regola ors nell'editor del codice, effettuiamo il "match" su un Fatto il cui secondo campo corrisponde a Gabe o Peter. Questo può essere letto come "Un Fatto name nel cui primo campo memorizziamo la variabile ?name e corrisponde [match] o a Gabe o a Peter."

Nella Regola and-nots, memorizziamo il secondo campo di un Fatto name nella variabile ?name. Questa volta diciamo "il valore nel primo campo non può corrispondere [match] a ryjo né a Zach e nemmeno a Zethus".

(clear)

(assert

(name Gabe)

(name ryjo)

(name John)

(name Zach)

(name Zethus)

(name Peter))

(defrule ors

(name ?name&Gabe|Peter)

=>

(println "Greetings, " ?name))

(defrule and-nots

(name ?name&~ryjo&~Zach&~Zethus)

=>

(println "Hi there, " ?name))

(run)

### Output:

Hi there, Peter  
Hi there, John  
Hi there, Gabe  
Greetings, Peter  
Greetings, Gabe

# 8. Vincoli di Predicati

I Vincoli del Predicati sembrano ostici, ma sono piuttosto semplici. Fondamentalmente consentono di specificare ulteriormente quale valore contiene un campo. Ad esempio, con le nostre attuali conoscenze, possiamo specificare corrispondenze *esatte* per valori come (name ?n&ryjo). Le Condizioni dei Predicati ci consentono di eseguire una funzione sul valore per determinare se corrisponde [match].

Nel nostro esempio di codice per questo capitolo, scriviamo una Regola di base che descrive una regola che consente l'ingresso in un bar. Negli USA è richiesto che una persona abbia almeno 21 anni. Se hanno meno di 21 anni non possono entrare nei bar.

Notare la forma familiare di ?variable&. Usiamo qui il simboll and (&) per indicare che la ?variable deve corrispondere [match] a determinati criteri. Questo ci permette di leggerlo come "l'età di una persona AND è maggiore o uguale a 21" nella Regola allow-entry-to-bar.

Notare inoltre che utilizziamo 2 diversi Fatti nella Memoria di Lavoro per attivare le nostre regole: i Fatti name e age. Notare anche che nella parte sinistra, LHS, delle nostre Regole specifichiamo che il primo campo di questi Fatti deve corrispondere [match] utilizzando ?person in entrambi. Poiché ?person è utilizzata in entrambe queste condizioni, diciamo che questa Regola è "attivata" quando questi Fatti hanno lo stesso valore nel loro primo campo.

Per esempio, il Fatto (name ryjo) e il Fatto (age ryjo 32) attiveranno, assieme, la Regola allow-entry-to-bar. Entrambi i valori nel primo campo corrispondono [match] a questi Fatti *E* il numero 32 è maggiore o uguale a 21. I Fatti (name ryjo) e (age someone 20) non attiveranno la Regola no-entry-to-bar. Il numero 20 è effettivamente inferiore a 21, ma questi due Fatti non attiveranno insieme la Regola. Tuttavia, i 2 Fatti (name someone) e (age someone 20) la attiveranno. Non solo 20 è minore di 21 come indicato nella Condizione del Predicato (>= ?a 21), ma il Simbolo someone corrisponde [match] in entrambi i Fatti.

(clear)

(defrule allow-entry-to-bar

(name ?person)

(age ?person ?a&:(>= ?a 21))

=>

(println ?person " is over 21 years old and can enter the bar!"))

(defrule no-entry-to-bar

(name ?person)

(age ?person ?a&:(< ?a 21))

=>

(println ?person " is not allowed in (under 21)."))

(assert

(name ryjo) (age ryjo 32)

(name someone) (age someone 20))

(run)

### Output:

someone is not allowed in (under 21).  
ryjo is over 21 years old and can enter the bar!

# 9. Test Condizionali

Questo esempio è simile alla sezione precedente. Invece di utilizzare &:, definiamo i nostri vincoli utilizzando l'Elemento Condizionale (test.

L'output di questo esempio dovrebbe essere molto simile a quello dell'esempio precedente.

(clear)

(defrule allow-entry-to-bar

(name ?person)

(age ?person ?a)

(test (>= ?a 21))

=>

(println ?person " is over 21 years old and can enter the bar!"))

(defrule no-entry-to-bar

(name ?person)

(age ?person ?a)

(test (< ?a 21))

=>

(println ?person " is not allowed in (under 21)."))

(assert

(name ryjo) (age ryjo 32)

(name someone) (age someone 20))

(run)

### Output:

someone is not allowed in (under 21).  
ryjo is over 21 years old and can enter the bar!

# 10. Uno per tutti

A volte si vuol sapere se tutti i Fatti nella Memoria di Lavoro corrispondono [match] a un determinato pattern. (forall fornisce questo controllo. Nel nostro esempio, (forall corrisponderà [match] se ogni Fatto (name nella Memoria di Lavoro può corrispondere [match] con un Fatto (age (e viceversa).

Nel nostro esempio, il Fatto (name another) non ha un Fatto (age con un valore corrispondente [match] ?n. Pertanto, il nostro motore ci dice che "il Roster è stato rifiutato".

(clear)

(assert

(name ryjo) (age ryjo 32)

(name someone) (age someone 20)

(name another))

(defrule accept-roster

(forall (name ?n) (age ?n ?))

=>

(println "Roster has been accepted"))

(defrule decline-roster

(not (forall (name ?n) (age ?n ?)))

=>

(println "Roster has been declined"))

(run)

### Output:

Roster has been declined

# 11. Gli Alieni Esistono

Questo è un esempio un po’ più lungo, ma mostra alcune cose combinate insieme. (exists corrisponderà [match] una volta se un Fatto nella Memoria di Lavoro corrisponde al pattern. Non verrà eseguito nuovamente finché la condizione continua a corrispondere [match]. Nel nostro esempio, confrontiamo "qualsiasi fatto che inizia con (age ed è seguito da due valori". Quando usiamo ?, non importa cosa siano questi valori; conta solo che ci siano.

Provare a modificare la Regola any-age-reported per acquisire il ?name e (println ?name) nella sezione destra, RHS, della Regola.

Usiamo anche (forall per determinare quando ogni utente ha segnalato la propria età.

Può sembrare strano rinunciare al controllo dell'ordine di esecuzione del codice. Da tenere presente che questo ci consente di concentrarci sulla Logica piuttosto che sul Flusso di Controllo del dominio. Rivedere la sezione **Performance Considerations** nella [Basic Programming Guide](https://clipsrules.sourceforge.io/documentation/v640/bpg.pdf) per imparare come scrivere codice in armonia con questo approccio.

(clear)

(defrule any-age-reported

(exists (age ? ?))

=>

(println "Users have begun to report their age"))

(defrule no-ages-reported

(not (age ? ?))

=>

(println "No user has reported their age yet..."))

(defrule reported-age

(name ?n)

(age ?n ?a)

=>

(format t "%s has reported their age: %d%n" ?n ?a))

(defrule has-not-reported-age

(name ?n)

(not (age ?n ?))

=>

(println ?n " has not yet reported their age"))

(defrule all-ages-reported

(forall (name ?n) (age ?n ?))

=>

(println "Everyone in the system has reported their age"))

(assert

(name ryjo)

(name someone)

(name another))

(println "First run results:")

(println "------------------")

(run)

(println " ")

(assert (age ryjo 32))

(println "Second run results:")

(println "-------------------")

(run)

(println " ")

(assert (age someone 20)

(age another 52))

(println "Third run results:")

(println "------------------")

(run)

### Output:

First run results:  
------------------  
another has not yet reported their age  
someone has not yet reported their age  
ryjo has not yet reported their age  
No user has reported their age yet...  
   
Second run results:  
-------------------  
Users have begun to report their age  
ryjo has reported their age: 32  
   
Third run results:  
------------------  
another has reported their age: 52  
Everyone in the system has reported their age  
someone has reported their age: 20

# 12. Mi leggi?

A volte è necessario ricevere un input dall'utente anziché "scolpire" il nome, l'età e altri dettagli. Possiamo ricevere input dall'utente utilizzando (readline).

In questo esempio, prendiamo l'input dell'utente, poi gli diciamo cosa ha inserito. Nota: quando si esegue questo esempio, sullo schermo verrà visualizzato un messaggio. Inserire il testo nel campo, poi cliccare sul pulsante "OK".

Allo stesso modo, (read) prenderà l'input da un utente, ma richiederà solo una singola parola. Cambiare (readline) con (read) nell'editor del codice. Se si inserisce "foo bar 123" nella casella di testo e poi si clicca sul pulsante "OK", verrà scritto solo "foo".

(clear)

; replace (readline) with (read) and note the differences

(println "You entered: " (readline))

(run)

### Output:

You entered: Ciao

# 13. La parola segreta

**Attenzione:** notare che in questo esempio c'è una riga commentata. Se si rimuove il commento da questa riga, il messaggio pop-up continuerà a essere visualizzato finché non si inserisce la parola secret.

Il motivo per cui si verifica questo ciclo è perché si ritrae, con (retract, il Fatto (word catturato come ?f nella parte sinistra, LHS, della Regola user-did-not-say-secret. Ciò attiva il pattern (not (word ?)) in get-word-from-user che è l'assenza di un Fatto (word con un campo nella Memoria di Lavoro.

Questo esempio è un po' più complesso, ma fornisce un buon esempio di come sia possibile "convalidare" l'input dell'utente. Questo non è l'*unico* modo, sia chiaro. Si tratta, tuttavia, di un buon esempio di come il "main loop" di un programma possa essere realizzato utilizzando solo Regole e Fatti.

(clear)

(defrule get-word-from-user

;uncommenting this line will cause a prompt to popup

;until you enter "secret" in the textbox ;)

;(not (word ?))

=>

(print "Enter your word: ")

(bind ?word (read))

(println ?word)

(assert (word ?word)))

(defrule user-said-secret

(word secret)

=>

(println "You said the secret word! Great job"))

(defrule user-did-not-say-secret

?f <- (word ~secret)

=>

(retract ?f)

(println "Try again..."))

(run)

### Output:

Enter your word: Uno  
Try again...

# 14. Così tante parole

La funzione (readline) acquisirà l'input dell'utente e restituirà una stringa. Se l'input dell'utente contiene spazi tra le parole, possiamo utilizzare (explode$ per suddividere l'input in un "valore multicampo". Questo ci consente di definire pattern nelle nostre regole che corrisponderanno [match] alle singole parole inserito dall'utente.

Diamo un'occhiata alla Regola first-word-one. Questa Regola si attiverà quando la prima parola inserita dall'utente sarà "**one**" .Allo stesso modo, la regola last-word-last si attiverà quando l'ultima parola inserita è "**last**".

La regola anywhere verrà attivata se "**anywhere**" viene inserito in un punto qualsiasi del prompt. $? corrisponderà [match] a qualsiasi numero di parole. Ad esempio, vediamo nella Regola counter il pattern (exploded-words $?words). Ciò significa "abbina [match] un numero qualsiasi di valori e fai riferimento ad essi come un "valore multicampo" ?words. Proprio come ?, $? ci permette di dire "corrisponde [match] su un numero qualsiasi di campi" senza farvi riferimento nella parte destra (RHS).

A proposito della Regola counter: dare un'occhiata a (length$ ?words). Come intuito, (length$ restituirà il numero di valori nel valore multicampo passato ad esso. In questo caso, restituisce il numero di parole (a cui fa riferimento ?words) immesse dall'utente nel prompt.

(clear)

(defrule get-words-from-user

=>

(assert (words (readline))))

(defrule catcher

(words ?words)

=>

(println "You said: " ?words)

(assert (exploded-words (explode$ ?words))))

(defrule counter

(exploded-words $?words)

=>

(println "That's " (length$ ?words) " words total."))

(defrule silence

(exploded-words)

=>

(println "Ah, the strong silent type..."))

(defrule first-word-one

(exploded-words one $?words)

=>

(println "Your first word was 'one'"))

(defrule last-word-last

(exploded-words $?words last)

=>

(println "Your last word was 'last'"))

(defrule anywhere

(exploded-words $? anywhere $?)

=>

(println "At some point, you typed 'anywhere'"))

(run)

### Output:

You said: Uno  
That's 1 words total.

# 15. Soprattutto la rilevanza

Utilizzando (declare (salience nella nostra Regola, possiamo specificare la "priorità" di una Regola. Questo ci permette di dichiarare che **alcune Regole devono sempre essere eseguite prima di altre**. Viene "scolpito" l'ordine di esecuzione delle regole e l'algoritmo Rete si basa su di esso.

Per default, le regole hanno una rilevanza pari a 0. Le Regole con rilevanza maggiore di 0 verranno eseguite prima di esse. Allo stesso modo, le Regole con una rilevanza inferiore verranno eseguite dopo di esse. Secondo la [Basic Programming Guide](https://clipsrules.sourceforge.io/documentation/v640/bpg.pdf), questa funzionalità dovrebbe essere utilizzata con parsimonia:

Nonostante il gran numero di valori possibili, con una buona progettazione raramente sono necessari più di cinque valori di salienza (rilevanza) in un programma semplice e dieci valori di salienza in un programma complesso.

Nella maggior parte dei casi, un Motore di Regole dovrebbe essere indipendente dall'ordine. Ciò consente a CLIPS di svolgere il "lavoro pesante" di determinare quando eseguire le Regole. Questo è sottile, ma piuttosto potente. Con le Regole, non si "passano" i valori a una "funzione" che viene eseguita e restituisce un valore. Invece, l'algoritmo "attiva" Regole basate su fatti "corrispondenti" [match] nella Memoria di Lavoro. Ciò disaccoppia i dati memorizzati nel sistema (Fatti) dalla logica (Regole).

Questo disaccoppiamento può aiutare a esprimere il dominio che si sta modellando con meno riferimenti al "flusso di controllo" del software. Più il software è trasparente, più il sistema avvicina l'utente alla "verità" del dominio.

(clear)

(defrule third

(declare (salience 0))

?f <- (my-fact)

=>

(println "Last!")

(retract ?f))

(defrule never-runs

(declare (salience -1))

(my-fact)

=>

(println ":("))

(defrule first

(declare (salience 2))

(my-fact)

=>

(println "First!"))

(defrule second

(declare (salience 1))

(my-fact)

=>

(println "Second!"))

(assert (my-fact))

(run)

### Output:

First!  
Second!  
Last!

# 16. Qual è la funzione?

Le funzioni consentono di fare riferimento a blocchi di codice che si scrivono ripetutamente nel programma. Se stessimo scrivendo un libro, potremmo trovare più semplice "racchiudere" il testo pronunciato tra virgolette e un riferimento all'oratore. Facciamo lo stesso per gridare.

Possiamo anche memorizzare formule difficili da ricordare come Funzioni. Personalmente, ho difficoltà a ricordare la formula per convertire i Fahrenheit in Celsius. Con l'aiuto di deffunction, posso memorizzare questo calcolo e farvi riferimento facilmente.

Notare le funzioni (\*, (- e (/. Queste sono le operazioni matematiche di base: moltiplicazione, sottrazione e divisione. In CLIPS, scriviamo il problema di matematica 1 + 2 come (+ 1 2). Questo perché CLIPS utilizza una sintassi "LISP-like".

(clear)

(deffunction say (?words)

(println "You say, \"" ?words ".\""))

(say "Hello, world")

(deffunction shout (?words)

(println "\"" ?words "!\" you shout."))

(shout "Hello, world")'

(deffunction fahrenheit-to-celsius (?f)

(\* (- ?f 32) (/ 5 9) ))

(println "50°F is " (fahrenheit-to-celsius 50) "°C")

### Output:

You say, "Hello, world."  
"Hello, world!" you shout.  
50°F is 10.0°C

# 17. Templatizzami

I template vengono utilizzati per definire una struttura formale per i Fatti. Utilizzando (deftemplate person nell'esempio, descriviamo la struttura di un Fatto (person. Abbiamo scritto i Fatti come elenchi di valori nei capitoli precedenti. Potremmo descrivere questi Fatti come "campi" ordinati. (person Sally "Hi") descriverebbe un Fatto (person con i valori Sally nel suo secondo campo e "Hi" nel terzo campo.

Con i Template, non siamo più obbligati a memorizzare le informazioni delle persone in un ordine particolare. Invece, memorizziamo i valori in nomi di "Slot". Ad esempio, possiamo scegliere di definire il nostro Fatto come (person (greeting "Hi") (name Sally)), con i saluti prima. Possiamo anche definire valori di "default" per questi nomi di slot e ometterli nei Fatti asseriti.

L'uso dei Template ci dà una certa flessibilità nel modo in cui definiamo i nostri Fatti e nel modo in cui scriviamo i pattern di corrispondenza nella sinistra delle nostre Regole (LHS). Osserviamo la Regola people-greet-each-other. Il nostro secondo pattern corrisponde [match] a "una persona il cui nome non è ?name1". Non abbiamo bisogno di includere un riferimento allo Slot (greeting di questo Fatto in questa Regola, quindi possiamo ometterlo.

Si noterà anche che definiamo un valore di default per (greeting nella definizione del Template. Esistono molti altri modi per descrivere uno Slot oltre a default. Leggere la [Basic Programming Guide](https://clipsrules.sourceforge.io/documentation/v640/bpg.pdf) nella sezione intitolata "Deftemplate Construct" per sapere cosa sono.

(clear)

(deftemplate person

(slot name)

(slot greeting (default Hello)))

(assert

(person (name Greg))

(person (name Sally) (greeting Hi))

(person (name Sam) (greeting Howdy)))

(deffunction greet (?name1 ?name2 ?greeting)

(println ?name1 " says \"" ?greeting "\" to " ?name2 "."))

(defrule people-greet-each-other

(person (name ?name1) (greeting ?greeting1))

(person (name ?name2&~?name1))

=>

(greet ?name1 ?name2 ?greeting1))

(run)

### Output:

Sam says "Howdy" to Sally.  
Sam says "Howdy" to Greg.  
Greg says "Hello" to Sam.  
Sally says "Hi" to Sam.  
Sally says "Hi" to Greg.  
Greg says "Hello" to Sally.

# 18. Chiaramente un reset

Parliamo di (clear) e di (reset). Conosciamo (clear) che rende il nostro environment come nuovo. Niente più Regole, niente più Fatti.

(reset) fa un po' meno. Ritira tutti i Fatti nel sistema e le attivazioni delle Regole lasciando le Regole stesse definite. In questo modo, si può utilizzare (reset) per riportare il motore a uno "stato" di default noto prima di eseguirlo nuovamente con un nuovo input.

Cosa succede se abbiamo bisogno che alcuni Fatti vengano asseriti per tornare allo stato di "default"? Introduciamo (deffacts. Quando si usa (reset), i Fatti definiti nei propri(deffacts verranno asseriti. Consideriamo l'esempio in questo capitolo e il suo output. Nella prima esecuzione, asseriamo con (assert altri 2 Fatti frutti. Di conseguenza, vediamo altre due persone mangiare il loro frutto preferito.

(clear)

(deffacts fruits

(fruit apple 1)

(fruit banana 1)

(fruit orange 1))

(deftemplate person

(slot name)

(slot favorite-fruit))

(deffacts people

(person (name Tom) (favorite-fruit apple))

(person (name Jenna) (favorite-fruit orange))

(person (name Rian) (favorite-fruit apple)))

(defrule people-eat-favorite-fruits

(person (name ?name) (favorite-fruit ?fruit))

?f <- (fruit ?fruit ?id)

=>

(println ?name " eats " ?fruit)

(retract ?f))

(defrule persons-favorite-fruit-missing

(person (name ?name) (favorite-fruit ?fruit))

(not (fruit ?fruit ?))

=>

(format t "There's no more of %s's favorite fruit %s!%n" ?name ?fruit))

(reset)

(assert (fruit apple 2) (fruit orange 2))

(println "Run 1:")

(println "------")

(run)

(println " ")

(reset)

(println "Run 2:")

(println "------")

(run)

### Output:

Run 1:  
------  
Jenna eats orange  
Tom eats apple  
Rian eats apple  
There's no more of Tom's favorite fruit apple!  
There's no more of Rian's favorite fruit apple!  
Jenna eats orange  
There's no more of Jenna's favorite fruit orange!  
   
Run 2:  
------  
Rian eats apple  
There's no more of Tom's favorite fruit apple!  
There's no more of Rian's favorite fruit apple!  
Jenna eats orange  
There's no more of Jenna's favorite fruit orange!

# 19. Chi guarda le statistiche?

(clear)

(watch statistics)

(defrule fibonacci

?fact <- (fib ?fib ?lastfib ?i&:(> ?i 1))

=>

(print ?fib " ")

(retract ?fact)

(assert (fib (+ ?fib ?lastfib) ?fib (- ?i 1))))

(defrule last-fibonacci-number

?fact <- (fib ?fib ? 1)

=>

(println ?fib)

(retract ?fact))

(assert (fib 1 0 10))

(run)

(unwatch statistics)

### Output:

1 1 2 3 5 8 13 21 34 55  
10 rules fired  
1 mean number of facts (1 maximum).  
0 mean number of instances (0 maximum).  
1 mean number of activations (1 maximum).

# 20. Regile: Attive!

(clear)

(watch activations)

(defrule b-exists-and-a-less-than-1

(b ?b)

(a ?a&:(< ?a 1))

=>

(println "there is b = " ?b ", AND a " ?a " is less than 1"))

(defrule a-less-than-1

(a ?a&:(< ?a 1))

=>

(println "a " ?a " is less than 1"))

(println " Step 1 ")

(println "==========")

(println "--- assert")

(assert (a 2))

(println "------ run")

(run)

(println " ")

(println " Step 2 ")

(println "==========")

(println "--- assert")

(assert (a 0))

(println "------ run")

(run)

(println " ")

(println " Step 3 ")

(println "==========")

(println "--- assert")

(assert (b foo) (a -1) (a 3))

(println "------ run")

(run)

(unwatch activations)

### Output:

Step 1  
==========  
--- assert  
------ run  
   
Step 2  
==========  
--- assert  
==> Activation 0 a-less-than-1: f-2  
------ run  
a 0 is less than 1  
   
Step 3  
==========  
--- assert  
==> Activation 0 b-exists-and-a-less-than-1: f-3,f-2  
==> Activation 0 a-less-than-1: f-4  
==> Activation 0 b-exists-and-a-less-than-1: f-3,f-4  
------ run  
there is b = foo, AND a -1 is less than 1  
a -1 is less than 1  
there is b = foo, AND a 0 is less than 1

# 21. Funzioni per l'osservazione

(clear)

(watch deffunctions)

(deffunction less-than-1 (?a)

(println "calculating whether " ?a " is less than 1...")

(< ?a 1))

(defrule b-exists-and-a-less-than-1

(b ?b)

(a ?a&:(less-than-1 ?a))

=>

(println "there is b = " ?b ", AND a " ?a " is less than 1"))

(defrule a-less-than-1

(a ?a&:(less-than-1 ?a))

=>

(println "a " ?a " is less than 1"))

(println " Step 1 ")

(println "==========")

(println "--- assert")

(assert (a 2))

(println "------ run")

(run)

(println " ")

(println " Step 2 ")

(println "==========")

(println "--- assert")

(assert (a 0))

(println "------ run")

(run)

(println " ")

(println " Step 3 ")

(println "==========")

(println "--- assert")

(assert (b foo) (a -1) (a 3))

(println "------ run")

(run)

(unwatch deffunctions)

### Output:

Step 1  
==========  
--- assert  
DFN >> less-than-1 ED:1 (2)  
calculating whether 2 is less than 1...  
DFN << less-than-1 ED:1 (2)  
------ run  
   
Step 2  
==========  
--- assert  
DFN >> less-than-1 ED:1 (0)  
calculating whether 0 is less than 1...  
DFN << less-than-1 ED:1 (0)  
------ run  
a 0 is less than 1  
   
Step 3  
==========  
--- assert  
DFN >> less-than-1 ED:1 (-1)  
calculating whether -1 is less than 1...  
DFN << less-than-1 ED:1 (-1)  
DFN >> less-than-1 ED:1 (3)  
calculating whether 3 is less than 1...  
DFN << less-than-1 ED:1 (3)  
------ run  
there is b = foo, AND a -1 is less than 1  
a -1 is less than 1  
there is b = foo, AND a 0 is less than 1

### 22. ???

Per ora è tutto! Grazie per aver seguito il tutorial fino a questo punto. Col tempo aggiungerà dell'altro, quindi ci si assicuri di aggiungere questa pagina ai segnalibri per poi ricontrollare :).

L'intera pagina è scritta da zero laddove possibile. Nessun framework, solo Javascript, CSS e HTML procedurali vecchio stile. Mi sono molto ispirato al [Tour of Go](https://go.dev/tour/welcome/1). Ho trovato molto utile non dover installare nulla quando ho imparato a usare Go anni fa. Ho usato [emscripten](https://emscripten.org/) per compilare [CLIPS](https://www.clipsrules.net/) in un binario WASM. Ho scritto da zero l'editor di codice + l'evidenziatore della sintassi con frammenti di codice estratti da [Stackoverflow](https://stackoverflow.com/). Ammetto che ne sono un po' fiero :).

Il mio obiettivo con questo tutorial è quello di divulgare CLIPS. Mi piacerebbe vederlo utilizzato in applicazioni più quotidiane perché è un modo molto potente di esprimere una logica aziendale complessa con meno righe di codice.

(clear)

(run)

### Output:

# Uno

## Due

### Tre

#### Quattro

##### Cinque

###### Sei

Code

Code No Border

Code inline