CLIPS 6.4
Guida Utente

di Joseph C. Giarratano, Ph.D.

Traduzione Italiana: 23/5/2024

Sommario

Readme

Il primo passo sulla strada verso la saggezza è l'ammissione dell'ignoranza.

Il secondo passo è capire che non lo si deve dire al mondo.

Questa sezione era precedentemente chiamata *Prefazione*, ma poiché nessuno la leggeva, l'ho rinominata con qualcosa di più convenzionale a cui gli utenti di computer sono condizionati a obbedire. Un altro suggerimento era quello di chiamare questa sezione *Da Non Leggere*, ma poiché oggi la gente crede a tutto ciò che legge, avevo paura che davvero non lo leggessero.

Lo scopo di una *Prefazione*, oops, scusatemi, un *Readme*, è quello di fornire **meta-conoscenza** sulla conoscenza contenuta in un libro. Il termine *meta-conoscenza* significa conoscenza sulla conoscenza. Quindi questa descrizione del *Readme* è in realtà meta-meta-conoscenza. Se a questo punto sei confuso o incuriosito, prosegui e leggi comunque questo libro perché ho bisogno di tutti i lettori che riesco a trovare.

Cos'è CLIPS?

CLIPS è uno strumento di sistema esperto originariamente sviluppato dal Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center. Sin dalla sua prima versione nel 1986, CLIPS è stato sottoposto a continui perfezionamenti e miglioramenti. Ora è utilizzato da migliaia di persone in tutto il mondo.

CLIPS è progettato per facilitare lo sviluppo di software per modellare la conoscenza o l'esperienza umana.

Esistono tre modi per rappresentare la conoscenza in CLIPS:

- *Regole*, destinate principalmente alla conoscenza euristica basata sull'esperienza.
- *Deffunctions* e *funzioni generiche*, destinate principalmente alla conoscenza procedurale.
- *Programmazione object-oriented*, anch'essa destinata principalmente alla conoscenza procedurale. Sono supportate le cinque caratteristiche generalmente accettate della programmazione orientata agli oggetti: classi, gestori di messaggi, astrazione, incapsulamento, ereditarietà e polimorfismo. Le regole possono modellare la corrispondenza tra oggetti e fatti.

Si può sviluppare software utilizzando solo regole, solo oggetti o una combinazione di oggetti e regole.

CLIPS è stato progettato anche per l'integrazione con altri linguaggi come C e Java. Infatti, CLIPS è l'acronimo di C Language Integrated Production System. Anche le regole e gli oggetti formano un sistema integrato poiché le regole possono corrispondere a fatti e oggetti. Oltre ad essere utilizzato come tool a sé stante, CLIPS può essere chiamato da un linguaggio procedurale, eseguire la sua funzione per poi restituire il controllo al programma chiamante. Allo stesso modo, il codice procedurale può essere definito come funzioni esterne e chiamato da CLIPS. Quando il codice esterno finisce l'esecuzione, il controllo ritorna a CLIPS.

Se si ha già familiarità con la programmazione orientata agli oggetti in altri linguaggi come Smalltalk, C++, Objective C o Java, saranno noti i vantaggi degli oggetti nello sviluppo di software. Se non si conosce la programmazione orientata agli oggetti, si vedrà che CLIPS è uno strumento eccellente per apprendere questo nuovo concetto di sviluppo software.

Di cosa tratta questo libro

La *CLIPS User's Guide* è un tutorial introduttivo sulle funzionalità di base di CLIPS. *Non* è da intendersi come una discussione esaustiva dell'intero tool. Il volume allegato a questo libro è il *CLIPS Reference Manual*, che fornisce una discussione completa ed esauriente di tutti gli argomenti di questo libro e molto altro ancora.

Chi dovrebbe leggere questo libro

Lo scopo della *CLIPS User's Guide* è fornire un'introduzione elementare e di facile lettura ai sistemi esperti per persone con poca o nessuna esperienza con i sistemi esperti.

La *CLIPS User's Guide* può essere utilizzata in classe o per l'autoapprendimento. L'unico prerequisito è che si abbia una conoscenza base di programmazione in un linguaggio di alto livello come Java, Ada, FORTRAN, C (OK, BASIC se non altro, ma non lo ammetteremo in pubblico e sconfesseremo questa affermazione se verrà chiesto).

Come utilizzare questo libro

La "CLIPS User's Guide" è progettata per coloro che desiderano una rapida introduzione alla programmazione di sistemi esperti in modo pratico. Gli esempi sono di carattere molto generale. Inoltre, poiché imparare un nuovo linguaggio può essere un'esperienza frustrante, la scrittura è in uno stile leggero e divertente (spero) rispetto ai libri di testo universitari seri, massicci e intimidatori. Si spera che l'umorismo non offenda nessuno col senso dell'umorismo.

Per ottenere il massimo beneficio, si dovrebbero digitare i programmi di esempio nel testo mentre si legge il libro. Digitando gli esempi si vedrà come dovrebbero funzionare i programmi e quali messaggi di errore compaiono se si commettono errori. L'output degli esempi viene mostrato o descritto dopo ogni esempio. Infine, si dovrebbe leggere il materiale corrispondente nel Manuale di riferimento di CLIPS (CLIPS Reference Manual) man mano che si esamina ogni capitolo della "CLIPS User's Guide".

Come qualsiasi altro linguaggio di programmazione, si imparerà a programmare in CLIPS solo scrivendoci dei programmi. Per imparare veramente la programmazione di sistemi esperti, si dovrebbe scegliere un problema di interesse e scriverlo in CLIPS.

Ringraziamenti

Ho gradito molto i consigli e le recensioni di questo libro da parte di molte persone. Grazie a Gary Riley, Chris Culbert, Brian Dantes, Bryan Dulock, Steven Lewis, Ann Baker, Steve Mueller, Stephen Baudendistel, Yen Huynh, Ted Leibfried, Robert Allen, Jim Wescott, Marsha Renals, Pratibha Boloor, Terry Feagin e Jack Aldridge. Un ringraziamento speciale a Bob Savely per aver supportato lo sviluppo di CLIPS.

Nota dell'editore

Sono state apportate modifiche minori al documento originale del Dr. Giarratano per riflettere i cambiamenti nelle versioni più recenti di CLIPS — Gary Riley.

Capitolo 1

Solo Fatti

Se si ignorano i fatti, non ci si preoccuperà mai di sbagliare

Questo capitolo presenta i concetti base di un sistema esperto. Si vedrà come inserire e rimuovere fatti in CLIPS.

Introduzione

CLIPS è un tipo di linguaggio progettato per scrivere applicazioni chiamate **sistemi esperti**. Un sistema esperto è un programma specificamente destinato a modellare la competenza o la conoscenza umana. Al contrario, normali i programmi come i quelli per la gestione delle buste paga, i word processor, i fogli di calcolo, i giochi per computer e così via, non sono destinati a includere competenze o conoscenze umane. (Una definizione di esperto è qualcuno che si trova a più di 50 miglia da casa e porta con sé una valigetta).

CLIPS è chiamato **tool** di sistema esperto perché è un ambiente completo per lo sviluppo di sistemi esperti che include funzionalità come un editor integrato e uno strumento di debug. La parola **shell** è riservata a quella porzione di CLIPS che esegue **inferenze** o ragionamenti. La shell di CLIPS fornisce gli elementi base di un sistema esperto:

1. **fact-list** e **instance-list:** Memoria globale per i dati
2. **knowledge-base**: Contiene tutte le regole, la **rule-base**
3. **inference engine**: Controlla l'esecuzione complessiva delle regole

Un programma scritto in CLIPS può essere costituito da regole, fatti e oggetti. Il motore di inferenza decide quali regole devono essere eseguite e quando. Un sistema esperto basato su regole scritto in CLIPS è un programma basato sui dati in cui i fatti e, volendo, gli oggetti, sono i dati che stimolano l'esecuzione tramite il motore di inferenza.

Questo è un esempio di come CLIPS differisce dai linguaggi procedurali come Java, Ada, BASIC, FORTRAN e C. Nei linguaggi procedurali, l'esecuzione può procedere senza dati. Cioè, le istruzioni in quei linguaggi sono sufficienti per provocare l'esecuzione. Ad esempio, un'istruzione come PRINT 2 + 2 potrebbe essere immediatamente eseguita in BASIC. Questa è un'istruzione completa che non richiede alcun dato aggiuntivo per essere eseguito. Tuttavia, in CLIPS, i dati sono necessari per provocare l'esecuzione delle regole.

Inizialmente, CLIPS aveva la capacità di rappresentare solo regole e fatti. Tuttavia, i miglioramenti della versione 6.0 consentono alle regole di abbinare oggetti e fatti. Inoltre, gli oggetti possono essere utilizzati *senza* regole inviando messaggi e quindi il motore di inferenza non è più necessario se si utilizzano solo oggetti. Nei capitoli dall'1 al 7 discuteremo i fatti e le regole di CLIPS. Le caratteristiche degli oggetti di CLIPS sono trattate nei capitoli dall'8 al 12.

L'inizio e la fine

Per avviare CLIPS, basta inserire il comando di esecuzione appropriato per il proprio sistema. Si dovrebbe vedere il prompt CLIPS apparire come segue:

```
CLIPS>
```

A questo punto si può iniziare inserendo i **comandi** direttamente in CLIPS. La modalità in cui si inseriscono i comandi diretti è detta **top-level**. Se si dispone di una versione con interfaccia grafica (GUI) di CLIPS, si possono anche **selezionare** alcuni comandi utilizzando il mouse o i tasti freccia anziché digitarli. Fare riferimento alla *CLIPS Interfaces Guide* per una discussione sui comandi supportati dalle varie GUI di CLIPS. Per semplicità e uniformità in questo libro, assumeremo che i comandi vengano digitati.

La modalità normale per uscire da CLIPS è con il comando **exit**. Basta digitare

```
(exit)
```

in risposta al prompt CLIPS, e poi premere il tasto Enter.

Creare una lista

Come con altri linguaggi di programmazione, CLIPS riconosce determinate parole chiave. Ad esempio, per inserire dati nella "fact-list", si può usare il comando assert.
Come esempio di *assert*, inserire quanto segue subito dopo il prompt CLIPS come mostrato:

```
CLIPS> (assert (duck))
```

Qui il comando assert prende (duck) come argomento. Assicurarsi di premere sempre il tasto Enter per inviare la riga a CLIPS. Si vedrà la seguente risposta

```
<Fact-1>
```

indicando che CLIPS ha memorizzato il fatto "duck" nell'elenco dei fatti e gli ha assegnato l'identificatore 1. Le **parentesi angolari** vengono usate come delimitatore in CLIPS per racchiudere il nome di un elemento. CLIPS assegnerà automaticamente un nome ai fatti utilizzando un numero crescente in sequenza ed elencherà l'indice dei fatti più alto quando vengono affermati uno o più fatti.

Notare che il comando (assert) e il suo argomento (duck) sono racchiusi tra parentesi. Come molti altri linguaggi di sistemi esperti, CLIPS ha una sintassi simile a LISP utilizzando parentesi come delimitatori. Sebbene CLIPS non sia scritto in LISP, lo stile di LISP ha influenzato lo sviluppo di CLIPS.

E verificarla

Supponiamo che si voglia vedere cosa c'è nella "fact-list". Se la versione di CLIPS supporta una GUI, si può semplicemente selezionare il comando appropriato dal menù. In alternativa, si possono immettere i comandi dalla tastiera. Di seguito descriveremo i comandi da tastiera poiché le selezioni della finestra sono autoesplicative.

Il comando da tastiera per vedere i fatti è col comando **facts**. Immettere "(facts)" in risposta al prompt CLIPS e CLIPS risponderà con un elenco di fatti nella "fact-list". Assicurarsi di racchiudere il comando tra parentesi altrimenti CLIPS non lo accetterà. Il risultato del comando "(facts)" in questo esempio dovrebbe essere

```
CLIPS> (facts)
f-1     (duck)
For a total of 1 fact.
CLIPS>
```

Il termine *f-1* è l'**identificatore del fatto** assegnato al fatto "(duck)" da CLIPS. A ogni fatto inserito nella "fact-list"' viene assegnato un identificatore univoco dei fatti che inizia con la lettera "f" seguita da un numero intero chiamato **fact-index**. All'avvio di CLIPS e dopo alcuni comandi come **clear** e **reset** (di cui parleremo più dettagliatamente in seguito), l'indice dei fatti verrà impostato su uno, e poi incrementato di uno man mano che viene affermato ogni nuovo fatto.

Cosa succede se si inserisce una seconda "duck" nell'elenco dei fatti? Proviamolo e vediamo. Si dichiara un nuovo (duck), quindi si impartisce un comando (facts) come segue

```
CLIPS> (assert (duck))
<Fact-1>
CLIPS> (facts)
f-1     (duck)
For a total of 1 fact.
CLIPS>
```

Il messaggio <Fact-1> viene restituito da CLIPS per indicare che il fatto esiste già. Si vedrà solo l'originale “f-1 (duck)”. Ciò dimostra che CLIPS non accetterà un'immissione duplicata di un fatto. Tuttavia, esiste un comando di override, **set-fact-duplication**, che consentirà l'immissione di fatti duplicati.
Ovviamente si possono inserire altri fatti diversi. Ad esempio, asserire un fatto (quack) e poi richiamare comando (facts). Si vedrà

```
CLIPS> (assert (quack))
<Fact-2>
CLIPS> (facts)
f-1     (duck)
f-2     (quack)
For a total of 2 facts.
CLIPS>
```

Notare che il fatto (quack) è ora nell'elenco dei fatti.

I fatti possono essere rimossi o **retracted** (ritrattati). Quando un fatto viene ritirato, gli altri fatti non cambiano i loro indici, e quindi potrebbero esserci indici di fatti “mancanti”. Per analogia, quando un giocatore di football lascia una squadra e non viene sostituito, i numeri di maglia degli altri giocatori non vengono tutti modificati a causa del numero mancante (a meno che non odino davvero quel ragazzo e vogliano dimenticare che abbia mai giocato per loro).

Chiarire i fatti

Il comando **clear** rimuove tutti i fatti dalla memoria, come mostrato di seguito.

```
CLIPS> (facts)
f-1     (duck)
f-2     (quack)
For a total of 2 facts.
CLIPS> (clear)
CLIPS> (facts)
CLIPS>
```

Il comando (clear) ripristina essenzialmente CLIPS al suo stato di avvio originale. Cancella la memoria di CLIPS e reimposta l'identificatore dei fatti a uno. Per vederlo, si asserisce (animal-is duck), quindi si controlla la "fact-list".

```
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (facts)
f-1     (animal-is duck)
For a total of 1 fact.
CLIPS>
```

Si noti che (animal-is duck) ha come identificatore f-1 perché il comando (clear) reimposta gli identificatori dei fatti. Il comando (clear) in realtà fa molto più che rimuovere semplicemente i fatti. Oltre a rimuovere tutti i fatti, (clear) rimuove anche tutte le regole, come si vedrà nel prossimo capitolo.

Campi sensibili e slurping

Si dice che un fatto come (duck) o (quack) consista di un singolo **field** (campo). Un *field* è un segnaposto (con o senza nome) a cui può essere associato un valore. Per fare una semplice analogia, si può pensare a un field come a una cornice. La cornice può contenere un'immagine, forse l'immagine del "duck" (papera) (per quelli di voi che sono curiosi di sapere come appare l'immagine di un "quack", potrebbe essere (1) una foto di una traccia dell'oscilloscopio di un'anatra che dice "quack", dove il segnale in ingresso proviene da un microfono, o (2) per quelli di voi che sono più inclini alla scienza, una trasformata veloce di Fourier del segnale "quack", o (3) un tele-venditore che vende una cura miracolosa per rughe, perdita di peso, ecc.). I segnaposto con nome vengono utilizzati solo con i **deftemplates**, descritti più dettagliatamente nel capitolo 5.

Il fatto (duck) ha un singolo segnaposto senza nome per il valore duck. Questo è un esempio di un fatto **single-field**. Un field è un segnaposto per un valore. Come analogia ai field, si pensi ai piatti (field) per contenere il cibo (valori).

L'**ordine** dei field senza nome è significativo. Ad esempio, se un fatto fosse definito

```
(Brian duck)
```

e interpretato secondo una regola secondo cui il cacciatore Brian ha sparato a un'anatra, quindi il fatto

```
(duck Brian)
```

significherebbe che il cacciatore ha sparato a Brian. Al contrario, l’ordine dei campi con nome non è significativo, come si vedrà più avanti con deftemplate.

In realtà, è buona ingegneria del software iniziare il fatto con una relazione che descriva i field. Un fatto migliore sarebbe

```
(hunter-game duck Brian)
```

per implicare che il primo campo sia il cacciatore e il secondo campo sia la selvaggina.

Sono ora necessarie alcune definizioni. Una **list** è un gruppo di elementi senza ordine implicito. Dire che una lista è **ordered** (ordinata) significa che la posizione nella lista è significativa. Un **multifield** è una sequenza di campi, ognuno dei quali può avere un valore. Gli esempi di (duck Brian) e (Brian duck) sono fatti multifield. Se un campo non ha

valore, per un campo vuoto può essere utilizzato come segnaposto il simbolo speciale **nil**, che significa “niente”. Per esempio,

```
(duck nil)
```

significherebbe che oggi l'assassino di anatre (duck) non ha vinto alcun trofeo.

Notare che *nil* è necessario per indicare un segnaposto, anche se non ha valore. Ad esempio, si pensi a un campo come a una casella di posta. C’è una grande differenza tra una casella di posta vuota e nessuna casella di posta. Senza il *nil*, il fatto diventa un fatto single-field (duck). Se una regola dipende da due campi, non funzionerà con un solo campo, come si vedrà in seguito.

Sono disponibili diversi **tipi** di campi: **float**, **integer**, **symbol**, **string**, **external-address**, **fact-address**, **instance-name** e **instance-address**. Il tipo di ciascun campo è determinato dal tipo di **valore** memorizzato nel campo. In un campo senza nome, il tipo è determinato *implicitamente* dal tipo inserito nel campo. Nei deftemplates, si può *esplicitamente* dichiarare il tipo di valore che un campo può contenere. L'uso di tipi espliciti rafforza i concetti di **ingegneria del software**, che è una disciplina di programmazione per produrre software di qualità.

Un **symbol** è un tipo di campo che inizia con un carattere ASCII stampabile ed è seguito facoltativamente da zero o più caratteri stampabili. I campi sono solitamente **delimitati** o racchiusi da uno o più spazi o parentesi. Per esempio,

```
(duck-shot Brian Gary Rey)
```

Tuttavia, questo potrebbe essere un deftemplate lecito se "shot" è definito come il nome di un campo, mentre "Brian Gary Rey" sono i valori associati al campo citato.
CLIPS è case-sensitive. Inoltre, alcuni caratteri hanno un significato speciale per CLIPS.

```
"  (  )  &  |  <  ~  ;  ?  $
```

“&”, “|” e “~” non possono essere utilizzati come simboli autonomi o come *qualsiasi* parte di un simbolo.

Alcuni caratteri fungono da delimitatori *terminando* un simbolo. I seguenti caratteri fungono da delimitatori per i simboli.

- qualsiasi carattere ASCII non stampabile, inclusi spazi, ritorni a capo, tabulazioni e avanzamenti di riga
- virgolette doppie, **"**
- parentesi di apertura e chiusura, **()**
- e commerciale, **&**
- barra verticale, **|**
- minore di, **<**. Notare che questo potrebbe essere il *primo* carattere di un simbolo
- tilde, **~**
- punto e virgola, **;** indica l'inizio di un commento, un ritorno a capo lo termina
- ? e $? potrebbe non iniziare un simbolo ma potrebbe trovarsi al suo interno

Il punto e virgola funge da inizio di un commento in CLIPS. Se si prova a inserire un punto e virgola, CLIPS penserà che si sta inserendo un commento e aspetterà che lo si finisca. Se si inserisce accidentalmente un punto e virgola nel livello superiore [top-level], digitare semplicemente una parentesi di chiusura e un ritorno a capo. CLIPS risponderà con un messaggio di errore e riapparirà il prompt CLIPS (Questa è una delle poche occasioni approvate nella vita in cui è necessario fare qualcosa di sbagliato per ottenere qualcosa di giusto).

Leggendo questo manuale, si impareranno i significati speciali dei caratteri di cui sopra. Ad eccezione di “&”, “|” e “~”, è possibile utilizzare gli altri come descritto. Tuttavia, potrebbe creare confusione a qualcuno che legge il programma e cerca di capire cosa sta facendo il programma. In generale, è meglio evitare di utilizzare questi caratteri nei simboli a meno che non si abbia una buona ragione per farlo.
Di seguito sono riportati esempi di simboli.

```
duck
duck1
duck_soup
```

```
duck-soup
duck1-1_soup-soup
d!?#%^
```

Il secondo tipo di campo è la **string**. Una stringa deve iniziare e terminare con virgolette doppie. Le virgolette doppie fanno parte del campo. Tra le virgolette doppie possono essere presenti zero o più caratteri di qualsiasi tipo. Seguono alcuni esempi di stringhe.

```
"duck"
"duck1"
"duck/soup"
"duck soup"
"duck soup is good!!!"
```

Il terzo e il quarto tipo di campo sono i **campi numerici**. Un campo che rappresenta un numero può essere di tipo **intero** o in **virgola mobile**. Un tipo a virgola mobile viene comunemente chiamato semplicemente **float.**

Tutti i numeri in CLIPS sono trattati come numeri interi “long long” o float a **doppia precisione**. I numeri senza punto decimale vengono trattati come numeri interi a meno che non siano esterni all'intervallo dei numeri interi. L'intervallo dipende dalla macchina dal numero di bit, N, utilizzati per rappresentare il numero intero come segue.

$$-2^{N-1}$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$2^{N-1} - 1$$

Per gli interi “long long” a 64 bit, ciò corrisponde a un intervallo di numeri

```
- 9.223.372.036.854.775.808
.
.
.
9.223.372.036.854.775.807
```

Come alcuni esempi di numeri, asserire i seguenti dati in cui l'ultimo numero è in notazione esponenziale e utilizza la "e" o la "E" per la potenza di dieci.

```
CLIPS> (clear)
CLIPS> (facts)
CLIPS> (assert (number 1))
<Fact-1>
CLIPS> (assert (x 1.5))
<Fact-2>
CLIPS> (assert (y -1))
<Fact-3>
CLIPS> (assert (z 65))
<Fact-4>
CLIPS> (assert (distance 3.5e5))
<Fact-5>
CLIPS> (assert (coordinates 1 2 3))
<Fact-6>
CLIPS> (assert (coordinates 1 3 2))
<Fact-7>
CLIPS> (facts)
f-1     (number 1)
f-2     (x 1.5)
f-3     (y -1)
f-4     (z 65)
f-5     (distance 350000.0)
f-6     (coordinates 1 2 3)
f-7     (coordinates 1 3 2)
For a total of 7 facts.
CLIPS>
```

Come si può vedere, CLIPS stampa il numero immesso in notazione esponenziale come 350000,0 perché converte dal formato potenza di dieci a virgola mobile se il numero è sufficientemente piccolo.

Notare che ogni fatto deve iniziare con un simbolo come “numero”, “x”, “y”, ecc. Prima della versione 6.0 di CLIPS era possibile inserire come dato di fatto solo un numero. Tuttavia, ora è richiesto un simbolo come primo campo. Inoltre, alcune parole riservate utilizzate da CLIPS non possono essere utilizzate come primo campo, ma possono essere utilizzate per altri campi. Ad esempio, la parola riservata *not* viene utilizzata per indicare un pattern negato e non può essere utilizzata come primo campo di un fatto.

Un **fact** è costituito da uno o più campi racchiusi tra parentesi destre e sinistre corrispondenti. Per semplicità discuteremo solo i fatti nei primi sette capitoli, ma la maggior parte della discussione sul pattern match si applica anche agli oggetti. Le eccezioni sono alcune funzioni come assert e retract che si applicano solo ai fatti, non agli oggetti. I modi corrispondenti di gestire gli oggetti sono discussi nei capitoli 8–12.

Un fatto può essere **ordinato** o **non ordinato**. Tutti gli esempi visti finora sono ordinati perché l'ordine dei campi fa la differenza. Ad esempio, si noti che CLIPS li considera come fatti separati sebbene in ciascuno siano utilizzati gli stessi valori "1", "2" e "3".

```
f-6     (coordinates 1 2 3)
f-7     (coordinates 1 3 2)
```

I fatti ordinati *devono* utilizzare la posizione del campo per definire i dati. Ad esempio, il fatto ordinato (duck Brian) ha due campi e così anche (Brian duck). Tuttavia, questi vengono considerati da CLIPS come due fatti separati poiché l'ordine dei valori dei campi è diverso. Al contrario, il fatto (duck-Brian) ha un solo campo a causa del "-" che concatena i due valori.

I fatti deftemplate, descritti più dettagliatamente in seguito, non sono ordinati perché utilizzano campi denominati per definire i dati. Questo è analogo all'uso delle strutture in C e in altri linguaggi.

Più campi normalmente sono separati da un **white space** costituito da uno o più spazi, tabulazioni, ritorni a capo (CR) o a capo (LF). Per esempio, inserire i seguenti esempi come mostrato e si vedrà che ogni fatto memorizzato è lo stesso.

```
CLIPS> (clear)
CLIPS> (assert (The duck says "Quack"))
<Fact-1>
CLIPS> (facts)
f-1     (The duck says "Quack")
For a total of 1 fact.
CLIPS> (clear)
CLIPS> (assert  (The     duck      says     "Quack"  ))
<Fact-1>
CLIPS> (facts)
f-1     (The duck says "Quack")
For a total of 1 fact.
CLIPS>
```

I ritorni a capo (CR) sono utilizzabili anche per migliorare la leggibilità. Nell'esempio seguente, viene digitato un ritorno a capo dopo ogni campo e il fatto affermato è lo stesso di prima quando il fatto veniva inserito su una riga.

```
CLIPS> (clear)
CLIPS> (assert (The
duck
says
"Quack"))
<Fact-1>
CLIPS> (facts)
f-1     (The duck says "Quack")
For a total of 1 fact.
CLIPS>
```

Tuttavia, fare attenzione se si inserisce un ritorno a capo all'interno di una stringa, come mostra l'esempio seguente.

```
CLIPS> (assert (The
duck
says
"Quack
"))
<Fact-2>
CLIPS> (facts)
f-1     (The duck says "Quack")
f-2     (The duck says "Quack
")
For a total of 2 facts.
CLIPS>
```

Come si vede, il ritorno a capo racchiuso tra virgolette doppie è stato restituito con la stringa per inserire le virgolette doppie di chiusura nella riga successiva. Questo è importante perché CLIPS considera il fatto f-1 distinto dal fatto f-2.

Si noti inoltre che CLIPS ha conservato le lettere maiuscole e minuscole nei campi del fatto. Cioè, la “T” di “The” e la “Q” di “Quack” sono maiuscole. Si dice che CLIPS sia **casesensitive** perché distingue tra lettere maiuscole e minuscole. Ad esempio, asserire i fatti (duck) e (Duck) e poi eseguire un comando (facts). Si vedrà che CLIPS consente di affermare (duck) e (Duck) come fatti diversi perché CLIPS distingue tra maiuscole e minuscole.

L'esempio seguente è un caso più realistico in cui i ritorni a capo vengono utilizzati per migliorare la leggibilità di un elenco. Per capirlo, asserire il fatto seguente in cui i ritorni a capo e gli spazi vengono utilizzati per inserire i campi nei punti appropriati su righe diverse. I trattini o i segni meno vengono utilizzati intenzionalmente per creare singoli campi, quindi CLIPS tratterà elementi come "fudge sauce" come un singolo campo.

```
CLIPS> (clear)
CLIPS>
(assert (grocery-list
ice-cream
cookies
candy
fudge-sauce))
<Fact-1>
CLIPS> (facts)
f-1     (grocery-list ice-cream cookies candy fudge-sauce)
For a total of 1 fact..
CLIPS>
```

Si nota che CLIPS ha sostituito i ritorni a capo e le tabulazioni con spazi singoli. Sebbene l'uso del "white space" per separare i fatti sia conveniente per una persona che legge un programma, CLIPS li converte in spazi singoli.

Una questione di stile

È una buona norma di stile nella programmazione basata regole quella di utilizzare il primo campo di un fatto per descrivere la *relazione* dei campi successivi. Se utilizzato in questo modo, il primo campo è chiamato *relazione*. I restanti campi del fatto vengono utilizzati per valori specifici. Un esempio è (lista-della-spesa biscotti gelati caramelle salsa-fondente). I trattini vengono utilizzati per inserire più parole in un unico campo.

Una buona documentazione è ancora più importante in un sistema esperto che in linguaggi come Java, C, Ada, ecc., perché le regole di un sistema esperto non vengono generalmente eseguite in modo sequenziale. CLIPS aiuta il programmatore a scrivere fatti descrittivi come questo mediante i deftemplates.

Un altro esempio di fatti correlati è (duck), (horse) e (cow). È stilisticamente meglio chiamarli

```
(animal-is duck)
(animal-is horse)
(animal-is cow)
```

o come fatto unico

```
(animals duck horse cow)
```

poiché la relazione *animal-is* o *animals* descrive la loro relazione e quindi fornisce della documentazione a chi legge il codice.

Le relazioni esplicite, *animal-is* e *animals*, hanno più senso per una persona rispetto al significato implicito di (duck), (horse) e (cow). Anche se questo esempio è abbastanza semplice da consentire a chiunque di capire le relazioni implicite, è facile cadere in una trappola scrivendo fatti in cui la relazione *non* è così ovvia (in effetti, è molto più facile complicare che semplificare, poiché le persone sono più colpite dalla complessità che dalla semplicità).

Distanziamento

Poiché gli spazi vengono utilizzati per separare più campi, ne consegue che gli spazi non possono essere semplicemente inclusi nei fatti. Per esempio,

```
CLIPS> (clear)
CLIPS> (assert (animal-is walrus))
<Fact-1>
CLIPS> (assert ( animal-is walrus ))
<Fact-1>
CLIPS> (assert (    animal-is    walrus  ))
```

```
<Fact-1>
CLIPS> (facts)
f-1     (animal-is walrus)
For a total of 1 fact.
CLIPS>
```

Viene affermato un solo fatto, (animal-is walrus), poiché CLIPS ignora gli spazi bianchi e considera tutti questi fatti equivalenti. Pertanto, CLIPS risponde con <Fatto-1> quando si tenta di inserire gli ultimi due fatti duplicati. CLIPS normalmente non consente l'inserimento di fatti duplicati a meno che non si modifichi l'impostazione set-fact-duplication.

Per includere spazi in un fatto, si devono usare le virgolette doppie Per esempio,

```
CLIPS> (clear)
CLIPS> (assert (animal-is "duck"))
<Fact-1>
CLIPS> (assert (animal-is "duck "))
<Fact-2>
CLIPS> (assert (animal-is " duck"))
<Fact-3>
CLIPS> (assert (animal-is " duck "))
<Fact-4>
CLIPS> (facts)
f-1     (animal-is "duck")
f-2     (animal-is "duck ")
f-3     (animal-is " duck")
f-4     (animal-is " duck ")
For a total of 4 facts.
CLIPS>
```

Notare che gli spazi fanno sembrare ciascuno di questi fatti diverso a CLIPS sebbene il significato sia lo stesso per una persona.

Cosa succede se si vogliono includere le virgolette doppie in un campo? Il modo corretto per inserire virgolette doppie in un fatto è con la **barra rovesciata**, "\", come mostra l'esempio seguente.

```
CLIPS> (clear)
CLIPS> (assert (single-quote "duck"))
<Fact-1>
CLIPS> (assert (double-quote "\"duck\""))
<Fact-2>
CLIPS> (facts)
f-1     (single-quote "duck")
f-2     (double-quote ""duck"")
For a total of 2 facts.
CLIPS>
```

Ritirare questo fatto

Ora che si sa come inserire i fatti nell'elenco dei fatti, è tempo di imparare come rimuoverli. La rimozione dei fatti dall'elenco dei fatti è chiamata *retraction* (ritrazione) e viene eseguita con il comando retract. Per ritrattare un fatto, è necessario specificare l'indice del fatto come argomento della ritrazione. Per esempio, impostare l'elenco di fatti come segue.

```
CLIPS> (clear)
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (assert (animal-sound quack))
<Fact-2>
CLIPS> (assert (The duck says "Quack."))
<Fact-3>
CLIPS> (facts)
f-1     (animal-is duck)
f-2     (animal-sound quack)
f-3     (The duck says "Quack.")
For a total of 3 facts.
CLIPS>
```

Per rimuovere l'ultimo fatto con indice f-3, inserire il comando "retract" e poi controllare i fatti in questo modo.

```
CLIPS> (retract 3)
CLIPS> (facts)
f-1     (animal-is duck)
f-2     (animal-sound quack)
For a total of 2 facts.
CLIPS>
```

Cosa succede se si tenta di eliminare un fatto già ritrattato, oppure un fatto inesistente? Proviamolo e vediamo.

```
CLIPS> (retract 3)
[PRNTUTIL1] Unable to find fact f-3.
CLIPS>
```

Notare che CLIPS emette un messaggio di errore se si prova a ritrattare un fatto inesistente. La morale è che non ci si può riprendere ciò che non si è dato.
Ora ritiriamo gli altri fatti nel modo seguente.

```
CLIPS> (retract 2)
CLIPS> (facts)
f-1     (animal-is duck)
For a total of 1 fact.
CLIPS> (retract 1)
CLIPS> (facts)
CLIPS>
```

Si possono anche ritirare più fatti contemporaneamente, come mostrato di seguito.

```
CLIPS> (clear)
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (assert (animal-sound quack))
<Fact-2>
CLIPS> (assert (The duck says "Quack."))
<Fact-3>
CLIPS> (retract 1 3)
CLIPS> (facts)
f-2     (animal-sound quack)
For a total of 1 fact.
CLIPS>
```

Per ritirare più fatti, basta elencare i numeri dei fact-id nel comando (retract).
Si può semplicemente usare (**retract ***) per ritrattare tutti i fatti, dove "*" indica *tutti* .

```
CLIPS> (clear)
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (assert (animal-sound quack))
<Fact-2>
CLIPS> (assert (The duck says "Quack."))
<Fact-3>
CLIPS> (facts)
f-1     (animal-is duck)
f-2     (animal-sound quack)
f-3     (The duck says "Quack.")
For a total of 3 facts.
CLIPS> (retract *)
CLIPS> (facts)
CLIPS>
```

Guardare questo fatto

CLIPS fornisce diversi comandi per aiutare a eseguire il debug dei programmi. Un comando consente di **osservare [watch] continuamente i fatti** che vengono affermati e ritrattati. Questo è più conveniente che dover digitare un comando (facts) più e più volte e cercare di capire cosa è cambiato nell'elenco dei fatti.
Per iniziare a guardare i fatti, si inserisce il comando (watch facts) come mostrato nell'esempio seguente.

```
CLIPS> (clear)
CLIPS> (watch facts)
CLIPS> (assert (animal-is duck))
==> f-1     (animal-is duck)
<Fact-1>
CLIPS>
```

Il simbolo della **doppia freccia destra**, **==>**, significa che un fatto sta *entrando* nella memoria mentre la **doppia freccia sinistra** indica che un fatto *sta lasciando* la memoria, come mostrato di seguito.

```
CLIPS> (reset)
<== f-1     (animal-is duck)
CLIPS> (assert (animal-is duck))
==> f-1     (animal-is duck)
<Fact-1>
CLIPS> (retract 1)
<== f-1     (animal-is duck)
CLIPS> (facts)
CLIPS>
```

Il comando (watch facts) fornisce un record che mostra la **dinamica**; o il cambiamento dello stato dell'elenco dei fatti. Al contrario, il comando (facts) mostra lo stato **statico** della lista dei fatti poiché visualizza il contenuto corrente della fact-list. Per disattivare la sorveglianza dei fatti, si digita (**unwatch facts**).

Ci sono molte cose che che si possono sorvegliare. Queste comprendono quanto segue, descritte più dettagliatamente nel *CLIPS Reference Manual*. Il **commento** in CLIPS comincia con un **punto e virgola**. Tutto ciò che segue il punto e virgola viene ignorato da CLIPS.

```
(watch facts)
; istanze usate con gli oggetti
(watch instances)
; slot usati con gli oggetti
(watch slots)
(watch rules)
(watch activations)
; messagggi usati con gli oggetti
(watch messages)
; message-handlers usati con gli oggetti
(watch message-handlers)
(watch generic-functions)
(watch methods)
(watch deffunctions)
; le compilazioni vengono osservate [watched] per default
(watch compilations)
(watch statistics)
(watch globals)
(watch focus)
; tutto vede tutto
(watch all)
```

Utilizzando più spesso le funzionalità di CLIPS, si troveranno questi comandi (watch) molto utili per il debug. Per disattivare un comando (watch), si immette un comando **unwatch**. Ad esempio, per disattivare il watching delle compilazioni, si inserisce (unwatch compilations).

Capitolo 2

Seguire le regole

Se vuoi arrivare ovunque nella vita, non infrangere le regole: stabiliscile!

Creare buone regole

Per svolgere un lavoro utile, un sistema esperto deve avere regole oltre che fatti. Avendo visto come i fatti vengono affermati e ritrattati, è tempo di vedere come funzionano le regole. Una regola è simile a un'istruzione IF THEN in un linguaggio procedurale come Java, C o Ada. Una regola IF THEN può essere espressa in un misto di linguaggio naturale e linguaggio informatico come segue:

```
IF certe condizioni sono vere
THEN esegui le seguenti azioni
```

Un altro termine per l'affermazione precedente è **pseudocodice**, che letteralmente significa *falso codice*. Anche se lo pseudocodice non può essere eseguito direttamente dal computer, costituisce una guida molto utile per scrivere codice eseguibile. Lo pseudocodice è utile anche per documentare le regole. Una traduzione delle regole dal linguaggio naturale a CLIPS non è molto difficile se si tiene presente l'analogia dell'IF THEN. Man mano che si acquisisce esperienza con CLIPS, si scoprirà che scrivere regole in CLIPS diventa facile. È possibile digitare le regole direttamente in CLIPS oppure caricarle da un file di regole creato con un editor di testo.

Lo pseudocodice per una regola sui suoni delle anatre [duck] potrebbe essere

```
IF l'animale è un'anatra
THEN il suono prodotto è un quack
```

Quello che segue è un fatto e una regola chiamata *duck* che è lo pseudocodice sopra espresso nella sintassi CLIPS. Il nome della regola segue immediatamente la parola chiave *defrule*. Sebbene sia possibile inserire una regola su una singola riga, è consuetudine inserire parti diverse su righe separate per facilitare la leggibilità e la modifica.

```
CLIPS> (unwatch facts)
CLIPS> (clear)
CLIPS> (assert  (animal-is duck))
<Fact-1>
CLIPS>
(defrule duck
   (animal-is duck)
=>
   (assert (sound-is quack)))
CLIPS>
```

Se si digita la regola correttamente come mostrato, si dovrebbe vedere riapparire il prompt CLIPS.

Altrimenti, si vedrà un messaggio di errore. Se si riceve un messaggio di errore, è probabile che sia stata digitata male una parola chiave o omessa una parentesi. Da ricordare, il numero di parentesi sinistra e destra deve sempre corrispondere in un'istruzione.

La stessa regola viene mostrata di seguito con i commenti aggiunti per corrispondere alle parti della regola. Viene mostrato anche il commento facoltativo **rule-header** tra virgolette, "Here comes the quack". Può esserci un solo commento di rule-header e deve essere inserito dopo il nome della regola e prima del primo **pattern**. Anche se ora stiamo discutendo solo del pattern matching dei fatti, in generale si può avere un pattern matching un **pattern entity**. Un *pattern entity* è un fatto o un'istanza di una classe definita dall'utente. Il pattern matching di oggetti verrà discusso più avanti.

CLIPS tenta di far corrispondere il pattern della regola con quello di un'entità. Ovviamente, è possibile utilizzare spazi, tabulazioni e ritorni a capo per separare gli elementi di una regola per migliorarne la leggibilità. Gli altri commenti iniziano con un

punto e virgola e continuano finché non viene premuto il tasto ritorno a capo per terminare una riga. I commenti vengono ignorati da CLIPS.

```
; Rule header
(defrule duck
; Comment
"Here comes the quack"
; Pattern
(animal-is duck)
=> ; THEN arrow
; Action
(assert (sound-is quack)))
```

In CLIPS può esistere un solo nome di regola alla volta.

L'inserimento dello stesso nome della regola, in questo caso "duck", sostituirà qualsiasi regola esistente con quel nome.
Cioè, mentre in CLIPS possono esserci molte regole, può essercene una sola chiamata “duck”. Ciò è analogo ad altri linguaggi di programmazione in cui è possibile utilizzare un solo nome di procedura per identificarla in modo univoco.
Di seguito è mostrata la sintassi generale di una regola.

```
(defrule rule_name "optional_comment"
(pattern_1)   ; Left-Hand Side (LHS)
(pattern_2)   ; of the rule consisting
.          ; of elements before
.          ; the "=>"
.
(pattern_N)
=>
(action_1)    ; Right-Hand Side (RHS)
(action_2)    ; of the rule consisting
.          ; of elements after
.          ; the "=>". The last ")"
.          ; balances the opening
(action_M))   ; "(" to the left of
; "defrule". Be sure all
; your parentheses
; balance or you will
; get error messages
```

L'intera regola deve essere racchiusa tra parentesi. Ciascuno dei pattern di regole e **azioni** deve essere racchiuso tra parentesi. Un'azione è in realtà una funzione che in genere non ha un **valore di ritorno**, ma esegue alcune azioni utili, come (assert) o (retract). Ad esempio, un'azione potrebbe essere (assert (duck)). Qui il nome della funzione è "assert" e il suo argomento è "duck". Notare che non vogliamo alcun valore restituito come un numero. Vogliamo invece che il fatto (duck) venga affermato. Una **funzione** in CLIPS è un pezzo di codice eseguibile identificato da un nome specifico, che restituisce un valore utile o esegue un effetto collaterale utile, come (printout).

Una regola ha spesso più pattern e azioni. Il numero di pattern e azioni non deve essere necessariamente uguale, motivo per cui sono stati scelti indici diversi, N e M, per i pattern di regole e le azioni.

È possibile scrivere zero o più pattern dopo l'intestazione della regola. Ciascun pattern è costituito da uno o più campi. Nella regola duck, il pattern è (animal-is duck), dove i campi sono "animal-is" e "duck". CLIPS tenta di abbinare i pattern di regole ai fatti nella fact-list. Se tutti i pattern di una regola corrispondono ai fatti, la regola viene **activated** e messa in **agenda**. L'agenda è una raccolta di **activazioni** che sono quelle regole che corrispondono ai pattern delle entità. In agenda ci possono stare zero o più attivazioni.

Il simbolo “**=>**” che segue i pattern in una regola si chiama **freccia** [arrow]. La freccia rappresenta l'inizio della parte THEN di una regola IF-THEN (e può essere letta come “implica”).

L'ultima parte di una regola è la lista di zero o più azioni che verranno eseguite quando la regola **si attiva** [fires]. Nel nostro esempio, l'unica azione è affermare il fatto (sound-is quack). Il termire *fires* significa che CLIPS ha selezionato una determinata regola da eseguire dall'agenda.

Un programma cesserà l'esecuzione quando non ci sono attivazioni nell'agenda.

Quando sono previste più attivazioni, CLIPS determina automaticamente quale è quella più appropriata da attivare. CLIPS ordina le attivazioni in agenda in termini di priorità crescente o di **salience**.

La parte della regola prima della freccia è detta left-hand side (**LHS**) mentre la parte dopo la freccia è detta right-hand side (**RHS**). Se non viene specificato alcun pattern, CLIPS automaticamente attiva la regola quando viene immesso un comando **(reset)**.

Come fa l'anatra?

CLIPS esegue sempre le azioni sul lato destro "RHS" della regola con la priorità più alta nell'agenda. Questa regola viene poi rimossa dall'agenda e vengono eseguite le azioni della nuova regola di massima "salience" (salienza). Questo processo continua finché non ci sono più attivazioni o finché non viene incontrato un comando di stop.
Si può controllare cosa c'è nell'agenda col comando **agenda**. Per esempio,

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

Il primo numero "0" è la "salience" (salienza) dell'attivazione "duck" e "f-1" è l'identificatore del fatto (animal-is duck), che corrisponde [match] all'attivazione. Se la salienza di una regola non è dichiarata esplicitamente, CLIPS le assegna il valore predefinito pari a zero, dove i possibili valori di salienza vanno da -10.000 a 10.000. In questo libro useremo la definizione del termine **default** per indicare il *modo standard*.

Se c'è una sola regola nell'agenda, verrà attivata. Poiché il pattern LHS (a sinistra) della regola del suono dell'anatra è

```
(animal-is duck)
```

tale pattern sarà soddisfatto dal fatto (animal-is duck) e quindi la regola del suono dell'anatra dovrebbe attivarsi.

Ogni campo del pattern è detto essere un **vincolo letterale**. Il termine **letterale** significa avere un valore costante, al contrario di una *variabile* il cui valore potrebbe cambiare. In questo caso, i letterali sono “animal-is” e “duck”.

Per eseguire un programma è sufficiente inserire il comando >**run**. Digitare (run) e premere return. Poi eseguire (facts) per verificare che il fatto sia stato affermato dalla regola.

```
CLIPS> (run)
CLIPS> (facts)
f-1     (animal-is duck)
f-2     (sound-is quack)
For a total of 2 facts.
CLIPS>
```

Prima di proseguire, salviamo la regola duck col comando **save** in modo da non doverla digitare nuovamente (se non è già salvata in un editor). Basta inserire un comando come

```
(save "duck.clp")
```

per salvare la regola dalla memoria CLIPS su disco e denominare il file "duck.clp" dove ".clp" è semplicemente un'estensione di comodo per ricordarci che si tratta di un file di codice sorgente CLIPS. Notare che il salvataggio del codice dalla memoria CLIPS in questo modo conserverà solo il commento facoltativo del rule-header tra virgolette e non eventuali commenti col punto e virgola.

Al diavolo la papera

A questo punto potrebbe sorgere una domanda interessante. Cosa succede se si esegue di nuovo (run)? C'è una regola e un fatto che soddisfa la regola, quindi la regola dovrebbe essere attivata. Ma, se si esegue di nuovo (run), la regola non si attiverà. Questo potrebbe risultare alquanto frustrante. Tuttavia, prima di fare qualcosa di drastico per alleviare la frustrazione, come prendere a calci la papera, si deve sapere qualcosa in più su alcuni principi di base dei sistemi esperti. Una regola viene attivata se i suoi pattern corrispondono a

1. un pattern entity completamente nuova che non esisteva prima o,
2. un pattern entity che esisteva prima ma che è stato "retracted" e "reasserted", cioè un "clone" del vecchio pattern entity e quindi ora un nuovo pattern entity.

La regola, e gli indici dei pattern corrispondenti, è l'attivazione. Se la regola o il pattern entity, o entrambi, cambiano, l'attivazione viene rimossa. Un'attivazione può anche essere rimossa tramite un comando o tramite un'azione di un'altra regola che è stata attivata prima e ha rimosso le condizioni necessarie per l'attivazione.

Lo "Inference Engine" ordina le attivazioni in base alla loro salienza [salience]. Questo processo di ordinamento è detto **risoluzione dei conflitti** perché elimina il conflitto che riguarda la decisione su quale regola debba essere attivata successivamente. CLIPS esegue la parte destra (RHS) della regola con la salienza massima nell'agenda e rimuove l'attivazione. Questa esecuzione si chiama "attivazione della regola" in analogia alla "attivazione di un neurone". Un neurone emette un impulso di tensione quando viene applicato uno stimolo appropriato. Dopo che un neurone si è attivato, subisce la **rifrazione** [refraction] e non può attivarsi nuovamente per un certo periodo di tempo. Senza la rifrazione, i neuroni continuerebbero a attivarsi ancora e ancora esattamente sullo stesso stimolo.

Senza la rifrazione, i sistemi esperti rimarrebbero sempre intrappolati in loop banali. Cioè, non appena una regola viene attivata, continua a basarsi sullo stesso fatto ancora e ancora. Nel mondo reale, lo stimolo che ha causato l'attivazione prima o poi scomparirebbe. Ad esempio, una vera papera potrebbe volare via o trovare lavoro nel cinema. Tuttavia, nel mondo dei computer, una volta archiviati, i dati rimangono lì finché non vengono esplicitamente rimossi o non viene tolta la corrente.

L'esempio seguente mostra le attivazioni e l'avvio di una regola. Si noti che i comandi (watch) vengono utilizzati per mostrare con maggiore attenzione ogni fatto e ogni attivazione. La freccia a destra indica un fatto o un'attivazione in entrata mentre una freccia a sinistra indica un fatto o un'attivazione in uscita.

```
; I commenti in blue/corsivo sono stati
; aggiunti per le spiegazioni. Non si
; vedranno nell'output effettivo
CLIPS> (clear)
CLIPS>
(defrule duck
(animal-is duck)
=>
(assert (sound-is quack)))
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (assert (animal-is duck))
==> f-1     (animal-is duck)
; Activation salience is 0 by default,
; then rule name:pattern entity index
==> Activation 0      duck: f-1
<Fact-1>
; Notice that duplicate fact
; cannot be entered
CLIPS> (assert (animal-is duck))
<Fact-1>
CLIPS> (agenda)
0      duck: f-1
For a total of 1 activation. CLIPS> (run)
==> f-2     (sound-is quack)
; Nothing on agenda after rule fires
; Even though fact matches rule,
; refraction will not allow this
; activation because the rule already
; fired on this fact
CLIPS> (agenda)
CLIPS> (facts)
f-1     (animal-is duck)
f-2     (sound-is quack)
For a total of 2 facts.
CLIPS> (run)
CLIPS>
```

Puoi far attivare nuovamente la regola se si ritrae [retract] il fatto e poi lo si asserisce come un fatto nuovo.

Mostrare le regole

A volte potrebbe essere utile vedere una regola mentre si è in CLIPS. C'è un comando chiamato **ppdefrule** – la stampa formattata della regola – che stampa una regola. Per visualizzare una regola, specificare il nome della regola come argomento di *ppdefrule*. Per esempio,

```
CLIPS> (ppdefrule duck)
(defrule MAIN::duck
(animal-is duck)
=>
(assert (sound-is quack)))
CLIPS>
```

CLIPS pone diverse parti della regola su righe diverse per motivi di leggibilità. I pattern prima della freccia sono ancora considerati il lato sinistro e le azioni dopo la freccia sono ancora considerate il lato destro della regola. Il termine *MAIN* si riferisce al modulo MAIN in cui si trova questa regola per default. È possibile definire moduli per inserire regole in modo analogo alle istruzioni che possono essere inserite in diversi pacchetti, moduli, procedure o funzioni di altri linguaggi di programmazione. L'uso dei moduli semplifica la scrittura di sistemi esperti con molte regole poiché queste possono essere raggruppate insieme con agende di ciascun modulo. Per ulteriori informazioni, consultare il *CLIPS Reference Manual*.

Cosa succede se si stampa una regola ma non se ne ricorda il nome? No problem. Basta usare il comando **rules** in risposta ad un prompt CLIPS e CLIPS stamperà i nomi di tutte le regole. Per esempio, inserire

```
CLIPS> (rules)
duck
For a total of 1 defrule.
CLIPS>
```

Scrivimi

Oltre ad asserire i fatti nella RHS delle regole, si possono anche stampare informazioni utilizzando la funzione **printout**. CLIPS ha anche una parola chiave per return/linefeed chiamata **crlf** che è molto utile per migliorare l'aspetto dell'output formattandolo su righe diverse. Tanto per cambiare, *crlf* non è incluso tra parentesi. Come esempio,

```
CLIPS>
(defrule duck
(animal-is duck)
=>
;; Assicurarsi di digitare la "t"
(printout t "quack" crlf))
==> Activation 0      duck: f-1
CLIPS> (run)
quack
CLIPS>
```

L'output è il testo tra virgolette doppie. Si deve digitare la lettera "**t**" dopo il comando printout. Questo dice a CLIPS di inviare l'output al **dispositivo di output standard** del computer. Generalmente, il dispositivo di output standard è il terminale (da qui la lettera "t" dopo printout). Tuttavia, questo può essere ridefinito in modo che quello di output sia qualche altro dispositivo, come un modem o un disco.

Altre funzionalità

Il comando **declare salience** fornisce un controllo esplicito su quali regole verranno messe nell'agenda. Si deve fare attenzione nell'usare questa funzionalità troppo liberamente per evitare che il programma diventi troppo controllato. Un modo per forzare la ri-attivazione di una regola è tramite il comando **refresh** regola.

Il comando **load** carica la regola precedentemente salvata su disco nel file "duck.clp" o qualunque nome e directory in cui sia stata salvata. È possibile caricare in CLIPS un file di regole creato con un editor di testo utilizzando il comando load.

Un modo più veloce per caricare i file è salvarli prima in un formato binario leggibile dalla macchina con il comando per il salvataggio binario chiamato **bsave**. Il comando per il caricamento binario, **bload**, può poi essere utilizzato per leggere queste regole binarie

nella memoria CLIPS molto più velocemente poiché i file non devono essere reinterpretati da CLIPS.

Altri due utili comandi consentono di salvare e caricare fatti utilizzando un file. Questi sono **savefacts** e **load-facts**. (save-facts) salverà tutti i fatti nella *fact-list* in un file mentre (load-facts) caricherà i fatti da un file nella *fact-list*.

Il comando **batch** consente di eseguire comandi da un file come se fossero stati digitati al livello più alto. Un altro comando utile fornisce un'interfaccia per il sistema operativo. Il comando **system** consente l'esecuzione di comandi o eseguibili del sistema operativo all'interno di CLIPS. Per ulteriori informazioni su tutti questi argomenti, consultare il *CLIPS Reference Manual*.

Capitolo 3

Aggiungere i dettagli

Il problema non è il quadro generale, ma i dettagli

Nei primi due capitoli riguardavano i fondamenti di CLIPS. Ora vedremo come costruire su queste basi per creare programmi più potenti.

Stop & Go

Finora abbiamo visto solo il tipo di programma più semplice, composto da una sola regola. Tuttavia, i sistemi esperti costituiti da una sola regola non sono molto utili. I sistemi esperti pratici possono essere costituiti da centinaia o migliaia di regole. Diamo ora un'occhiata a un'applicazione che richiede più regole.

Supponiamo che si voglia scrivere un sistema esperto per determinare come un robot mobile dovrebbe rispondere a un semaforo. È meglio scrivere questo tipo di problema utilizzando più regole. Ad esempio, le regole per le situazioni di semaforo rosso e verde possono essere scritte in questo modo.

```
(defrule red-light
(light red)
=>
(printout t "Stop" crlf))

(defrule green-light
(light green)
=>
(printout t "Go" crlf))
```

Dopo che le regole sono state inserite in CLIPS, si afferma un fatto (light red) e si esegue [run]. Si vedrà stampato "Stop". Ora si asserisce un fatto (light green) e run. Verrà stampato “Go”.

Andare a piedi

Riflettendoci, esistono altre possibilità oltre ai semplici casi rosso, verde e giallo. Alcuni semafori hanno anche una freccia verde per le svolte protette a sinistra. Alcuni hanno una mano che si illumina per indicare se una persona può attraversare oppure no. Alcuni hanno cartelli che dicono di attraversare o meno. Quindi, a seconda che il nostro robot stia camminando o guidando, potrebbe dover prestare attenzione a segnali diversi.

Oltre alle informazioni sullo stato della luce, devono essere fornite anche le informazioni relative al cammino a piedi o alla guida. È possibile creare regole per coprire queste condizioni, ma devono avere più di un pattern. Ad esempio, supponiamo di voler attivare una regola se il robot sta camminando e se il segnale di attraversamento dice "attraversa" [walk]. Una regola potrebbe essere scritta così:

```
(defrule take-a-walk
(status walking)
(walk-sign walk)
=>
(printout t "Go" crlf))
```

La regola di cui sopra ha due pattern. Entrambi i pattern devono essere soddisfatti dai fatti nella fact-list affinché la regola venga applicata. Per vedere come funziona, inserire la regola e poi affermare i fatti (status walking) e (walk-sign walk). Col (run), il programma stamperà “Go” poiché entrambi i pattern sono soddisfatti e la regola viene attivata.

Si possono avere un numero qualsiasi di pattern o azioni in una regola. Il punto importante da comprendere è che la regola viene posta nell'agenda solo se *tutti* i pattern sono soddisfatti dai fatti. Questo tipo di restrizione è detto **logical AND conditional element** (**CE**) in riferimento alla relazione AND della logica booleana. Una relazione AND si dice vera solo se tutte le sue condizioni sono vere.

Poiché i pattern sono del tipo logico AND, la regola non verrà attivata se solo uno dei pattern è soddisfatto. Tutti i fatti devono essere presenti prima che il lato sinistro di una regola sia soddisfatto e la regola venga inserita nell'agenda.

Una questione di strategia

La parola **strategia** era originariamente un termine militare per la pianificazione e le operazioni di guerra. Oggi, il termine *strategia* è comunemente usato negli affari (perché gli affari sono una guerra) per riferirsi ai piani di alto livello di un'organizzazione per raggiungere i propri obiettivi, ad esempio: "Guadagnare un sacco di soldi vendendo hamburger più grassi di chiunque altro al mondo!"

Nei sistemi esperti, un uso del termine *strategia* è nella risoluzione dei conflitti delle attivazioni. Si potrebbe dire: “Bene, progetterò semplicemente il mio sistema esperto in modo che possa essere attivata una sola regola alla volta. Quindi non c’è bisogno di risolvere conflitti”. La buona notizia è che se ci si riesce, la risoluzione dei conflitti non è necessaria. La cattiva notizia è che questo successo dimostra che l'applicazione può essere ben rappresentata da un programma sequenziale. Quindi lo si potrebbe codificare in C, Java o Ada in primo luogo, senza preoccuparti di scriverlo come un sistema esperto.

CLIPS offre sette diverse modalità di risoluzione dei conflitti: profondità [depth], ampiezza [breadth], LEX, MEA, complessità [complexity], semplicità [simplicity] e casuale [random]. È difficile dire che una sia chiaramente migliore di un'altra senza considerare l’applicazione specifica. Anche allora, potrebbe essere difficile giudicare quale sia il “migliore”. Per ulteriori informazioni sui dettagli di queste strategie, consultare il *CLIPS Reference Manual*.

La **depth strategy** è la **strategia** di **default** standard di CLIPS. Il default viene impostato automaticamente al primo avvio di CLIPS. Successivamente è possibile modificare il default. Nella "depth strategy", le nuove attivazioni vengono inserite nell’agenda dopo quelle con maggiore salienza, ma prima delle attivazioni con uguale o minore salienza. Tutto ciò significa semplicemente che l’agenda è ordinata dalla rilevanza più alta a quella più bassa.

In questo libro, tutte le discussioni e gli esempi presupporranno una "depth strategy".

Ora che tutte queste diverse impostazioni opzionali sono disponibili, assicurarsi, prima di eseguire un sistema esperto sviluppato da qualcun altro, che le proprie impostazioni siano le stesse delle loro. Altrimenti, si potrebbe scoprire che l'operazione è inefficiente o addirittura errata. In effetti, è una buona idea codificare esplicitamente tutte le impostazioni in qualsiasi sistema che si sviluppa in modo che venga configurato correttamente.

Deffacts

Lavorando con CLIPS, ci si potrebbe stancare di digitare le stesse asserzioni dal livello più alto. Se si intendono utilizzare le stesse asserzioni ogni volta che viene eseguito un programma, si possono caricare prima le asserzioni da un disco utilizzando un file batch. Un modo alternativo per inserire i fatti è utilizzare la parola chiave define fact, **deffacts**. Per esempio,

```
CLIPS> (unwatch facts)
CLIPS> (unwatch activations)
CLIPS> (clear)
CLIPS>
(deffacts walk "Some facts about walking"
; status fact to be asserted
(status walking)
; walk-sign fact to be asserted
(walk-sign walk))
CLIPS> (reset)
; reset causes facts from
; deffacts to be asserted
CLIPS> (facts)
f-1     (status walking)
f-2     (walk-sign walk)
For a total of 2 facts.
CLIPS>
```

Il nome richiesto di questa istruzione deffacts, *walk*, segue la parola chiave deffacts. Dopo il nome c'è un commento facoltativo tra virgolette doppie. Come il commento facoltativo di una regola, il commento (deffacts) verrà mantenuto con i (deffacts) dopo che è stato

caricato da CLIPS. Dopo il nome o il commento ci sono i fatti che verranno asseriti nella fact-list. I fatti in una dichiarazione deffacts vengono asseriti utilizzando il comando CLIPS (reset).

Il comando (reset) ha un vantaggio rispetto al comando (clear) in quanto (reset) non elimina tutte le regole. Il (reset) lascia le regole intatte. Come (clear), rimuove tutte le regole attivate dall'agenda e rimuove anche tutti i vecchi fatti dalla fact-list. Dare un comando (reset) è un modo consigliato per avviare l'esecuzione del programma, soprattutto se il programma è stato eseguito in precedenza e la fact-list è ingombra di vecchi fatti.

In sintesi, il (reset) fa due cose per i fatti.

1. Rimuove i fatti esistenti dalla fact-list, il che potrebbe rimuovere le regole attivate dall'agenda.
2. Asserisce fatti da affermazioni esistenti (deffacts).

In realtà, (reset) esegue anche operazioni corrispondenti sugli oggetti. Elimina istanze e asserisce istanze da **definstances**.

Eliminazione selettiva

Il comando **undeffacts** elimina un (deffacts) dall'asserzione dei fatti eliminando i deffacts dalla memoria. Per esempio,

```
CLIPS> (undeffacts walk)
CLIPS> (reset)
CLIPS> (facts)
CLIPS>
```

Questo esempio dimostra come i passi [walk] (deffacts) siano stati eliminati. Per ripristinare un'istruzione deffacts dopo un comando (undeffacts), è necessario immettere nuovamente l'istruzione deffacts. Oltre ai fatti, CLIPS consente anche di eliminare le regole in modo selettivo utilizzando **undefrule**.

Osservazioni!

Si possono osservare (**watch rules**)le regole attivate e le attivazioni (**watch activations**) nell'agenda. **watch statistics** stampa le informazioni sul numero di regole attivate, il tempo di esecuzione, regole al secondo, numero medio di fatti, numero massimo di fatti, numero medio di attivazioni e numero massimo di attivazioni. Le informazioni statistiche sono essere utili per "mettere a punto" (**tuning up**) un sistema esperto per ottimizzarne la velocità. Un altro comando, è **watch compilations** e mostra le informazioni quando vengono caricate le regole. Il comando **watch all** guarderà tutto.

La stampa delle informazioni di "watch" sullo schermo o su disco con il comando **dribble** rallenterà un po' il programma perché CLIPS impiega più tempo per stampare o salvare su disco. Il comando **dribble-on** memorizzerà tutti gli input e output immessi al prompt dei comandi in un file su disco finché non verrà immesso il comando **dribble-off**. Questo è utile per fornire una registrazione permanente di tutto ciò che accade. Questi comandi sono i seguenti.

```
(dribble-on <filename>)
(dribble-off <filename>)
```

Un altro utile comando di debug è (run) che accetta come argomento facoltativo il numero di attivazioni della regola. Ad esempio, un comando (run 21) direbbe a CLIPS di eseguire il programma e poi di interromperlo dopo 21 attivazioni di regole. Un comando (run 1) consente di scorrere un programma attivando una regola alla volta.

Proprio come molti altri linguaggi di programmazione, CLIPS dà anche la possibilità di impostare dei **breakpoint**. Un breakpoint è semplicemente un indicatore che interrompe l'esecuzione di CLIPS appena prima di eseguire una regola specifica. Un breakpoint viene impostato dal comando **set-break**. Il comando **remove-break** rimuoverà un breakpoint impostato. **show-breaks** elencherà tutte le regole per le quali sono impostati dei breakpoint. La sintassi di questi comandi per l'argomento <rulename> è mostrata di seguito.

```
(set-break <rulename>)
(remove-break <rulename>)
```

```
(show-breaks)
```

Una buona corrispondenza

Ci sono situazioni in cui si è certi che una regola dovrebbe essere attivata ma non lo è. Sebbene sia possibile che ciò sia dovuto a un bug in CLIPS, non è molto probabile a causa della grande abilità delle persone che hanno programmato CLIPS (NOTA: ANNUNCIO COMMERCIALE A PAGAMENTO PER GLI SVILUPPATORI).

Nella maggior parte dei casi, il problema si verifica a causa del modo in cui è stata scritta la regola. Come aiuto per il debug, CLIPS ha il comando **matches** che può dire quali pattern in una regola corrispondono ai fatti. I pattern che non corrispondono impediscono l'attivazione della regola. Una ragione comune per cui un pattern non corrisponde a un fatto deriva dall'ortografia errata di un elemento del pattern o dall'asserzione del fatto.

L'argomento di (matches) è il nome della regola di cui verificare le corrispondenze. Per vedere come funziona (matches), prima si esegue (clear), poi si immette la regola seguente.

```
(defrule take-a-vacation
; Conditional element 1
(work done)
; Conditional element 2
(money plenty)
; Conditional element 3
(reservations made)
=>
(printout t "Let's go!!!" crlf))
```

Di seguito viene mostrato come viene utilizzato (matches). Immettere i comandi come mostrato. Notare che (watch facts) è attivato. Questa è una buona idea quando si asseriscono i fatti manualmente poiché dà l'opportunità di controllare l'ortografia dei fatti.

```
CLIPS> (watch facts)
CLIPS> (assert (work done))
==> f-1     (work done)
<Fact-1>
CLIPS> (matches take-a-vacation)
Matches for Pattern 1 f-1
Matches for Pattern 2
None
Matches for Pattern 3
None
; CE is conditional element
Partial matches for CEs 1 - 2
None
Partial matches for CEs 1 - 3
None
Activations
None
; The return value indicates the total patterns
; matched, the total partial matches, and the
; total activations
(1 0 0)
CLIPS>
```

Il fatto con fact-identifier f-1 corrisponde al primo pattern o elemento condizionale nella regola ed è segnalato da (matches). Dato che una regola ha N pattern, il termine **partial matches** si riferisce a qualsiasi insieme di corrispondenze dei primi N-1 pattern con i fatti. Cioè, le corrispondenze parziali iniziano con il primo pattern in una regola e terminano con qualsiasi pattern fino all'ultimo (N-esimo) pattern escluso. Non appena non è possibile effettuare una corrispondenza parziale, CLIPS non effettua più alcun controllo. Ad esempio, una regola con quattro pattern avrà corrispondenze parziali del primo e del secondo pattern e anche del primo, del secondo e del terzo pattern. Se tutti gli N pattern corrispondono, la regola verrà attivata.

Altre funzionalità

Alcuni comandi aggiuntivi sono utili con i deffacts. Per esempio, il comando **list-deffacts** elencherà i nomi dei deffacts attualmente caricati in CLIPS. Un altro comando utile è **ppdeffacts** che stampa i fatti memorizzati in un deffacts.

Altre funzioni consentono di manipolare facilmente le stringhe:

assert-string: Esegue una stringa di asserzione prendendo una stringa come argomento e affermandola come un fatto non-stringa.
str-cat: Costruisce una stringa tra virgolette singole da singoli elementi mediante concatenazione. **str-index**: Restituisce un indice di stringa della prima occorrenza di una sottostringa.
sub-string: Restituisce una sottostringa da una stringa.
str-compare: Esegue un confronto tra stringhe.
str-length: Restituisce la lunghezza della stringa.
sym-cat: Restituisce un simbolo concatenato.

Per stampare una variabile multicampo senza parentesi, il modo più semplice è quello di utilizzare la funzione implode di stringa, implode$.

Capitolo 4

Interessi Variabili

Niente cambia più del cambiamento

Il tipo di regole viste finora illustra una semplice corrispondenza di pattern con fatti. In questo capitolo si impareranno modi molto efficaci per abbinare [match] e manipolare i fatti.

Diventiamo variabili

Proprio come con altri linguaggi di programmazione, CLIPS ha **variabili** per memorizzare dei valori. A differenza di un fatto, che è **statico** o immutabile, il contenuto di una variabile è **dinamico** man mano che cambiano i valori ad essa assegnati. Al contrario, una volta affermato un fatto, i suoi campi possono essere modificati solo ritirando [retracting] e affermando un nuovo fatto con i campi modificati. Anche l'azione *modify* (descritta più avanti nel capitolo su deftemplate) agisce ritrattando e affermando un fatto modificato, come si può vedere controllando l'indice dei fatti.

Il nome di una variabile, o **identificatore di variabile**, è sempre scritto con un punto interrogativo seguito da un simbolo che è il nome della variabile. Il formato generale è

```
?<variable-name>
```

Le variabili globali, che verranno descritte più dettagliatamente in seguito, hanno una sintassi leggermente diversa.

Proprio come in altri linguaggi di programmazione, i nomi delle variabili dovrebbero essere significativi per un buon stile. Seguono alcuni esempi di nomi di variabili validi.

```
?x
?noun
?color
?sensor
?valve
?ducks-eaten
```

Prima di poter utilizzare una variabile, è necessario assegnarle un valore. Come esempio di un caso in cui un valore *non* è assegnato, provare a inserire quanto segue e CLIPS risponderà con il messaggio di errore mostrato.

```
CLIPS> (unwatch all)
CLIPS> (clear)
CLIPS>
(defrule test
=>
(printout t ?x crlf))

[PRCCODE3] Undefined variable x referenced in RHS of defrule.

ERROR:
(defrule MAIN::test
=>
(printout t ?x crlf))
CLIPS>
```

CLIPS restituisce un messaggio di errore quando non riesce a trovare un valore **vincolato** [bound] a ?x. Il termine *vincolato* indica l'assegnazione di un valore a una variabile. Solo le variabili globali sono vincolate [bound] in *tutte* le regole. Tutte le altre variabili sono vincolate solo *all'interno* di una regola. Prima e dopo l'attivazione di una regola, le variabili non-globali non sono vincolate [bound] e quindi CLIPS genererà un messaggio di errore se si tenta di interrogare una variabile non-bound.

Asserire

Un uso comune delle variabili è quello di far corrispondere un valore sul lato sinistro (LHS) e poi affermare questa variabile associata sul lato destro RHS. Per esempio, inserire

```
(defrule make-quack
(duck-sound ?sound)
=>
```

```
(assert (sound-is ?sound)))
```

Ora asserire (duck-sound quack), poi il (run) del programma. Verificate i fatti e vedrete che la regola ha prodotto (sound-is quack) perché la variabile ?sound era [bound] a quack.

Naturalmente è anche possibile utilizzare una variabile più di una volta. Per esempio, inserire quanto segue. Eseguire nuovamente un (reset) e nuovamente l'asserzione (duck-sound quack).

```
(defrule make-quack
(duck-sound ?sound)
=>
(assert (sound-is ?sound ?sound)))
```

Quando la regola viene attivata, verrà prodotto (sound-is quack quack) poiché la variabile ?sound viene utilizzata due volte.

Come fa la papera

Le variabili vengono comunemente utilizzate anche nell'output di stampa, come in

```
(defrule make-quack
(duck-sound ?sound)
=>
(printout t "The duck said "
?sound crlf))
```

Eseguire un (reset), inserire questa regola e asserire il fatto e poi (run) per scoprire la papera come fa. Come si modificherebbe la regola per inserire virgolette doppie attorno al quack nell'output?

Si possono utilizzare più di una variabile in uno schema, come mostra l'esempio seguente.

```
CLIPS> (clear)
CLIPS>
(defrule whodunit
(duckshoot ?hunter ?who)
=>
(printout t ?hunter " shot "
?who crlf))
CLIPS> (assert (duckshoot Brian duck))
<Fact-1>
; Duck dinner tonight!
CLIPS> (run)
Brian shot duck
CLIPS> (assert (duckshoot duck Brian))
<Fact-2>
; Brian dinner tonight!
CLIPS> (run)
duck shot Brian
; Missing third field
CLIPS> (assert (duckshoot duck))
<Fact-3>
; Rule doesn't fire,
; no output
CLIPS> (run)
CLIPS>
```

Notare quale grande differenza faccia l'ordine dei campi nel determinare chi ha sparato a chi. Si può anche vedere che la regola *non* è stata attivata quando il fatto (duckshoot duck) è stato affermato. La regola non è stata attivata perché nessun campo del fatto corrispondeva al secondo vincolo del pattern, ?who.

The Happy Bachelor

La retrazione [Retraction] è molto utile nei sistemi esperti e solitamente viene eseguita sul lato destro (RHS) anziché al livello superiore. Prima che un fatto possa essere ritratto, deve essere specificato in CLIPS. Per ritrarre un fatto da una regola, il **fact-address** (indirizzo del fatto) deve prima essere associato [bound] a una variabile sul lato sinistro (LHS).

C'è una grande differenza tra il binding di una variabile al contenuto di un fatto e il binding a una variabile al fact-address. Negli esempi visti come (duck-sound ?sound), una variabile era legata [bound] al valore di un campo. Cioè, ?sound è stato "bound" a quack. Tuttavia, se si desidera rimuovere il fatto i cui contenuti sono (duck-sound quack), è necessario prima comunicare a CLIPS l'*indirizzo* del fatto da ritirare.

Il fact-address viene specificato utilizzando la **freccia sinistra**, “<-”. Per crearlo, basta digitare un simbolo “<” seguito da un “-”. Come esempio di ritrattazione di fatti da una regola,

```
CLIPS> (clear)
CLIPS> (assert (bachelor Dopey))
<Fact-1>
CLIPS> (facts)
f-1     (bachelor Dopey)
For a total of 1 fact.
CLIPS>
(defrule get-married
?duck <- (bachelor Dopey)
=>
(printout t "Dopey is now happily married "
?duck crlf)
(retract  ?duck))
CLIPS> (run)
Dopey is now happily married <Fact-1>
CLIPS> (facts)
CLIPS>
```

Si noti che (printout) stampa l'indice dei fatti di ?duck, <Fatto-1>, poiché la freccia sinistra lega [bound] l'indirizzo del fatto a ?duck. Inoltre, non esiste alcun fatto (bachelor Dopey) perché è stato ritirato [retracted].

Le variabili possono essere utilizzate per acquisire un valore di fatto contemporaneamente a un indirizzo, come mostrato nell'esempio seguente. Per comodità è stato definito anche un (deffacts).

```
CLIPS> (clear)
CLIPS>
(defrule marriage
?duck <- (bachelor ?name)
=>
(printout t ?name
" is now happily married"
crlf)
(retract  ?duck))
CLIPS>
(deffacts good-prospects
(bachelor Dopey)
(bachelor Dorky)
(bachelor Dicky))
CLIPS> (reset)
CLIPS> (run)
Dicky is now happily married
Dorky is now happily married
Dopey is now happily married
CLIPS>
```

Notare come la regola si attivava su *tutti* i fatti che corrispondevano al pattern (bachelor ?name). CLIPS ha anche una funzione chiamata **fact-index** che può essere utilizzata per restituire l'indice dei fatti di un indirizzo di un fatto.

Non è importante

Invece di associare [binding] un valore di campo a una variabile, la presenza di un campo non vuoto può essere rilevata da sola utilizzando un **wildcard** (carattere jolly). Ad esempio, supponiamo che si stia gestendo un servizio di appuntamenti per anatre e che una papera affermi di uscire solo con anatre il cui nome è Dopey. In realtà, in questa specifica sono presenti due criteri poiché implica che l'anatra debba avere più di un nome. Quindi un semplice (bachelor Dopey) non è adeguato perché c'è un solo nome nel fatto.

Questo tipo di situazione, in cui viene specificata solo una parte del fatto, è molto comune e molto importante. Per risolvere questo problema, è possibile utilizzare un carattere jolly per abbinare i Dopeys.

La forma più semplice di carattere jolly è chiamata **single-field wildcard** ed è visualizzata da un punto interrogativo, “**?**”. “?” è anche detto un **single-field constraint**. Un carattere jolly a campo singolo indica *esattamente* un campo, come mostrato di seguito.

```
CLIPS> (clear)
CLIPS>
(defrule dating-ducks
```

```
(bachelor Dopey ?)
=>
(printout t "Date Dopey"
crlf))
CLIPS>
(deffacts duck
(bachelor Dicky)
(bachelor Dopey)
(bachelor Dopey Mallard)
(bachelor Dinky Dopey)
(bachelor Dopey Dinky Mallard))
CLIPS> (reset)
CLIPS> (run)
Date Dopey
CLIPS>
```

Il pattern include un carattere jolly per indicare che il cognome di Dopey non è importante. Finché il nome è Dopey e c'è un cognome (ma nessun secondo nome), la regola sarà soddisfatta e verrà attivata. Poiché il modello ha tre campi di cui uno è un carattere jolly a campo singolo, solo il fatto di *esattamente* tre campi può soddisfarlo. In altre parole, solo i Dopeys con esattamente due nomi possono soddisfare questa paperella.

Supponiamo di voler specificare Dopeys con esattamente tre nomi? Tutto quello che si dovrebbe fare è scrivere un pattern come

```
(bachelor Dopey ? ?)
```

o, se solo persone con tre nomi il cui secondo nome era Dopey,

```
(bachelor ? Dopey ?) o, se solo il cognome fosse Dopey, come nel seguente:
(bachelor ? ? Dopey)
```

Un'altra possibilità interessante si verifica se Dopey *deve* essere il primo nome, ma sono accettabili solo quei Dopey con due *o* tre nomi. Un modo per risolvere questo problema è scrivere due regole. Per esempio

```
(defrule eligible
(bachelor Dopey ?)
=>
(printout t "Date Dopey" crlf))
(defrule eligible-three-names
(bachelor Dopey ? ?)
=>
(printout t "Date Dopey" crlf))
```

Inserire ed eseguire questo per vedere che vengono stampati Dopeys con due e tre nomi. Naturalmente, se non si vogliono appuntamenti anonimi, si devono associare i nomi dei Dopey con una variabile e stamparli.

Dare di matto

Invece di scrivere regole separate per gestire ciascun campo, è molto più semplice utilizzare il **carattere jolly multicampo**. Questo è il simbolo del dollaro seguito da un punto interrogativo, "**$?**" e rappresenta *zero o più campi*. Notare come questo contrasta con il carattere jolly a campo singolo che deve corrispondere esattamente a un campo.

Le due regole per gli appuntamenti possono ora essere scritte in un'unica regola come segue.

```
CLIPS> (clear)
CLIPS>
(defrule dating-ducks
(bachelor Dopey $?)
=>
(printout t "Date Dopey" crlf))
CLIPS>
(deffacts duck
(bachelor Dicky)
(bachelor Dopey)
(bachelor Dopey Mallard)
(bachelor Dinky Dopey)
(bachelor Dopey Dinky Mallard))
CLIPS> (reset)
CLIPS> (run)
Date Dopey
Date Dopey
Date Dopey
CLIPS>
```

I caratteri jolly hanno un altro uso importante perché possono essere allegati a un campo simbolico per creare una variabile come ?x, $?x, ?name o $?name. La variabile può

essere una **variabile a campo singolo** o una **variabile multicampo** a seconda che un "?" o "$?" viene utilizzato sul lato sinistro (LHS). Si noti che sulla destra (RHS) viene utilizzato solo un ?x, dove la "x" può essere qualsiasi nome di variabile. Si può pensare a "$" come a una funzione il cui argomento è un carattere jolly a campo singolo o una variabile a campo singolo e restituisce rispettivamente un carattere jolly multicampo o una variabile multicampo.

Come esempio di variabile multicampo, la seguente versione della regola stampa anche i campi del nome del fatto corrispondente perché una variabile è equiparata ai campi del nome che corrispondono:

```
CLIPS>
(defrule dating-ducks
(bachelor Dopey $?name)
=>
(printout t "Date Dopey " ?name crlf))
CLIPS> (reset)
CLIPS> (run)
Date Dopey (Dinky Mallard)
Date Dopey (Mallard)
Date Dopey ()
CLIPS>
```

Come si vede, sul lato destro (LHS), il pattern multicampo è *$?name* ma è *?name* se usato come variabile sul lato destro (RHS). Con enter e run, si vedranno i nomi di tutti i Dopey idonei. Il carattere jolly multicampo si occupa di un numero qualsiasi di campi. Inoltre, notare che i valori multicampo vengono restituiti racchiusi tra parentesi.

Supponiamo che si voglia una corrispondenza di tutte le papere che hanno un Dopey da qualche parte nel loro nome, non necessariamente come nome. La seguente versione della regola abbinerebbe [match] tutti i fatti che contengono un Dopey e quindi stamperebbe i nomi:

```
CLIPS>
(defrule dating-ducks
(bachelor $?first Dopey $?last)
=>
(printout t "Date "
?first
" Dopey "
?last crlf))
CLIPS> (reset)
CLIPS> (run)
Date () Dopey (Dinky Mallard)
Date (Dinky) Dopey ()
Date () Dopey (Mallard)
Date () Dopey ()
CLIPS>
```

Il pattern corrisponde a *qualsiasi* nome che contenga Dopey ovunque al suo interno.

È possibile combinare caratteri jolly a campo singolo e multicampo. Per esempio, il pattern

```
(bachelor ? $? Dopey ?)
```

significa che il nome e il cognome possono essere qualsiasi cosa e che il nome immediatamente precedente all'ultimo deve essere Dopey. Questo pattern richiede inoltre che il fatto corrispondente abbia *almeno quattro* campi, poiché il campo "$?" corrisponde a *zero* o più campi e tutti gli altri devono corrispondere *esattamente* a quattro.

Sebbene in molti casi le variabili multicampo possano essere essenziali per il pattern matching, il loro utilizzo eccessivo può causare molta inefficienza a causa dell'aumento dei requisiti di memoria e dell'esecuzione più lenta.

Come regola generale di stile, si dovrebbe usare $? solo quando non si conosce la lunghezza dei campi. Non usare $? semplicemente per comodità di digitazione.

Lo scapolo ideale

Le variabili utilizzate nei pattern hanno una proprietà importante e utile, che può essere definita come segue.

La prima volta che una variabile viene vincolata [bound], mantiene quel valore solo all'interno della regola, sia su LHS che su RHS, a meno che non venga modificato su RHS.

Ad esempio, nella regola seguente

```
(defrule bound
(number-1 ?num)
(number-2 ?num)
=>)
```

Se ci sono alcuni fatti

```
f-1     (number-1 0)
f-2     (number-2 0)
f-3     (number-1 1)
f-4     (number-2 1)
```

allora la regola può essere attivata solo dalla coppia f-1, f-2, e dall'altra coppia f-3, f-4. Cioè, il fatto f-1 non può corrispondere a f-4 perché quando ?num è [bound] a 0 nel primo pattern, anche il valore di ?num nel secondo pattern *deve* essere 0. Allo stesso modo, quando ?num è [bound] a 1 nel primo pattern, il valore di ?num nel secondo pattern *deve* essere 1. Si noti che la regola verrà attivata *due volte* da questi quattro fatti: un'attivazione per la coppia f-1, f-2 e l'altra attivazione per la coppia f-3, f-4.

Come esempio più pratico, inserire la seguente regola. Si noti che la stessa variabile, ?name, viene utilizzata in entrambi i modelli. Prima di eseguire un (reset) e un (run), inserire anche un comando (watch all) in modo da poter vedere cosa succede durante l'esecuzione.

```
CLIPS> (clear)
CLIPS>
(defrule ideal-duck-bachelor
(bill big ?name)
(feet wide ?name)
=>
(printout t "The ideal duck is "
?name crlf))
CLIPS>
(deffacts duck-assets
(bill big Dopey)
(bill big Dorky)
(bill little Dicky)
(feet wide Dopey)
(feet narrow Dorky)
(feet narrow Dicky))
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (reset)
==> f-1     (bill big Dopey)
==> f-2     (bill big Dorky)
==> f-3     (bill little Dicky)
==> f-4     (feet wide Dopey)
==> Activation 0      ideal-duck-bachelor: f-1,f-4
==> f-5     (feet narrow Dorky)
==> f-6     (feet narrow Dicky)
CLIPS> (run)
The ideal duck is Dopey
CLIPS>
```

Quando il programma viene eseguito, il primo pattern corrisponde [match] a Dopey e Dorky poiché entrambi hanno grossi conti. La variabile ?name è associata a ciascun nome. Quando CLIPS tenta di soddisfare il secondo pattern della regola, solo la variabile ?name [bound] a Dopey soddisfa anche il secondo pattern di (feet wide).

L'anatra fortunata

Nella vita si verificano molte situazioni in cui è saggio fare le cose in modo sistematico. In questo modo, se le aspettative non si realizzano, si può riprovare sistematicamente (come il noto algoritmo per trovare il coniuge perfetto sposandosi più e più volte).

Un modo per organizzarsi è tenere una lista. (Nota: se *veramente* si vogliono impressionare le persone, mostrare loro la liste dei propri elenchi). Nel nostro caso, terremo un elenco di papere scapole, con la prospettiva più probabile di matrimonio davanti. Una volta identificato la papera scapola ideale, lo inseriremo in cima alla lista come l'anatra fortunata.

Il seguente programma mostra come ciò può essere fatto aggiungendo un paio di regole alla regola dello scapolo ideale.

```
(defrule ideal-duck-bachelor
(bill big ?name)
(feet wide ?name)
=>
(printout t "The ideal duck is "
?name crlf)
(assert (move-to-front ?name)))

(defrule move-to-front
?move-to-front <- (move-to-front ?who)
?old-list <-
(list $?front ?who $?rear)
=>
(retract ?move-to-front ?old-list)
(assert (list ?who ?front ?rear))
(assert (change-list yes)))

(defrule print-list
?change-list <- (change-list yes)
(list $?list)
=>
(retract ?change-list)
(printout t "List is : " ?list crlf))

(deffacts duck-bachelor-list
(list Dorky Dinky Dicky))
(deffacts duck-assets
(bill big Dicky)
(bill big Dorky)
(bill little Dinky)
(feet wide Dicky)
(feet narrow Dorky)
(feet narrow Dinky))
```

La lista originale viene data nel deffacts duck-bachelor-list. Una volta eseguito il programma, fornirà un nuovo elenco di probabili candidati.

```
CLIPS> (unwatch all)
CLIPS> (reset)
CLIPS> (run)
The ideal duck is Dicky
List is : (Dicky Dorky Dinky)
CLIPS>
```

Notare l'asserzione (change-list yes) nella regola move-to-front. Senza questa asserzione, la regola printlist si attiverebbe sempre sulla lista originale. Questa affermazione è un esempio di **control fact** creato per controllare l'attivazione di un'altra regola. I fatti di controllo sono molto importanti per controllare l'attivazione di determinate regole e si deve studiare attentamente questo esempio per capire perché viene utilizzato. Un altro metodo di controllo sono i moduli, come discusso nel *CLIPS Reference Manual*.

La regola move-to-front rimuove la vecchia lista e afferma il nuovo elenco. Se il vecchio elenco non venisse ritirato [retracted], ci sarebbero due attivazioni in agenda per la regola print-list, ma ne verrebbe attivata solo una. Ne verrà attivato solo uno perché la regola print-list rimuove il fatto di controllo richiesto per l'altra attivazione della stessa regola. Non si saprebbe in anticipo quale verrà attivato, quindi il vecchio elenco potrebbe essere stampato invece del nuovo elenco.

Capitolo 5

Farlo con stile

Oggi stile, domani chissà

In questo capitolo conosceremo la parola chiave *deftemplate*, che sta per **define template**. Questa funzionalità può aiutare a scrivere regole i cui pattern hanno una struttura ben definita.

Mr. Wonderful

Deftemplate è analogo alla definizione di una struttura in C. Cioè, deftemplate definisce un gruppo di campi correlati in uno schema simile al modo in cui una struttura C è un gruppo di dati correlati. Un deftemplate è una lista di nomi di campi detti **slot**. Deftemplate consente l'accesso per nome anziché specificando l'ordine dei campi. Deftemplate contribuisce ad un buon stile nei programmi dei sistemi esperti ed è un prezioso strumento di ingegneria del software.

Uno *slot* è un nome per un **single-slot** o un **multislot**. Un single-slot o semplicemente *slot* contiene esattamente un campo mentre uno multislot contiene zero o più campi. In un deftemplate è possibile utilizzare qualsiasi numero di slot singoli o multislot. Per scrivere uno slot, fornire il nome del campo (attribute) seguito dal valore del campo. Notare che un multislot con un valore non è strettamente uguale a uno slot singolo. Per analogia, si pensi a un armadio (il multislot) che può contenere stoviglie. Una credenza con un piatto non è la stessa cosa di un piatto (single-slot). Tuttavia, il valore di uno slot singolo (o variabile) può corrispondere a un multislot (o multislot variabile) che ha un campo.

Come esempio di relazione deftemplate, considerare gli attributi di un'anatra che potrebbe essere considerata una buona prospettiva matrimoniale:

name "Dopey Wonderful"
assets rich
age 99

Un deftemplate può essere definito per la relazione *prospect* come segue, dove gli spazi e i commenti vengono utilizzati per la leggibilità e la spiegazione.

```
; name of deftemplate relation
(deftemplate prospect
; optional comment in quotes
"vital information"
; name of field
(slot name
; type of field
(type STRING)
; default value of field name
(default ?DERIVE))
; name of field
(slot assets
; type of field
(type SYMBOL)         ; default value of field assets
(default rich))
; name of field
(slot age
; type. NUMBER can be
; INTEGER or FLOAT
(type NUMBER)
; default value of field age
(default 80)))
```

In questo esempio, i componenti di deftemplate sono strutturati come:

- Un nome di relazione deftemplate
- Attributi detti *campi*
- Il tipo del campo, che può essere uno qualsiasi dei tipi consentiti: simbolo, stringa, numero e altri.
- Il valore di default per il valore del campo

Questo particolare deftemplate ha tre slot singolo chiamati *name*, *assets* e *age*.

I **valori di default di deftemplate** vengono inseriti da CLIPS quando viene eseguito un (reset) se non sono definiti valori espliciti. Per esempio, inserire il deftemplate per il potenziale cliente dopo un comando (clear) e asserirlo come mostrato.

```
CLIPS> (assert (prospect))
<Fact-1>
CLIPS> (facts)
f-1     (prospect (name "") (assets rich) (age 80))
For a total of 1 fact.
CLIPS>
```

Come si vede, CLIPS ha inserito il valore di default della stringa nulla, "", per il campo name poiché quello è il valore di default per una STRINGA. Allo stesso modo, anche i default di assets e age sono stati inseriti da CLIPS. Tipi diversi hanno simboli diversi di default come la stringa nulla, "", per STRING; il numero intero 0 per INTEGER; il float 0.0 per FLOAT; e così via. La parola chiave ?DERIVE seleziona il tipo appropriato di vincolo per quello slot, ad esempio, la stringa nulla, "", per uno slot di tipo STRING.

È possibile impostare in modo esplicito i valori del campo, come illustrato nell'esempio seguente.

```
CLIPS>
(assert (prospect (age 99)
(name "Dopey")))
<Fact-2>
CLIPS> (facts)
f-1     (prospect (name "") (assets rich) (age 80))
f-2     (prospect (name "Dopey") (assets rich) (age 99))
For a total of 2 facts.
CLIPS>
```

Notare che l'ordine in cui vengono digitati i campi non ha importanza poiché si tratta di nomi di campi.

Nel deftemplate, è importante rendersi conto che NUMBER *non* è un tipo di campo primitivo come simbolo, stringa, intero e float. NUMBER è in realtà un tipo composto che può essere intero *o* float. Viene utilizzato nei casi in cui all'utente non interessa il tipo di numeri memorizzati. Un'alternativa a NUMBER sarebbe specificare i tipi come segue.

```
(slot age
(type INTEGER FLOAT)
(default 80)))
```

Bye-Bye

In generale un deftemplate con N slot ha la seguente struttura generale::

```
(deftemplate <name>
(slot-1)
(slot-2)
.
.
.
(slot-N))
```

In un deftemplate, i valori degli attributi possono essere specificati in modo più preciso rispetto a un valore semplice come *80* o *rich*. Per esempio, in questo deftemplate, un **tipo** di valore è specificato.

I valori dei campi possono essere specificati elencandoli esplicitamente o fornendo un intervallo di valori. Gli *allowed-symbols* possono essere qualsiasi tipo primitivo come SYMBOL, STRING, INTEGER, FLOAT e così via. Alcuni esempi di valori enumerati deftemplate sono mostrati di seguito:

allowed-symbols	rich filthy-rich loaded
allowed-strings	"Dopey" "Dorky" "Dicky"
allowed-numbers	1 2 3 4.5 -2.001 1.3e-4
allowed-integers	-100 53
allowed-floats	-2.3 1.0 300.00056
allowed-values	"Dopey" rich 99 1.e9

Non ha senso specificare sia un intervallo numerico che valori consentiti per lo stesso campo deftemplate. Ad esempio, se si specifica (allowed-integers 1 4 8), ciò contraddice la specifica di un intervallo compreso tra 1 e 10 tramite (range 1 10). Se i numeri sono

sequenziali, come 1, 2, 3, è possibile specificare un intervallo che corrisponda esattamente: (range 1 3). Tuttavia, l'intervallo sarebbe ridondante rispetto alla specifica degli interi consentiti. Pertanto, l'intervallo e i valori consentiti sono *mutualmente esclusivi*. Cioè, se si specifica un intervallo, non si possono specificare i valori consentiti e *viceversa*. In generale, l'attributo range non può essere utilizzato insieme a allowed-values, allowed-numbers, allowed-integers o allowed-floats.

Senza le informazioni facoltative, segue il deftemplate e una regola che lo utilizza.

```
CLIPS> (clear)
CLIPS>
; name of deftemplate
(deftemplate prospect
; name of field
(slot name
; default value of field name
(default ?DERIVE))
; name of field
(slot assets
; default value of field assets
(default rich))
; name of field
(slot age
; default value of field age
(default 80)))
CLIPS>
(defrule matrimonial_candidate
(prospect (name ?name)
(assets ?net_worth)
(age ?months))
=>
(printout t "Prospect: "
?name crlf
?net_worth crlf
?months " months old"
crlf))
CLIPS>
(assert
(prospect (name "Dopey Wonderful")
(age 99)))
<Fact-1>
CLIPS> (run)
Prospect: Dopey Wonderful
rich
99 months old
CLIPS>
```

Si noti che il valore di default di *rich* è stato utilizzato per Dopey poiché il campo asset non è stato specificato nel comando assert.

Se al campo asset viene assegnato un valore specifico come *poor*, il valore specificato per *assets* di *poor* sovrascrive il valore di default di *rich* come mostrato nel seguente esempio sul nipote meschino di Dopey.

```
CLIPS> (reset)
CLIPS>
(assert
(prospect (name "Dopey Notwonderful")
(assets poor)
(age 95)))
<Fact-1>
CLIPS> (run)
Prospect: "Dopey Notwonderful"
poor
95 months old
CLIPS>
```

Un pattern deftemplate è utilizzabile proprio come qualsiasi pattern normale. Ad esempio, la seguente regola eliminerà le prospects (prospettive) indesiderabili.

```
CLIPS> (undefrule matrimonial_candidate)
CLIPS>
(defrule bye-bye
?bad-prospect <-
(prospect (assets poor)
(name ?name))
=>
(retract ?bad-prospect)
(printout t "bye-bye " ?name crlf))
CLIPS> (reset)
CLIPS>
(assert
```

```
(prospect (name "Dopey Wonderful")
(assets rich)))
<Fact-1>
CLIPS>
(assert
(prospect (name "Dopey Notwonderful")
(assets poor)))
<Fact-2>
CLIPS> (run)
bye-bye Dopey Notwonderful
CLIPS>
```

Senza vincoli

Si noti che finora sono stati utilizzati solo campi singoli per i pattern negli esempi. Cioè, i valori dei campi per *name*, *assets* e *age*, erano tutti valori singoli. In alcuni tipi di regole, si potrebbero volere più campi. Deftemplate consente l'uso di più valori in un *multislot*.

Come esempio di multislot, supponiamo di voler trattare il nome della relazione *prospect* come campi multipli. Ciò fornirebbe maggiore flessibilità nell'elaborazione di prospects poiché qualsiasi parte del nome potrebbe essere abbinata al pattern. Di seguito è mostrata la definizione di deftemplate che utilizza multislot e la regola rivista per la corrispondenza del pattern su più campi. Si noti che ora viene utilizzato un pattern multislot, $?name, per far corrispondere [match] tutti i campi che compongono "name". Per comodità viene fornito anche un (deffacts).

```
CLIPS> (clear)
CLIPS>
(deftemplate prospect
(multislot name
(type SYMBOL)
(default ?DERIVE))
(slot assets
(type SYMBOL)
(allowed-symbols
poor rich wealthy loaded)
(default rich))
(slot age
(type INTEGER)       ; The older
(range 80 ?VARIABLE) ; the
(default 80)))       ; better!!!
CLIPS>
(defrule happy_relationship
(prospect (name $?name)
(assets ?net_worth)
(age ?months))
=>
(printout t "Prospect: "
; Note: not $?name
?name crlf
?net_worth crlf
?months " months old"
crlf))
CLIPS>
(deffacts duck-bachelor
(prospect (name Dopey Wonderful)
(assets rich)
(age 99)))
CLIPS> (reset)
CLIPS> (run)
Prospect: (Dopey Wonderful)
rich 99 months old
CLIPS>
```

Nell'output, le parentesi attorno al nome di Dopey vengono inserite da CLIPS per indicare che si tratta di un valore multislot. Se si confronta l'output di questa versione multislot con quella a slot singolo, si vedrà che le virgolette attorno a "Dopey Wonderful" sono scomparse. Il nome *slot* non è una stringa nella versione multislot, quindi CLIPS tratta il nome come due campi indipendenti, *Dopey* e *Wonderful*.

Cosa c'è in name

Deftemplate semplifica notevolmente l'accesso a un campo specifico in un pattern perché il campo desiderato può essere identificato dal nome dello slot. L'azione **modify** è

utilizzabile per modificare un fatto in un'unica azione specificando uno o più slot del template da modificare.

Per esempio, si considerino le seguenti regole che mostrano cosa succede quando duck-bachelor Dopey Wonderful perde tutti le sue fish comprando poster di Paperino e "banana fishsplit" per la sua nuova papera, Dixie.

```
CLIPS> (undefrule *)
CLIPS>
(defrule make-bad-buys
?prospect <- (prospect (name $?name)
(assets rich)
(age ?months))
=>
(printout t "Prospect: "
; Note: not $?name
?name crlf
"rich" crlf
?months " months old"
crlf crlf)
(modify ?prospect (assets poor)))
CLIPS>
(defrule poor-prospect
?prospect <- (prospect (name $?name)
(assets poor)
(age ?months))
=>
(printout t "Ex-prospect: "
; Note: not $?name
?name crlf
poor crlf
?months " months old"
crlf crlf))
CLIPS>
(deffacts duck-bachelor
(prospect (name Dopey Wonderful)
(assets rich)
(age 99)))
CLIPS> (reset)
CLIPS> (run)
Prospect: (Dopey Wonderful)
rich
99 months old

Ex-prospect: (Dopey Wonderful)
poor 99 months old

CLIPS>
```

Eseguendo un comando (facts) come segue, si vedrà che il fatto f-1 originariamente corrispondente a (prospect (assets rich) (age 99) (name Dopey Wonderful)) è stato modificato e lo slot delle risorse è stato impostato a povero [poor].

```
CLIPS> (facts)
f-1     (prospect (name Dopey Wonderful)
(assets poor) (age 99))
For a total of 1 fact.
CLIPS>
```

La regola make-bad-buys viene attivata da un ricco prospect come specificato dallo slot assets. Questa regola modifica gli assets in *poor* utilizzando l'azione di modifica. Notare che lo slot *assets* è accessibile per nome. Senza un deftemplate sarebbe necessario enumerare tutti i campi tramite variabili singole o utilizzando un carattere jolly, il che è meno efficiente. Lo scopo della regola poor-prospect è semplicemente quello di stampare i prospetti poveri, dimostrando così che la regola make-bad-investments ha effettivamente modificato gli asset.

Capitolo 6

Essere funzionali

La funzionalità è l'inverso dello stile

In questo capitolo si vedranno funzioni più potenti per la corrispondenza dei pattern e alcune che sono molto utili con le variabili multicampo. Si vedrà anche come vengono eseguiti i calcoli numerici.

Non il mio vincolo

Riconsideriamo il problema di progettare un sistema esperto per aiutare un robot ad attraversare una strada. Una regola che si sarebbe dovuta seguire.

```
(defrule green-light
(light green)
=>
(printout t "Walk" crlf))
```

Un'altra regola riguarderebbe il caso del semaforo rosso.

```
(defrule red-light
(light red)
=>
(printout t "Don't walk" crlf))
```

Una terza regola riguarderebbe il caso in cui un segnale stradale dica di non attraversare. Ciò avrebbe la precedenza sul semaforo verde.

```
(defrule walk-sign
(walk-sign-says dont-walk)
=>
(printout t "Don't walk" crlf))
```

Le regole precedenti sono semplificate e non coprono tutti i casi come il guasto del semaforo. Ad esempio, cosa fa il robot se la luce è rossa o gialla e il segnale di attraversamento dice di attraversare?

Un modo per gestire questo caso è utilizzare un **vincolo di campo** per limitare i valori che un pattern può avere a sinistra (LHS). Il vincolo del campo agisce come un vincolo sui pattern.

Un tipo di vincolo di campo è chiamato **connective constraint** (vincolo connettivo). Esistono tre tipi di vincoli connettivi. Il primo è detto **vincolo ~**. Il suo simbolo è la **tilde**, "**~**". Il vincolo ~ agisce sull'unico valore che lo segue immediatamente e *non* consentirà quel valore.

Come semplice esempio di vincolo ~, supponiamo di voler scrivere una regola che stampi "Non camminare" se il semaforo non è verde. Un approccio potrebbe essere quello di scrivere regole per ogni possibile condizione del semaforo, inclusi tutti i possibili malfunzionamenti: giallo, rosso, giallo lampeggiante, rosso lampeggiante, verde lampeggiante, giallo lampeggiante, giallo lampeggiante e rosso lampeggiante e così via. Tuttavia, un approccio molto più semplice consiste nell'utilizzare il vincolo ~ come mostrato nella regola seguente:

```
(defrule walk
(light ~green)
=>
(printout t "Don't walk" crlf))
```

Utilizzando il vincolo ~, questa regola svolge il lavoro di molte altre regole che richiedono la specifica di ciascuna condizione del semaforo.

Prudenza

Il secondo vincolo connettivo è il **vincolo barra**, "**|**". Il vincolo connettivo "|" viene utilizzato per consentire la corrispondenza [match] di qualsiasi gruppo di valori.

Per esempio, supponiamo di volere una regola che stampi "Be cautious" se il semaforo era giallo o lampeggiava in giallo. L'esempio seguente mostra come farlo utilizzando il vincolo "|".

```
CLIPS> (clear)
CLIPS>
(defrule cautious
(light yellow|blinking-yellow)
=>
(printout t "Be cautious" crlf))
CLIPS> (assert (light yellow))
<Fact-1>
CLIPS> (assert (light blinking-yellow))
<Fact-2>
CLIPS> (agenda)
0      cautious: f-2
0      cautious: f-1
For a total of 2 activations.
CLIPS>
```

E via

Il terzo tipo di vincolo connettivo è il **vincolo connettivo &**. Il simbolo del vincolo connettivo & è la **e commerciale**, "**&**". Il vincolo & forza i vincoli connessi a corrispondere [match] in unione, come si vedrà negli esempi seguenti. Il vincolo & normalmente viene utilizzato solo con gli altri vincoli, altrimenti non è di grande utilità pratica. Ad esempio, supponiamo di voler avere una regola che verrà attivata da un fatto giallo o giallo lampeggiante. È abbastanza semplice: basta usare il vincolo connettivo | come fatto nell'esempio precedente. Ma supponiamo che si voglia identificare anche il color? del semaforo

La soluzione è quella di associare una variabile al colore corrispondente [match] utilizzando "&" e poi stampare la variabile. È qui che "&" è utile, come mostrato di seguito.

```
(defrule cautious
(light ?color&yellow|blinking-yellow)
=>
(printout t
"Be cautious because light is "
?color crlf))
```

La variabile ?color sarà associata [bound] al colore corrispondente al campo giallo|giallo lampeggiante.

La "&" è utile anche con la "~". Supponiamo, ad esempio, di volere una regola che si attivi quando il semaforo non è né giallo né rosso.

```
(defrule not-yellow-red    (light ?color&~red&~yellow)
=>
(printout t "Go, since light is "
?color crlf))
```

È elementare

Oltre a trattare fatti simbolici, CLIPS può anche eseguire calcoli numerici. Tuttavia, si dovrebbe tenere presente che un linguaggio di sistema esperto come CLIPS non è progettato principalmente per il calcolo numerico. Sebbene le funzioni matematiche di CLIPS siano molto potenti, in realtà sono pensate per la modifica dei numeri su cui ragiona il programma applicativo. Altri linguaggi come FORTRAN sono migliori per l'elaborazione dei numeri in cui viene fatto poco o nessun ragionamento simbolico. Si troverà utile la capacità di calcolo di CLIPS in molte applicazioni.

CLIPS fornisce le funzioni aritmetiche e matematiche di base +, /, *, -, div, max, min, abs, float e integer. Per ulteriori dettagli, consultare il *CLIPS Reference Manual*.

Le espressioni numeriche sono rappresentate in CLIPS secondo lo stile LISP. Sia in LISP che in CLIPS, un'espressione numerica che normalmente verrebbe scritta come 2 + 3 deve essere scritta nella **forma prefissa**, (+ 2 3). Nella forma prefissa di CLIPS, la funzione precede gli argomenti e le parentesi devono racchiudere l'espressione numerica. Il modo consueto di scrivere espressioni numeriche è chiamato **forma infissa** perché le funzioni matematiche sono fissate tra gli **argomenti**.

Le funzioni possono essere utilizzate a sinistra (LHS) e a destra (RHS). Ad esempio, quanto segue mostra come viene utilizzata l'operazione aritmetica di addizione sulla destra (RHS) di una regola per asserire un fatto contenente la somma di due numeri ?x e ?y. Notare che i commenti sono in notazione infissa solo per chiarezza poiché l'infissa non può essere valutata da CLIPS.

```
CLIPS> (clear)
CLIPS>
(defrule addition
(numbers ?x ?y)
=>
; Add ?x + ?y
(assert (answer-plus (+ ?x ?y))))
CLIPS> (assert (numbers 2 3))
<Fact-1>
CLIPS> (run)
CLIPS> (facts)
f-1     (numbers 2 3)
f-2     (answer-plus 5)
For a total of 2 facts.
CLIPS>
```

Una funzione può essere utilizzata sul lato sinistro (LHS) se un **segno uguale**, **=**, viene utilizzato per dire a CLIPS di valutare la seguente espressione invece di usarla letteralmente per la il pattern matching. L'esempio seguente mostra come l'ipotenusa viene calcolata sul lato sinistro (LHS) e utilizzata per la corrispondenza [match] con alcuni articoli in stock. La funzione **elevamento a potenza**, "******", viene utilizzata per elevare al quadrato i valori x e y. Il primo argomento dell'elevamento a potenza è il numero che deve essere elevato alla potenza del secondo argomento.

```
CLIPS> (clear)
CLIPS>
(deffacts database
(stock A 2.0)
(stock B 5.0)
(stock C 7.0)) CLIPS>
(defrule addition
(numbers ?x ?y)
; Hypotenuse
(stock ?ID =(sqrt (+ (** ?x 2)
(** ?y 2))))
=>
(printout t "Stock ID=" ?ID crlf))
CLIPS> (reset)
CLIPS> (assert (numbers 3 4))
<Fact-4>
; Stock ID matches
; hypotenuse calculated
CLIPS> (run)
Stock ID=B
CLIPS>
```

Argomentazioni estese

Gli argomenti in un'espressione numerica possono essere estesi oltre due per molte funzioni matematiche. La stessa sequenza di calcoli aritmetici viene eseguita per più di due argomenti. L'esempio seguente illustra come vengono utilizzati tre argomenti. La valutazione procede da sinistra a destra. Prima di inserirli, tuttavia, si potrebbe eseguire un (clear) per eliminare eventuali vecchi fatti e regole.

```
(defrule addition    (numbers ?x ?y ?z)
=>
; ?x + ?y + ?z
(assert (answer-plus (+ ?x ?y ?z))))
```

Inserire il programma precedente e asserire (numbers 2 3 4). Dopo l'esecuzione, si vedranno i seguenti fatti. Notare che il fact-indices potrebbero essere diversi se è stato eseguito un (reset) anziché un (clear) prima di caricare questo programma.

```
CLIPS> (facts)
f-1     (numbers 2 3 4)
f-2     (answer-plus 9)
For a total of 2 facts.
CLIPS>
```

L'equivalente infissa di un'espressione CLIPS con più argomenti può essere espressa come

```
arg [function arg]
```

dove le parentesi quadre indicano che possono esserci più termini.

Oltre alle funzioni matematiche di base, CLIPS dispone di **funzioni matematiche estese** tra cui trigonometria, iperbolica e così via. Per un elenco completo, consultare il

CLIPS Reference Manual. Queste sono chiamate funzioni *Extended Math* perché non sono considerate funzioni matematiche di base come “+”, “-”, ecc.

Risultati misti

Nel gestire le espressioni, CLIPS cerca di mantenere la stessa modalità degli argomenti. Per esempio,

```
; both integer arguments
; give integer result
CLIPS> (+ 2 2)
4
; both floating-point arguments
; give floating-point result
CLIPS> (+ 2.0 2.0)
4.0
; mixed arguments
; give floating-point result
CLIPS> (+ 2 2.0)
4.0
CLIPS>
```

Si noti che nell'ultimo caso di argomenti misti, CLIPS converte il risultato nel tipo a virgola mobile a doppia precisione standard.

È possibile convertire in modo esplicito un tipo in un altro utilizzando le funzioni float e integer, come dimostrato negli esempi seguenti.

```
; convert integer
; to float
CLIPS> (float (+ 2 2))
4.0
; convert float
; to integer
CLIPS> (integer (+ 2.0 2.0))
4
CLIPS>
```

Le parentesi vengono utilizzate per specificare esplicitamente l'ordine di valutazione dell'espressione, se lo si desidera. Nell'esempio di ?x + ?y * ?z, il modo infisso consueto per valutarlo è moltiplicare ?y per ?z e poi aggiungere il risultato a ?x. Tuttavia, in CLIPS, è necessario scrivere esplicitamente la precedenza se si desidera questo ordine di valutazione, come segue.

```
(defrule mixed-calc
(numbers ?x ?y ?z)
=>
; ?y * ?z + ?x
(assert (answer (+ ?x (* ?y ?z)))))
```

In questa regola, viene valutata per prima l'espressione tra parentesi più interna; quindi ?y viene moltiplicato per ? z. Il risultato viene aggiunto a ?x.

Bound Bachelors

L'analogo all'assegnazione di un valore a una variabile sul lato sinistro (LHS) tramite pattern matching è col **binding** di un valore a una variabile sul lato destro (RHS) utilizzando la **bind function**. È conveniente associare le variabili sul lato destro (RHS) se gli stessi valori verranno utilizzati ripetutamente.

Come semplice esempio di calcolo matematico, eseguiamo il bind prima del risultato a una variabile e poi stampiamo la **variabile bound**.

```
CLIPS> (clear)
CLIPS>
(defrule addition
(numbers ?x ?y)
=>
(assert (answer (+ ?x ?y)))
(bind ?answer (+ ?x ?y))
(printout t "answer is "
?answer crlf))
CLIPS> (assert (numbers 2 2))
<Fact-1>
CLIPS> (run)
answer is 4
CLIPS> (facts)
f-1     (numbers 2 2)
f-2     (answer 4)
For a total of 2 facts.
CLIPS>
```

Il (bind) può essere utilizzato anche sul lato destro (RHS) per il bind di valori singoli o multicampo a una variabile. (bind) viene utilizzato per il bind di zero, uno o più valori a una variabile *senza l'operatore "$"*. Ricordare che sul lato sinistro (LHS) si può creare un pattern multicampo solo utilizzando l'operatore "$" su un campo, come "$? x". Tuttavia, "$" non è necessario sul lato destro (RHS) perché gli argomenti di (bind) dicono esplicitamente a CLIPS esattamente quanti valori associare. In effetti, il "$" è un'appendice inutile sulla destra (RHS).

La seguente regola illustra alcuni bind di variabili sulla destra (RHS). La **funzione valore multicampo**, **create$**, viene utilizzata per creare un valore multicampo. La sua sintassi generale è la seguente.

```
(create$ <arg1> <arg2> ... <argN>)
```

dove qualsiasi numero di argomenti può essere accodato insieme per creare un valore multicampo. Questo valore multicampo, o un valore a campo singolo, può quindi essere associato [bound] a una variabile come mostrato nelle azioni sulla destra (RHS) della regola seguente.

```
CLIPS> (clear)
CLIPS>
(defrule bind-values-demo
=>
(bind ?duck-bachelors
(create$ Dopey Dorky Dinky))
(bind ?happy-bachelor-mv
(create$ Dopey))
(bind ?none (create$))
(printout t
"duck-bachelors "
?duck-bachelors crlf
"duck-bachelors-no-() "
(implode$ ?duck-bachelors) crlf
"happy-bachelor-mv "
?happy-bachelor-mv crlf
"none " ?none crlf))
CLIPS> (reset)
CLIPS> (run)
duck-bachelors (Dopey Dorky Dinky)
duck-bachelors-no-() Dopey Dorky Dinky
happy-bachelor-mv (Dopey)
none ()
CLIPS>
```

Fare le cose proprie

Proprio come gli altri linguaggi, CLIPS permette di definire le funzioni con **deffunction**. La deffunction è conosciuta a livello globale, il che fa risparmiare la fatica di ripetere sempre le stesse azioni.

Le deffunctions contribuiscono anche alla leggibilità. Si può chiamare una deffunction proprio come qualsiasi altra funzione. Una deffunction può essere utilizzata anche come argomento di un'altra funzione. Un (printout) è utilizzabile *ovunque* in una deffunction anche se non è l'ultima azione perché la stampa è un effetto collaterale della chiamata alla funzione (printout).

La sintassi generale di una deffunzione è mostrata di seguito.

```
(deffunction <function-name>
[optional comment]
; argument list. Last one may
; be optional multifield arg.
(?arg1 ?arg2 ... ?argM [$?argN])
; action1 to action(K-1)
; do not return a value
; only last action returned
(<action1>
<action2>
...
<action(K-1)>
<actionK>)
```

Gli *?arg* sono **argomenti fittizi**, il che significa che i nomi degli argomenti non entreranno in conflitto con i nomi delle variabili in una regola se sono uguali. Il termine argomento fittizio è talvolta chiamato **parametro** in altri libri.

Sebbene ciascuna azione possa aver restituito valori dalle chiamate alle funzioni all'interno dell'azione, questi vengono bloccati dalla deffunction in modo che non vengano restituiti all'utente. La deffunction restituirà solo il valore dell'*ultima* azione, <actionK>. Questa azione può essere una funzione, una variabile o una costante.

Di seguito è riportato un esempio di come viene definita una deffunction per calcolare l'ipotenusa e poi viene utilizzata in una regola. Anche se i nomi delle variabili nella regola sono gli stessi degli argomenti fittizi, non c'è conflitto. Ecco perché sono *fittizi*, perché non significano nulla.

```
CLIPS> (clear)
CLIPS>
(deffunction hypotenuse   ; name
; dummy arguments
(?a ?b)
; action
(sqrt(+ (* ?a ?a) (* ?b ?b))))
CLIPS>
(defrule calculate-hypotenuse
(dimensions ?base ?height)
=>
(printout t "Hypotenuse="
(hypotenuse ?base ?height)
crlf))
CLIPS> (assert (dimensions 3 4))
<Fact-1>
CLIPS> (run)
Hypotenuse=5.0
CLIPS>
```

Le deffunction sono utilizzabili con valori multicampo, come mostra l'esempio seguente.

```
CLIPS> (clear)
CLIPS>
(deffunction count ($?arg)
(length$ $?arg))
CLIPS> (count 1 2 3 a duck "quacks")
6
CLIPS>
```

Altre funzionalità

Seguono altre funzioni utili. Per ulteriori informazioni, consultare il *CLIPS Reference Manual*.

round: Arrotonda all'intero più vicino. Se esattamente tra due numeri interi, arrotonda all'infinito negativo

integer: Tronca la parte decimale di un numero.

format: Formatta l'output.

list-deffunctions: Elenca tutte le deffunction.

ppdeffunction: Stampa deffunction formattandola.

undeffunction: Elimina una deffunction se non è attualmente in esecuzione e non viene citata altrove. Specificando "*" per <name> si elimina tutto.

length$: Numero di campi o numero di caratteri in una stringa o un simbolo.

nth$: Campo specificato se esiste, altrimenti nil.

member$: Numero del campo se esiste un valore letterale o una variabile, altrimenti FALSE.

subsetp: Restituisce TRUE se un valore multicampo è un sottoinsieme di un altro valore multicampo, altrimenti FALSE.

delete$: Dato un numero di campo, elimina il valore nel campo.

explode$: Ogni elemento stringa viene restituito come parte di un nuovo valore multicampo.

subseq$: Restituisce un intervallo specificato di campi.

replace$: Sostituisce un valore specificato.

Capitolo 7

Come avere il controllo

Da giovane sei controllato dal mondo, Da adulto dovresti controllare il mondo

Finora abbiamo visto la sintassi di base di CLIPS. Ora vedremo come applicare la sintassi a programmi più potenti e complessi. Si vedrà anche anche una nuova sintassi per l'input e come confrontare valori e generare loop.

Iniziamo dalla lettura

Oltre che col matching a pattern, una regola può ottenere informazioni in un altro modo. CLIPS può leggere le informazioni digitate dalla tastiera utilizzando la funzione **read**.

L'esempio seguente mostra come viene utilizzato (read) per immettere i dati. Notare che non è necessario alcun extra (crlf) dopo (read) per posizionare il cursore su una nuova riga. (read) reimposta automaticamente il cursore su una nuova riga.

```
CLIPS> (clear)
CLIPS>
(defrule read-input
=>
(printout t "Name a primary color"
crlf)
(assert (color (read))))
CLIPS>
(defrule check-input
?color <-
(color ?color-read&red|yellow|blue)
=>
(retract ?color)
(printout t "Correct" crlf))
CLIPS> (reset)
CLIPS> (agenda)
0      read-input: *
For a total of 1 activation.
CLIPS> (run)
Name a primary color
red
Correct
CLIPS> (reset)
CLIPS> (run)
Name a primary color
green
CLIPS>    ; No "correct"
```

La regola è progettata per utilizzare l'input da tastiera sul lato destro (RHS), quindi è conveniente attivare la regola non specificando alcun pattern sul lato sinistro (LHS), in modo che venga attivata automaticamente quando si verifica un (reset). Quando l'attivazione della regola read-input viene visualizzata dal comando (agenda), viene stampato un * anziché un identificatore di fatto come f-1. L'* viene utilizzato per indicare che il pattern è soddisfatto, ma non da un fatto specifico.

La funzione (read) non è una funzione generica che leggerà qualsiasi cosa digitata sulla tastiera. Una limitazione è che (read) leggerà solo un campo. Quindi se prova a leggere

```
primary color is red
```

verrà letto solo il primo campo, “primary”. Per leggere con (read) tutto l'input, è necessario racchiuderlo tra virgolette doppie. Naturalmente, una volta che l'input è racchiuso tra virgolette doppie, si tratta di un singolo campo letterale. È quindi possibile accedere alle sottostringhe “primary”, “color”, “is” e “red” con **str-explode** o con le **funzioni sub-string**.

La seconda limitazione di (read) è che non è possibile inserire parentesi a meno che non siano racchiuse tra virgolette doppie. Così come non si può asserire un fatto contenente parentesi, non si può usare (read) per leggere le parentesi direttamente se non come valori letterali.

La funzione **readline** viene utilizzata per leggere più valori finché non viene terminata da un ritorno a capo. Questa funzione legge i dati come una stringa. Per asserire i dati (readline), viene utilizzata una funzione (assert-string) per asserire il fatto nonstring, proprio come immesso da (readline). Segue un esempio di primo livello di (assert-string).

```
CLIPS> (clear)
CLIPS>
(assert-string "(primary color is red)")
<Fact-1>
CLIPS> (facts)
f-1     (primary color is red)
For a total of 1 fact.
CLIPS>
```

Si noti che l'argomento di (assert-string) deve essere una stringa. Di seguito viene mostrato come affermare un fatto di più campi da (readline).

```
CLIPS> (clear)
CLIPS>
(defrule test-readline
=>
(printout t "Enter input" crlf)
(bind ?string (readline))
(assert-string
(str-cat "(" ?string ")")))
CLIPS> (reset)
CLIPS> (run)
Enter input
primary color is red
CLIPS> (facts)
f-1     (primary color is red)
For a total of 1 fact.
CLIPS>
```

Poiché (assert-string) richiede l'asserzione delle parentesi attorno alla stringa, la funzione (str-cat) viene utilizzata per inserirle attorno a ?string.

Sia (read) che (readline) possono essere utilizzati anche per leggere informazioni da un file specificando il nome logico del file come argomento. Per ulteriori informazioni, consultare il *CLIPS Reference Manual*.

Essere efficienti

CLIPS è un linguaggio basato su regole che utilizza un algoritmo di pattern-matching molto efficiente chiamato **Rete Algorithm**, ideato da Charles Forgy della Carnegie-Mellon University per la sua shell OPS. Il termine *Rete* è latino per *net* e descrive l'architettura software del processo di pattern-matching.

È molto difficile fornire regole precise che migliorino sempre l'efficienza di un programma gestito dal Rete Algorithm. Tuttavia, le seguenti dovrebbero essere considerate come linee guida generali che possono aiutare:

1. Inserire prima i pattern più specifici in una regola. I pattern con variabili e caratteri jolly non associati (unbound) dovrebbero trovarsi più in basso nella lista di pattern di regole. Un fatto di controllo dovrebbe essere messo al primo posto nei pattern.
2. I pattern con meno fatti corrispondenti [match] dovrebbero essere visualizzati per primi per ridurre al minimo i match parziali.
3. I pattern che vengono spesso ritirati e asseriti, i **volatile pattern**, dovrebbero essere inseriti per ultimi nell'elenco dei pattern.

Come si vede, queste linee guida sono potenzialmente contraddittorie. Un pattern non specifico può avere poche corrispondenze [match] (linee guida 1 e 2). Dove dovrebbe andare? La linea guida generale è quella di ridurre al minimo le modifiche delle corrispondenze [match] parziali da un ciclo del Motore di Inferenza a quello successivo. Ciò potrebbe richiedere molto impegno da parte del programmatore nel guardare i "match" parziali. Una soluzione alternativa è semplicemente quella di acquistare un computer più veloce o una scheda acceleratrice. Ciò sta diventando sempre più interessante poiché il prezzo dell'hardware scende sempre mentre il prezzo del lavoro umano aumenta sempre più. Poiché CLIPS è progettato per la portabilità, qualsiasi codice sviluppato su una macchina dovrebbe funzionare su un'altra.

Altre funzionalità

L'**elemento condizionale test** fornisce un modo molto potente con cui confrontare numeri, variabili e stringhe su LHS. (test) viene utilizzato come pattern sul lato sinistro (LHS). Una regola verrà attivata solo se il (test) è soddisfatto insieme ad altri pattern.
CLIPS fornisce molte funzioni predefinite. Le funzioni logiche sono:

not: *Not* booleano
and: *And* booleano
or: *Or* booleano

Le funzioni aritmetiche sono:

/: Divisione
*: Moltiplicazione
+: Addizione
-: Sottrazione

Le funzioni di confronto sono:

eq: Uguale (qualsiasi tipo). Confronta tipo e grandezza. **neq**: Non uguale (qualsiasi tipo).
=: Uguale (tipo numerico). Confronta la [magnitude].
<>: Diverso (tipo numerico).
>=: Maggiore o uguale a
>: Maggiore di.
<=: Minore o uguale a.
<: Minore di.

Tutte le funzioni di confronto tranne “eq” e “neq” restituiranno un messaggio di errore se vengono utilizzate per confrontare un numero e un non numero. Se il tipo non è noto in anticipo, è necessario utilizzare le funzioni “eq” e “neq”. La funzione eq controlla la stessa grandezza *e* il tipo dei suoi argomenti mentre la funzione “=" controlla *solo* la grandezza dei suoi argomenti (numerici) e non se ne preoccupa se sono interi o in virgola mobile.

Le **funzioni logiche** di CLIPS sono **and**, **or** e **not**. Possono essere utilizzate nelle espressioni come funzioni booleane. In CLIPS, vero e falso sono rappresentati dai simboli TRUE e FALSE. Si noti che il maiuscolo *deve* essere utilizzato per i valori logici in CLIPS.

Oltre a tutte le funzioni predefinite, si possono scrivere **funzioni esterne** o **funzioni definite dall'utente** in C, Ada o altri linguaggi procedurali e collegarti [link] a CLIPS. Queste funzioni esterne vengono poi utilizzate come qualsiasi funzione predefinita.

CLIPS dà anche la possibilità di specificare un **elemento and condizionale** esplicito, un **elemento or condizionale** e un **elemento condizionale not** a sinistra (LHS). L'assenza di un fatto viene specificata come pattern sul lato sinistro (LHS) utilizzando l'elemento condizionale “not”.

L'alterazione delle nostre informazioni per conformarsi alla realtà si chiama **mantenimento della verità** [truth maintenance]. Cerchiamo cioè di mantenere lo stato della nostra mente in modo da contenere solo informazioni vere in modo da ridurre al minimo i conflitti con il mondo reale.

Sebbene le persone possano farlo abbastanza facilmente (la pratica rende perfetti), è difficile per i computer perché normalmente non sanno quali entità del pattern sono **logicamente dipendenti** da altre entità del pattern. CLIPS ha una funzionalità per supportare il mantenimento della verità che taggherà internamente quelle entità del pattern che dipendono logicamente da altre. Se queste altre entità del pattern vengono ritirate [retracted], CLIPS ritirerà [retract] automaticamente quelle logicamente dipendenti. L'**elemento condizionale logico** utilizza la parola chiave **logical** attorno a un pattern per indicare che le entità del pattern corrispondente [match] forniscono **supporto logico** alle asserzioni sul destra (RHS).

Sebbene il supporto logico funzioni per le asserzioni, *non riasserisce* fatti ritrattati [retracted]. La morale è che, se si perde qualcosa a causa di informazioni errate, non si può recuperarlo (come perdere denaro su consiglio dell'agente di cambio).

CLIPS ha due funzioni per aiutare con il supporto logico. La funzione **dependencies** elenca le corrispondenze [match] parziali da cui un'entità pattern riceve supporto logico, o nessuna se non c'è supporto. La seconda funzione logica è **dependents** che elenca tutte le entità del pattern che ricevono supporto logico da un'entità del pattern [pattern entity].

Il vincolo connettivo utilizza “&”, “|” o “~”. Un altro tipo di vincolo di campo è chiamato **vincolo di predicato** e viene spesso utilizzato per il pattern matching di campi più complessi. Lo scopo di un vincolo sul predicato è quello di vincolare un campo in base al risultato di un'espressione booleana. Se il valore booleano restituisce FALSE, il vincolo non è soddisfatto e il pattern matching fallisce. Si troverà che il vincolo del predicato è molto utile con i pattern numerici.

Una **funzione predicato** è quella che restituisce un valore FALSE o non-FALSE. I **due punti**, “**:**” seguiti da una funzione di predicato sono chiamati **vincolo di predicato**. Il **":"** può essere preceduto da “&”, “|”, o “~” o può stare da solo come nel pattern (fact :(> 2 1)). Viene generalmente utilizzato con il vincolo connettivo & come “**&:**”. L'elenco seguente contiene alcune delle funzioni predicato definite da CLIPS:

(**evenp** <arg>): Controlla se <arg> è un numero pari.
(**floatp** <arg>): Controlla se <arg> è un numero in virgola mobile.
(**integerp** <arg>): Controlla se <arg> è intero.
(**lexemep** <arg>): Controlla se <arg> è un simbolo o una stringa.
(**multifieldp** <arg>): Controlla se <arg> è un valore multicampo.
(**numberp** <arg>): Controlla se <arg> è float o intero.
(**oddp** <arg>): Controlla se <arg> è un numero dispari.
(**pointerp** <arg>): Controlla se <arg> è un indirizzo esterno.
(**stringp** <arg>): Controlla se <arg> è una stringa.
(**symbolp** <arg>): Controlla se <arg> è un simbolo.

Ci sono spesso casi in cui è conveniente avere valori conosciuti globalmente in un sistema esperto. Ad esempio, è inefficiente dover ridefinire costanti universali come π. CLIPS fornisce il costrutto **defglobal** in modo che i valori possano essere universalmente conosciuti da tutte le regole.

Un altro tipo di funzione utile sono i numeri casuali. CLIPS ha una funzione **random** che restituisce un valore intero “casuale”. La funzione dei numeri casuali di CLIPS in realtà restituisce numeri **pseudocasuali**, il che significa che non sono veramente casuali ma sono generati da una formula matematica. Per la maggior parte degli scopi i numeri pseudocasuali andranno bene. Si noti che la funzione random di CLIPS utilizza la funzione rand della libreria ANSI C che potrebbe non essere disponibile su tutti i computer che non aderiscono a questo standard. Per ulteriori informazioni su tutti questi argomenti, consultare il *CLIPS Reference Manual.*

Oltre a controllare i fatti per controllare l'esecuzione dei programmi, CLIPS fornisce un modo di controllo più diretto mediante l'assegnazione esplicita della salienza [salience] alle regole. Il problema principale associato all'uso esplicito della salienza mentre apprendendo CLIPS è la tendenza ad abusare della salienza e scrivere programmi sequenziali. Questo uso eccessivo vanifica l’intero scopo dell’utilizzo di un linguaggio basato su regole, che è quello di fornire un veicolo naturale per quelle applicazioni meglio rappresentate dalle regole. Allo stesso modo, i linguaggi procedurali sono i migliori per applicazioni fortemente orientate al controllo, mentre i linguaggi orientati agli oggetti sono i migliori per rappresentare gli oggetti. CLIPS ha parole chiave chiamate **declare salience** che possono essere utilizzate per impostare esplicitamente la priorità delle regole.

La salienza viene impostata utilizzando un valore numerico compreso tra il valore più piccolo di -10000 e quello più alto di 10000. Se una regola non ha rilevanza assegnata esplicitamente dal programmatore, CLIPS assume una rilevanza pari a zero. Si noti che una salienza pari a zero è a metà strada tra il valore di salienza più grande e quello più piccolo. Una salienza pari a zero non significa che la regola non ha salienza ma, piuttosto, che ha un livello di priorità intermedio.

CLIPS fornisce alcune strutture di programmazione procedurale che possono essere utilizzate sulla destra (RHS). Queste strutture sono **while** e **if then else** che si trovano anche nei moderni linguaggi di alto livello come Ada, C e Java.

Un'altra funzione utile con i cicli (while) è **break** che termina il ciclo (while) attualmente in esecuzione. La funzione **return** termina immediatamente la funzione deffunction, la funzione generica, il metodo o il message-handler attualmente in esecuzione.

Qualsiasi funzione può essere richiamata dalla parte destra (RHS), il che contribuisce notevolmente alla potenza di CLIPS. Sono disponibili molte altre funzioni CLIPS che possono restituire numeri, simboli o stringhe. Queste funzioni possono essere utilizzate per i loro valori restituiti o per i loro **effetti collaterali**. Un esempio di funzione utilizzata solo per il suo effetto collaterale è (printout). Il valore restituito da (printout) non ha significato. L'importanza della (printout) risiede nel suo effetto collaterale sull'output. In generale, le funzioni possono avere argomenti nidificati se appropriati per l'effetto desiderato.

Prima che sia possibile accedere a un file in lettura o scrittura, è necessario aprirlo utilizzando la funzione **open**. Il numero di file che possono essere aperti contemporaneamente dipende dal sistema operativo e dall'hardware. Quando non c'è più bisogno di accedere a un file, lo si deve chiudere con la funzione **close**. A meno che un file non venga chiuso, non vi è alcuna garanzia che le informazioni scritte in esso verranno salvate.

Il **nome logico** di un file è il modo in cui CLIPS identifica il file. Il nome logico è un nome globale con il quale CLIPS riconosce questo file in *tutte* le regole. Sebbene il nome logico possa essere identico al nome file, si può utilizzare qualcosa di diverso. Un altro vantaggio di un nome logico è che lo si può facilmente sostituire un nome file diverso senza apportare modifiche sostanziali al programma.

La funzione per leggere i dati da un file è la familiare (read) o (readline). L'unica cosa nuova che da fare è specificare il nome logico da cui leggere come argomento di (read) o (readline).

Per leggere con (read) più di un campo, è necessario utilizzare un ciclo. Anche con (readline), è necessario un ciclo per leggere più righe. Un ciclo può essere scritto facendo in modo che una regola ne attivi un'altra o con un ciclo while. Il ciclo non dovrebbe tentare di leggere oltre la fine del file altrimenti il sistema operativo emetterà un messaggio di errore. Per evitare ciò, CLIPS restituisce un campo simbolico **EOF** se si tenta di leggere oltre la fine del file (EOF).

La funzione di **valuazione**, **eval**, viene utilizzata per valutare qualsiasi stringa o simbolo tranne i costrutti di tipo "def" come defrule, deffacts, ecc., come se immessi nel livello più alto. La funzione **build** si occupa dei costrutti di tipo “def”. La funzione (build) è il complemento di (eval). La funzione build valuta una stringa o un simbolo come se fosse immesso al livello più alto e restituisce TRUE se l'argomento è un costrutto legale di tipo def come (defrule), (deffacts) e così via.

Capitolo 8

Questioni di eredità

> *Il modo più semplice per ottenere ricchezza è ereditarla; il secondo modo migliore è ricavarla grazie al lavoro degli altri; sposare la ricchezza è troppo simile al lavoro.*

Questo capitolo è una panoramica della programmazione orientata agli oggetti in CLIPS. A differenza della programmazione basata su regole in cui si può semplicemente entrare subito e scrivere una regola senza preoccuparsi di cos'altro c'è nel sistema, la programmazione orientata agli oggetti richiede una conoscenza materiale essenziale.

Come essere oggettivi

Una caratteristica chiave di una buona progettazione del programma è la *flessibilità*. Sfortunatamente, la rigida metodologia delle tecniche di programmazione strutturata non fornisce la flessibilità necessaria per modifiche rapide, affidabili ed efficienti. Il **paradigma della programmazione orientata agli oggetti** (**OOP**) , fornisce questa flessibilità.

Il termine *paradigma* deriva dalla parola greca *paradeigma* che significa modello, esempio o pattern. In informatica, un *paradigma* è una metodologia coerente e organizzata per cercare di risolvere un problema. Oggi esistono molti paradigmi di programmazione come OOP, **procedurale**, **basato su regole** e **connessionista**. Il termine **sistemi neurali artificiali** è un sinonimo moderno del vecchio termine *connessionista*.

La programmazione tradizionale è procedurale perché enfatizza algoritmi o procedure nella risoluzione dei problemi. Molti linguaggi sono stati sviluppati per supportare questo paradigma procedurale, come Pascal, C, Ada, FORTRAN e BASIC. Questi linguaggi sono stati adattati anche per la **progettazione orientata agli oggetti** (**OOD**) aggiungendo estensioni o imponendo una metodologia di progettazione ai programmatori. Al contrario, sono stati sviluppati nuovi linguaggi per fornire l'OOP, che *non* è uguale all'OOD. Si può fare OOD in qualsiasi linguaggio, anche in assembly.

CLIPS fornisce tre paradigmi: regole, oggetti e procedure. Si imparerà di più sugli oggetti nel **CLIPS Object-Oriented Language** (**COOL**) che è integrato con le regole e i paradigmi basati sulle procedure di CLIPS. CLIPS supporta il paradigma procedurale attraverso funzioni generiche, deffunzioni e funzioni esterne definite dall'utente. A seconda dell'applicazione, è possibile utilizzare regole, oggetti, procedure o una loro combinazione.

Piuttosto che imporre un unico paradigma all'utente, la nostra filosofia è che una varietà di strumenti specializzati, un approccio **multi-paradigma**, è meglio che cercare di costringere tutti a utilizzare un unico strumento generico. Per analogia, anche se si può usare martello e chiodi per fissare *qualsiasi cosa*, ci sono casi in cui sono preferiti altri elementi di fissaggio. Ad esempio, si immagini di allacciare i pantaloni con un martello e chiodi invece che con una cerniera. (NOTA: se qualcuno usa martello e chiodi sui pantaloni, si prega di contattare il Guinness dei primati.)

Le cose di classe

In OOP una classe è un **template** che descrive le caratteristiche comuni o gli **attributi** degli oggetti. Si noti che questo uso del termine template non è lo stesso di *deftemplate* descritto in un capitolo precedente. Qui, la parola template viene utilizzata nel senso di uno strumento utilizzato per costruire oggetti con attributi comuni. Per analogia, un righello è un template per disegnare linee rette mentre uno stampino per biscotti è un template sinuoso.

Le classi di oggetti sono organizzate in una gerarchia o in un grafo per descrivere le relazioni degli oggetti in un sistema. Ogni classe è un'**astrazione** di un sistema del mondo reale o di qualche altro sistema logico che stiamo cercando di modellare. Ad esempio, un modello astratto di un sistema del mondo reale potrebbe essere un'automobile. Un altro

modello astratto di sistema logico potrebbe essere rappresentato da strumenti finanziari come azioni e obbligazioni, o numeri complessi. Il termine *astrazione* si riferisce a (1) la descrizione astratta di un oggetto del mondo reale o di un altro sistema che stiamo cercando di modellare, o (2) il processo di rappresentazione di un sistema in termini di classi. L'astrazione è una delle cinque caratteristiche generalmente accettate di un vero linguaggio OOP. Le altre sono **ereditarietà**, **incapsulamento**, **polimorfismo** e **associazione dinamica** [dynamic binding]. Questi termini verranno spiegati in dettaglio durante la lettura di questo libro. CLIPS supporta tutte e cinque queste funzionalità.

Il termine *astratto* significa che non ci occupiamo dei dettagli non essenziali. Una descrizione astratta di un sistema complesso è una descrizione semplificata che si concentra su informazioni rilevanti per uno scopo specifico. Pertanto, il sistema è rappresentato da un modello più semplice e di più facile comprensione. Come esempio familiare, quando alcune persone guidano un'auto, utilizzano un modello astratto di guida composto da due elementi: il volante e l'acceleratore. Cioè, a queste persone non interessano le centinaia di componenti che compongono un'automobile, né la teoria dei motori a combustione interna, il codice della strada e così via. Saper usare solo volante e acceleratore è il loro modello astratto di guida.

Una delle cinque caratteristiche fondamentali dell'OOP è l'ereditarietà. Le classi sono organizzate in una gerarchia con le classi più generali in alto e le classi più specializzate in basso. Ciò consente di definire facilmente nuove classi come perfezionamenti specializzati o modifiche di classi esistenti.

L'uso dell'ereditarietà può accelerare notevolmente lo sviluppo del software e aumentare l'affidabilità perché non è necessario creare nuovo software da zero ogni volta che viene creato un nuovo programma. L'OOP semplifica l'utilizzo del **codice riutilizzabile**. I programmatori OOP spesso utilizzano librerie di oggetti costituite da centinaia o migliaia di oggetti. Questi oggetti possono essere utilizzati o modificati a piacere in un nuovo programma. Oltre alle librerie di oggetti di pubblico dominio, numerose aziende commercializzano librerie di oggetti commerciali. Sebbene il concetto di componenti software riutilizzabili esista fin dagli albori delle librerie di subroutine FORTRAN negli anni '60, il concetto non è mai stato utilizzato con così tanto successo per lo sviluppo di software generale.

Per poter definire una classe è necessario specificare una o più **classi genitori** o **superclassi** della classe da definire. Per analogia con le superclassi, ogni persona ha genitori; le persone non nascono spontaneamente (anche se a volte potresti chiederti se alcune persone hanno davvero avuto dei genitori). L'opposto di una superclasse è una **classe figlia** o **sottoclasse**.

Ciò determina l'ereditarietà della nuova classe. Una sottoclasse eredita **attributi** da una o più superclassi. Il termine *attributo* in COOL si riferisce alle **proprietà** di un oggetto, nomi di **slot** che lo descrivono. Ad esempio, un oggetto per rappresentare una persona potrebbe avere slot per nome, età, indirizzo e così via.

Un'**istanza** è un oggetto che ha *valori* per gli slot come John Smith, 28, 1000 Main St., Clear Lake City, TX. Le classi di livello inferiore ereditano automaticamente i propri slot dalle classi di livello superiore, a meno che gli slot non siano esplicitamente bloccati. Vengono definiti nuovi slot in aggiunta a quelli ereditati per impostare tutti gli attributi che descrivono la classe.

Il **comportamento** [behavior] di un oggetto è definito dai suoi **gestori di messaggi** [message-handlers], o **gestori** in breve. Un gestore di messaggi per un oggetto risponde a **messaggi** ed esegue le azioni richieste. Ad esempio, inviando il messaggio

```
(send [John_Smith] print)
```

farebbe sì che il gestore di messaggi appropriato stampi i valori degli slot dell'istanza John_Smith. Le istanze sono generalmente specificate tra **parentesi quadre**, **[ ]**. Un messaggio inizia con la funzione **send**, seguita dal nome dell'istanza, dal nome del

messaggio e da eventuali argomenti richiesti. Ad esempio, nel caso del messaggio print, non ci sono argomenti. Un **oggetto** in CLIPS è un'istanza di una classe.

L'incapsulamento di slot e gestori all'interno di un oggetto è un'altra delle cinque caratteristiche generalmente accettate di un OOP. Il termine *incapsulato* significa che una classe è definita in termini di slot e gestori [handler]. Sebbene un oggetto di una classe possa ereditare slot e gestori dalle sue superclassi, con alcune eccezioni discusse più avanti, *i valori degli slot dell'oggetto non possono essere alterati o esaminati senza inviare un messaggio all'oggetto*.

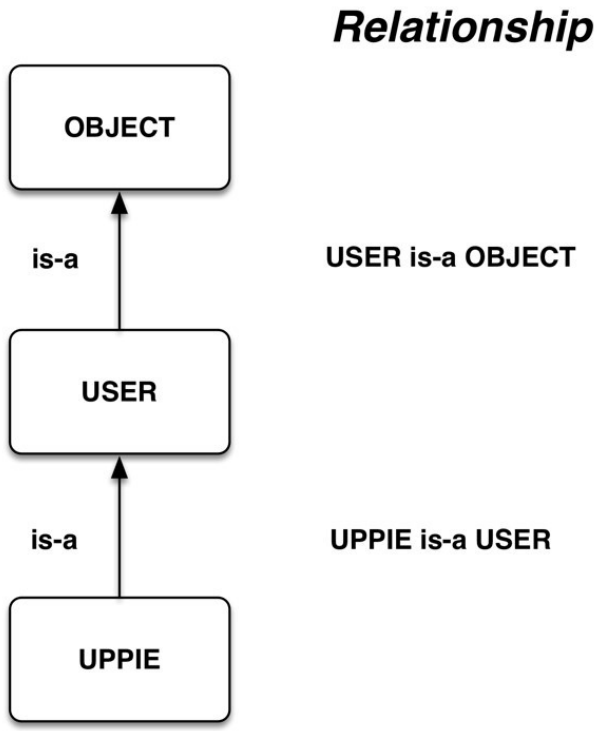
La classe radice o semplicemente root di CLIPS è una classe di sistema predefinita chiamata OBJECT. La classe di sistema predefinita **USER** è una sottoclasse di OBJECT.

Come si ottengono gli UPPIE

Ad esempio, supponiamo di voler definire una classe chiamata UPPIE, che è un termine colloquiale per **affarista, capitalista**. Notare che in questo libro seguiremo la convenzione di scrivere le classi tutte in maiuscolo.

La Figura 8.1 illustra come gli UPPIE ottengono la loro eredità fino alla classe radice OBJECT. Si noti che UPPIE è definito come una sottoclasse di USER. I riquadri o **nodi** rappresentano le classi mentre le frecce di collegamento sono dette **link**. Le linee vengono spesso utilizzate al posto delle frecce per semplicità nel disegno. Inoltre, poiché CLIPS supporta solo link **is-a**, da ora in poi la relazione “is-a” non verrà scritta esplicitamente accanto a ciascun collegamento.

Fig. 8.1 La classe UPPIE



La convenzione che seguiremo per la **relazione** tra le classi è che la coda della freccia si trova sulla sottoclasse mentre la testa punta sulla superclasse. Le relazioni in Fig. 8.1 seguono questa convenzione. Un'altra possibile convenzione è quella di utilizzare le frecce per indicare le sottoclassi.

Il link "is-a" indica l'ereditarietà degli slot da una classe alla sua sottoclasse. Una classe può avere zero o più sottoclassi. Tutte le classi tranne OBJECT devono avere una superclasse. Poiché UPPIE eredita anche tutti gli slot di USER e USER eredita tutti gli slot di OBJECT, ne consegue che UPPIE eredita anche tutti gli slot di OBJECT. Lo stesso principio di ereditarietà si applica anche ai gestori di messaggi di ciascuna classe. Ad esempio, UPPIE eredita tutti i gestori [handler] di USER e OBJECT.

L'ereditarietà di slot e gestori è particolarmente importante in OOP poiché significa che non è necessario ridefinire le proprietà e il comportamento di ogni nuova classe di oggetti definita. Invece, ogni nuova classe eredita tutte le proprietà e il comportamento dalle classi di livello più alto. Poiché il nuovo comportamento viene ereditato, potrebbe ridurre sostanzialmente la **verifica e validazione** (**V&V**) dei gestori. V&V significa essenzialmente che il prodotto è stato costruito correttamente e che soddisfa i requisiti. Il compito di verificare e validare il software può richiedere più tempo e denaro rispetto allo sviluppo del software stesso, soprattutto se il software influisce sulla vita umana e sulla proprietà. L'ereditarietà dei gestori consente un riutilizzo efficiente del codice esistente e accelera lo sviluppo.

Le classi sono definite in CLIPS utilizzando il costrutto **defclass**. La classe UPPIE è definita in un'istruzione come segue.

```
(defclass UPPIE (is-a USER))
```

Notare la somiglianza tra la relazione UPPIE-USER nella Figura 8.1 e il costrutto (defclass).

Non è necessario inserire le classi USER o OBJECT poiché queste sono classi predefinite e quindi CLIPS conosce già la loro relazione. Infatti, se si tenta di definire USER o OBJECT, verrà visualizzato un messaggio di errore poiché non è possibile modificare le classi predefinite, a meno che non si modifichi il codice sorgente di CLIPS.

Poiché CLIPS è **case-sensitive**, i comandi e le funzioni *devono* essere inseriti in minuscolo. Le classi di sistema predefinite come USER e OBJECT *devono* essere immesse in maiuscolo. Sebbene sia possibile inserire classi definite dall'utente in minuscolo o maiuscolo, seguiremo la convenzione di utilizzare le maiuscole per le classi per una questione di leggibilità.

Il formato di base del comando defclass per definire solo le classi e non gli slot è:

```
(defclass <class>
(is-a <direct-superclasses>))
```

L'elenco delle classi, <direct-superclasses>, è detto **elenco di precedenza delle superclassi dirette** perché definisce come una classe è collegata alle sue superclassi dirette. Le superclassi dirette di una classe sono una o più classi che prendono il nome dalla parola chiave is-a. Nel nostro esempio, la classe DUCKLING è la superclasse diretta di DUCK. Notare che almeno *una* superclasse diretta deve essere specificata nell'elenco di precedenza delle superclassi dirette.

Se l'elenco diretto delle superclassi fosse il seguente,

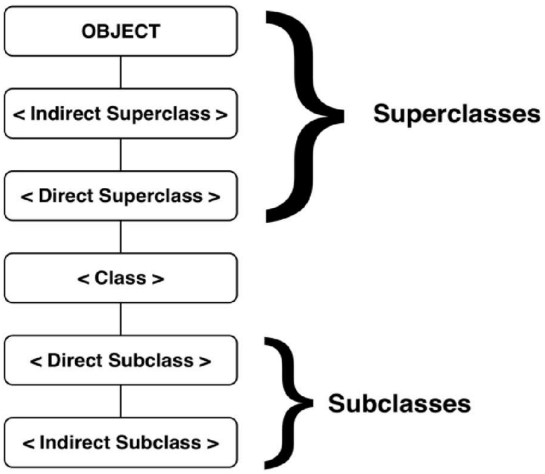
```
(defclass DUCK
(is-a DUCKLING USER OBJECT))
```

allora USER e OBJECT sarebbero anche superclassi dirette di DUCK. In questo esempio non fa alcuna differenza se oltre a DUCKLING vengono specificati anche USER e OBJECT. Infatti, poiché USER e OBJECT sono classi predefinite che sono sempre collegate in modo tale che USER is-a OBJECT e OBJECT sia la root, non è mai necessario specificarle tranne quando si definisce una sottoclasse di USER. Poiché USER eredita solo da OBJECT, non è necessario specificare OBJECT se è specificato USER.

Le **superclassi indirette** di una classe sono tutte le classi *non* chiamate dopo "is-a" che contribuiscono con slot e gestori di messaggi per ereditarietà. Nel nostro esempio, le superclassi indirette sono USER e OBJECT. Una classe eredita slot e gestori di messaggi da *tutte* le sue superclassi dirette e indirette. Pertanto, DUCK eredita da DUCKLING, USER e OBJECT.

Una **sottoclasse diretta** è connessa tramite un *singolo* link alla classe superiore. Una **sottoclasse indiretta** ha *più di un link*. La Fig. 8.2 riassume la terminologia della classe.

Fig. 8.2 Relazioni di classi



La classe radice OBJECT è l'*unica* classe che non ha una superclasse.

L'uso di questa nuova e fantasiosa terminologia ci permette di affermare il *Principio di ereditarietà OOP*: una classe "può" ereditare da tutte le sue superclassi.

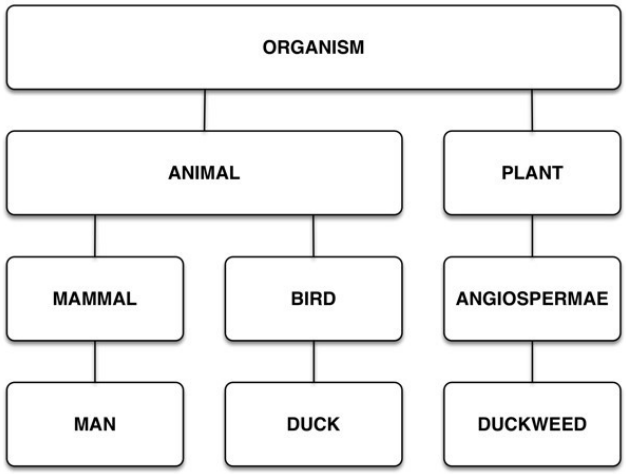
Questo è un concetto semplice ma potente pienamente sfruttato in OOP. Questo principio significa che gli slot e i gestori di messaggi possono essere ereditati per risparmiarci la fatica di ridefinirli per nuove sottoclassi. Inoltre, gli slot possono essere facilmente personalizzati per nuove sottoclassi come modifiche e come composizioni di slot della superclasse. Consentendo un riutilizzo semplice e flessibile del codice esistente, i tempi e i costi di sviluppo del programma vengono ridotti. Inoltre, il riutilizzo del codice funzionante ed esistente riduce al minimo la quantità di verifica e convalida necessarie. Tutti questi vantaggi facilitano le attività di manutenzione del programma di debug, modifiche e miglioramenti una volta rilasciato il codice.

Il motivo per cui si utilizza *può* nel principio è quello di enfatizzare che l'ereditarietà degli slot da una classe può essere bloccata includendo un aspetto no-inherit nella definizione dello slot della classe.

Le classi dirette e indirette di una classe sono tutte quelle che si trovano su un **percorso di ereditarietà** verso OBJECT. Un *percorso di ereditarietà* è un insieme di nodi connessi tra la classe e OBJECT. Nel nostro esempio, il singolo percorso di ereditarietà di DUCK è DUCK, DUCKLING, USER e OBJECT. Vedremo esempi più avanti, come la Figura 8.5, in cui una classe ha **più percorsi di ereditarietà** verso OBJECT.

La Fig. 8.3 illustra una **tassonomia** molto semplificata degli organismi illustrando l'ereditarietà in Natura. Il termine *tassonomia* indica una classificazione. Le tassonomie biologiche sono progettate per mostrare la parentela degli organismi. Cioè, una tassonomia biologica enfatizza le somiglianze tra gli organismi raggruppandoli.

Fig. 8.3 Tassonomia semplificata degli organismi viventi con link is-a



In una tassonomia come quella della Figura 8.3, le linee di connessione sono *tutte* link is-a. Ad esempio, un'ANATRA [DUCK] è un UCCELLO [BIRD]. Un BIRD is-a ANIMAL. Un ANIMAL is-a ORGANISM e così via. Sebbene il patrimonio genetico di ogni *individuo* sia diverso, le caratteristiche di MAN e di DUCK sono le stesse per ciascuna specie.

Nella Fig. 8.3 si nota che la classe più generale, ORGANISM, è in alto, mentre le classi più specializzate sono più in basso nella tassonomia. Nella terminologia CLIPS, diremmo che ciascuna sottoclasse eredita gli *slot* delle sue classi madri. Ad esempio, poiché i mammiferi sono a sangue caldo e danno alla luce piccoli vivi, ad eccezione dell'ornitorinco, la classe MAN eredita gli attributi della classe genitrice MAMMAL (mammifero). La superclasse diretta di MAMMAL è ANIMAL e la sottoclasse diretta diMAMMAL è MAN. La superclasse indiretta dei MAMMAL è ORGANISM.

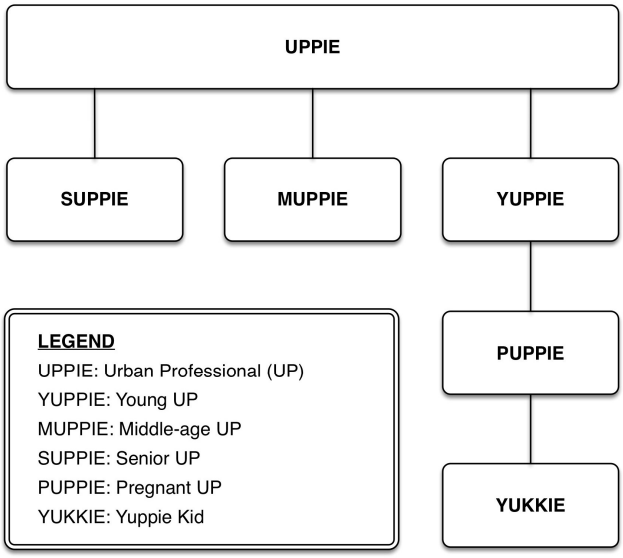
Le altre classi come BIRD, DUCK e così via non hanno alcuna relazione con MAMMAL perché non si trovano su un **percorso di ereditarietà** dalla classe più generale ORGANISM. Un percorso di ereditarietà è qualsiasi percorso da una classe a un'altra che non implica il **backtracking** o il ritracciamento del percorso. Una classe come PLANT non è su un percorso di ereditarietà verso MAMMAL perché dovremmo tornare indietro fino a ORGANISM prima di continuare fino a MAMMAL. Pertanto, MAMMAL non ottiene automaticamente alcuno slot da PLANT o da qualsiasi altra classe non sul percorso di ereditarietà verso MAMMAL. Questo modello di ereditarietà rispecchia il mondo reale, poiché altrimenti potremmo avere l'erba che cresce sulle nostre teste invece dei capelli.

Lo YUKKIE illegittimo

Ora che si ha l'idea di base delle classi, aggiungiamo alcune classi aggiuntive al diagramma UPPIE della Fig. 8.1 per rendere l'esempio più realistico. Questo tipo di sviluppo mediante l'aggiunta di classi di livello inferiore è il modo in cui viene eseguita l'OOP, aggiungendo classi da quelle più generali a quelle più specifiche.

La Fig. 8.4 mostra il diagramma di ereditarietà dello YUKKIE illegittimo**.** Per semplicità, le classi OBJECT e USER non sono mostrate. La gerarchia della Figura 8.4 è un **albero** perché ogni nodo ha esattamente un genitore.

Fig. 8.4 Lo YUKKIE illegittimo



Un esempio familiare di struttura organizzativa ad albero è quello spesso utilizzato dalle aziende che hanno una **gerarchia** composta da presidente, vicepresidenti, capi dipartimento, manager e così via fino al dipendente più in basso. In questo caso, la struttura gerarchica rispecchia l'autorità delle persone all'interno dell'organizzazione. Gli alberi sono generalmente utilizzati per le organizzazioni di persone perché ogni persona ha esattamente un capo, tranne il capo che non ha un capo. I nodi nell'organigramma rappresentano le posizioni come presidente, vicepresidente, ecc. Le linee che collegano le

posizioni sono i **rami** che indicano la divisione delle responsabilità. I collegamenti sono spesso chiamati rami di un albero.

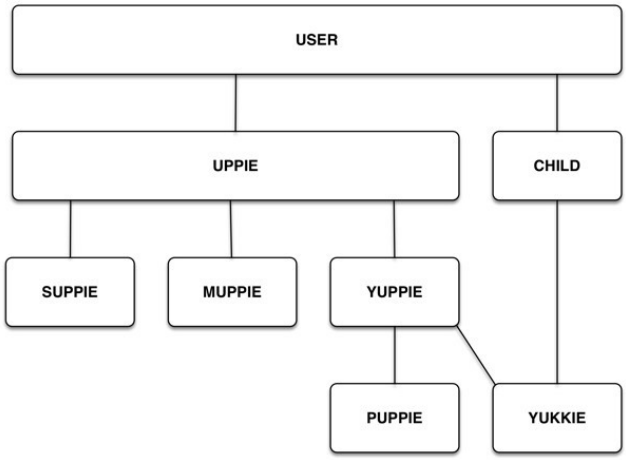
Nella Figura 8.4, ogni classe tranne YUKKIE è legale o **legittima**. Ad esempio, un SUPPIE is-a UPPIE. Un MUPPIE is-a UPPIE. Un YUPPIE is-a UPPIE. Un PUPPIE is-a YUPPIE (non sono ammesse mamme YUPPIE). Vorremmo anche dire che uno YUKKIE is-a YUPPIE e uno YUKKIE is-a UPPIE per eredità. Tuttavia, noi *non* vogliamo dire che uno YUKKIE è un PUPPIE, che è ciò che significa è un link tra YUKKIE e un PUPPIE.

Il link "is-a" tra YUKKIE e PUPPIE è un errore naturale da fare per una persona poiché uno YUKKIE è il figlio di un PUPPIE (in realtà un ex-PUPPIE dopo il parto). Anche se creare un link "is-a" tra YUKKIE e PUPPIE consente allo YUKKIE di ereditare da YUPPIE e UPPIE come desiderato, produce anche una relazione illegittima dicendo che uno YUKKIE is-a PUPPIE. Ciò significa che uno YUKKIE erediterà tutti gli slot di un PUPPIE. Supponendo che uno degli slot PUPPIE specifichi di quanti mesi è incinta il PUPPIE, ciò significa che ogni bambino Yuppie avrà uno slot per indicare di quanti mesi è incinto anch'esso!

È possibile correggere la figura. Però, dobbiamo utilizzare un grafo anziché un albero. A differenza degli alberi, in cui ogni nodo tranne la radice ha esattamente un genitore, ogni nodo in un grafo può avere zero o più nodi collegati a sé. Un esempio familiare di grafo è una mappa stradale in cui le città sono i nodi e le strade sono i collegamenti che le collegano. Un'altra differenza tra alberi e grafici è che la maggior parte dei tipi di alberi ha una struttura gerarchica, mentre i tipi generali di grafici no.

La Fig. 8.5 mostra la classe legittima Yuppie YUKKIE. Una nuova classe CHILD è stata creata ed è un link "is-a" tra YUKKIE e le sue due superclassi, YUPPIE e CHILD. Notare che non esiste più un link illegittimo tra YUKKIE e PUPPIE.

Fig. 8.5 Il legittimo YUKKIE



Questo è un grafo perché la classe YUKKIE ha due superclassi dirette invece di una sola come in un albero. Questo è anche un grafico gerarchico perché le classi sono organizzate utilizzando i link "is-a" dal più generale, USER, al più specifico, SUPPIE, MUPPIE, PUPPIE e YUKKIE. Utilizzando la Figura 8.5, possiamo dire che uno YUKKIE è uno YUPPIE e anche che uno YUKKIE è un CHILD.

Di seguito sono riportati i comandi per aggiungere le sottoclassi mostrate in Fig. 8.5.

```
CLIPS> (clear)
CLIPS> (defclass UPPIE (is-a USER))
CLIPS> (defclass CHILD (is-a USER))
CLIPS> (defclass SUPPIE (is-a UPPIE))
CLIPS> (defclass MUPPIE (is-a UPPIE))
CLIPS> (defclass YUPPIE (is-a UPPIE))
CLIPS> (defclass PUPPIE (is-a YUPPIE))
CLIPS> (defclass YUKKIE (is-a YUPPIE CHILD))
```

L'ordine in cui le classi vengono definite deve essere tale che una classe venga definita prima delle sue sottoclassi. Così,

```
(defclass CHILD
(is-a USER))
```

must be entered *before*

```
(defclass YUKKIE
(is-a YUPPIE CHILD))
```

CLIPS emetterà un messaggio di errore se si prova ad inserire la classe YUKKIE prima della classe CHILD.

Notate l'ordine da sinistra a destra con cui SUPPIE, MUPPIE e YUPPIE sono disegnati nella Figura 8.5. Ciò corrisponde all'ordine in cui queste classi vengono inserite in CLIPS ed è la convenzione che seguiremo. Si può anche vedere perché CHILD è disegnato a destra di UPPIE in quanto è stato inserito *dopo* la classe UPPIE.

Nella Fig. 8.5, notare che il link YUKKIE—YUPPIE è disegnato a sinistra del link YUKKIE—CHILD. Un'altra convenzione che seguiremo è quella di scrivere le superclassi dirette da sinistra a destra nella lista di precedenza secondo il loro ordine da sinistra a destra disegnato in un grafo. L'ordinamento di YUPPIE CHILD nella lista di precedenza di YUKKIE viene fatto per soddisfare questa convenzione.

Fammelo vedere

CLIPS fornisce una serie di funzioni per mostrare informazioni sulle classi, come funzioni di predicato per verificare se una classe è una superclasse o una sottoclasse di un'altra.

La funzione **superclassp** restituisce TRUE se <class1> è una superclasse di <class2> e FALSE altrimenti. La funzione **subclassp** restituisce TRUE se <class1> è una sottoclasse di <class2> e FALSE altrimenti. La forma generale di entrambe le funzioni è

```
(function <class1> <class2>)
```

Per esempio,

```
CLIPS> (superclassp UPPIE YUPPIE) TRUE
CLIPS> (superclassp YUPPIE UPPIE)
FALSE
CLIPS> (subclassp YUPPIE UPPIE) TRUE
CLIPS> (subclassp UPPIE YUPPIE)
FALSE
CLIPS>
```

Ora controlliamo se CLIPS ha accettato tutte queste nuove classi. Un modo per farlo è utilizzare il comando **list-defclasses**. Di seguito è riportato l'output del comando.

```
CLIPS> (list-defclasses)
FLOAT
INTEGER
SYMBOL
STRING
MULTIFIELD
EXTERNAL-ADDRESS
FACT-ADDRESS
INSTANCE-ADDRESS
INSTANCE-NAME
OBJECT
PRIMITIVE
NUMBER
LEXEME
ADDRESS
INSTANCE
USER
UPPIE
CHILD
SUPPIE
MUPPIE
YUPPIE
PUPPIE
YUKKIE
For a total of 23 defclasses.
CLIPS>
```

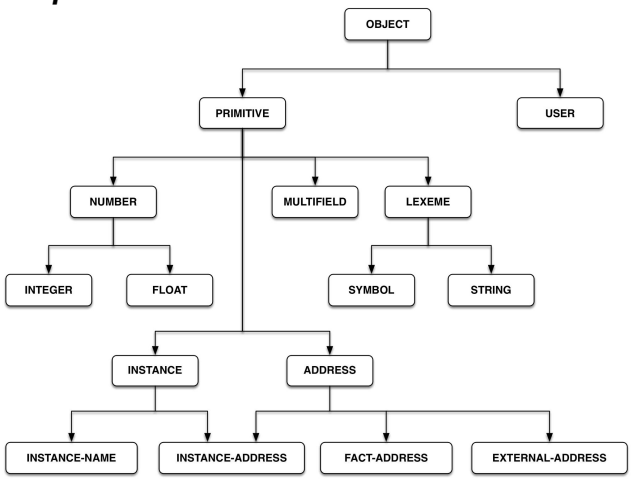
Si noti che il comando (list-defclasses) non indica la struttura gerarchica delle classi. Cioè, list-defclasses non indica quali classi sono sottoclassi o superclassi di altre.

Guardando l'elenco, si vedranno tutte le classi user-defined inserite: UPPIE, CHILD, YUPPIE, MUPPIE, SUPPIE, PUPPIE e YUKKIE. Oltre alle classi di sistema predefinite OBJECT e USER discusse finora, esistono numerose altre classi predefinite. Si dovrebbe riconoscere che la maggior parte di esse ha lo stesso nome dei familiari tipi di CLIPS di cui

si è discusso nei capitoli precedenti. I tipi predefiniti di CLIPS sono definiti anche come classi in modo che possano essere utilizzati con COOL.

Il diagramma generale dell'ereditarietà delle classi predefinite dal *CLIPS Reference Manual*, è mostrato in Fig. 8.6, dove le frecce puntano alle sottoclassi.

Fig. 8.6 Le classi CLIPS predefinite



La classe OBJECT è la radice dell'albero ed è collegata tramite **rami** alle sue sottoclassi. I termini *ramo* [branch], **edge**, *link* e **arco** sono sostanzialmente sinonimi in quanto indicano tutti una connessione tra i nodi. Ogni sottoclasse è al di sotto delle sue superclassi.

Ciascuna sottoclasse ha una **specificità** maggiore rispetto alla sua superclasse. Il termine *specificità* significa che una classe è più restrittiva. Ad esempio, LEXEME è una superclasse di SYMBOL e STRING. Se vi dicessero che un oggetto appartiene alla classe LEXEME, si saprebbe che potrebbe essere solo un SYMBOL o una STRING. Tuttavia, se un oggetto è un SYMBOL, non può essere una STRING e *viceversa*. Pertanto, le classi SYMBOL e STRING sono più *specifiche* di LEXEME.

Il comando **browse-classes** mostra la gerarchia delle classi tramite indentazione.

```
CLIPS> (browse-classes)
OBJECT
PRIMITIVE
NUMBER
INTEGER
FLOAT
LEXEME
SYMBOL
STRING
MULTIFIELD
ADDRESS
EXTERNAL-ADDRESS
FACT-ADDRESS
INSTANCE-ADDRESS *
INSTANCE
INSTANCE-ADDRESS *
INSTANCE-NAME
USER
UPPIE
SUPPIE
MUPPIE
YUPPIE
PUPPIE
YUKKIE *
CHILD
YUKKIE *
CLIPS>
```

L'asterisco dopo il nome di una classe indica che ha più superclassi.

Il comando (browse-classes) ha un argomento opzionale che specifica la classe iniziale per le sottoclassi da mostrare. Questo è utile se non si è interessati a elencare tutte le classi. Gli esempi seguenti illustrano come mostrare porzioni del grafo YUPPIE di Fig. 8.5, chiamate **sottoalberi** o **sottografi** a seconda che i nodi e i link formino un albero o un grafo.

```
CLIPS> (browse-classes UPPIE)
```

```
UPPIE
SUPPIE
MUPPIE
YUPPIE
PUPPIE
YUKKIE *
CLIPS> (browse-classes YUPPIE)
YUPPIE
PUPPIE
YUKKIE *
CLIPS> (browse-classes YUKKIE)
YUKKIE *
CLIPS>
```

Una **classe astratta** è progettata solo per l'ereditarietà. Per la classe astratta USER non possono essere definite **istanze dirette**, che sono istanze definite direttamente per una classe. Oltre alle informazioni sulla classe, viene descritta la **precedenza dell'ereditarietà** delle classi. Questa è un **lista ordinata** in cui l'ordine da sinistra a destra indica la precedenza dalla più alta alla più bassa con cui le classi contribuiscono per ereditarietà. La precedenza dell'ereditarietà elenca tutte le superclassi di una classe fino alla classe radice OBJECT. Si può anche vedere che le informazioni dirette sulle superclassi indicano la superclasse che si trova un link sopra una classe mentre l'elenco di precedenza dell'ereditarietà mostra *tutte* le superclassi.

Anche se una classe non ha istanze dirette, avrà **istanze indirette** se ha sottoclassi che hanno istanze. Le istanze indirette di una classe sono tutte le istanze delle sue sottoclassi.

Una **classe concreta** può avere istanze dirette. Ad esempio, data una classe concreta COW, l'istanza diretta Elsie sarebbe la famosa venditrice televisiva. Normalmente anche le classi che ereditano da classi astratte sono classi astratte. Tuttavia, le classi che ereditano dalle classi di sistema, come USER, sono considerate concrete se non diversamente specificato. Quindi UPPIE e YUPPIE sono anche classi concrete.

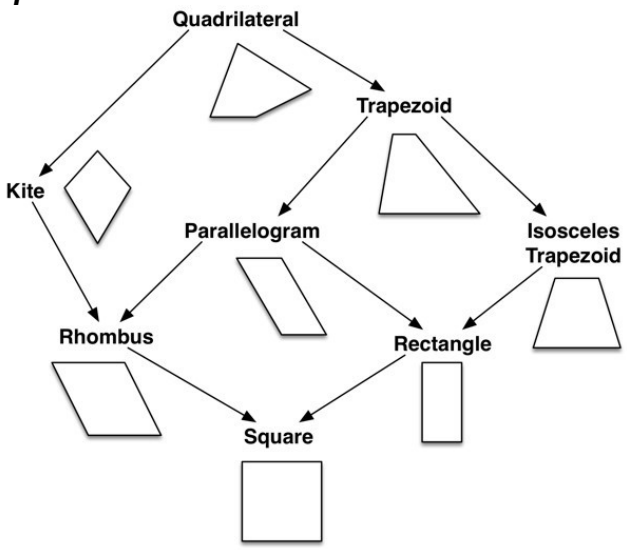
Si consiglia vivamente che tutte le classi definite in CLIPS siano sottoclassi di USER. CLIPS fornirà automaticamente i gestori per print, init ed delete se le classi sono sottoclassi di USER.

La classe YUKKIE ha **ereditarietà multipla** poiché ha due superclassi dirette, CHILD e YUPPIE. Se si ripensa all'analogia tra l'eredità e i capi di un'organizzazione, la questione dell'eredità multipla solleva una domanda interessante: chi è il capo? Nel caso di una struttura ad albero, ogni classe ha una sola superclasse diretta (boss) e quindi è facile capire da chi prendere ordini. Tuttavia, nel caso dell'ereditarietà multipla, sembrano esserci più "capi" di uguale autorità che sono le superclassi dirette.

Nel caso di classi disposte in un albero, cioè di ereditarietà singola, l'ereditarietà è semplicemente costituita da tutte le classi lungo il percorso di ereditarietà che riporta a OBJECT. Il percorso di ereditarietà in un albero è solo il percorso più breve da una classe a OBJECT. Questo concetto di ereditarietà si applica anche ad un sottografo di un grafo che è un albero. Ad esempio, la precedenza di ereditarietà dei sottografi UPPIE, SUPPIE, MUPPIE, YUPPIE e PUPPIE è un albero poiché ciascuno di essi ha un solo genitore. Pertanto, la precedenza di ereditarietà di ciascuno di essi è la sequenza più breve di collegamenti a OBJECT. Ad esempio, la precedenza di ereditarietà di PUPPIE è PUPPIE, YUPPIE, UPPIE, USER e OBJECT.

La Fig. 8.7 è un altro esempio di grafico. Si noti che alcuni nodi come il rombo [rhombus] hanno più di un genitore.

Fig. 8.7 Il grafico del quadrilatero



Per semplicità, in questo libro discuteremo solo dell'ereditarietà singola. Per ulteriori dettagli sull'ereditarietà multipla, consultare il *CLIPS Reference Manual*.

Altre funzionalità

Di seguito vengono mostrate alcune altre funzioni utili con le classi.

ppdefclass: Stampa in modo formattato la struttura interna della declass.
undefclass: Elimina la classe.
describe-class: Informazioni aggiuntive sulle classi.
class-abstractp: Lla funzione del predicato restituisce TRUE se la classe è astratta.

Per ulteriori informazioni su queste funzioni e sugli argomenti menzionati in questo capitolo, consultare il *CLIPS Reference Manual*.

Capitolo 9

Messaggi significativi

È sempre meglio compiacere i superiori piuttosto che i subordinati.
— Bowen

In questo capitolo si approfondiranno le classi e gli oggetti chiamati istanze. Si vedrà come specificare gli attributi delle classi utilizzando gli slot e come inviare messaggi agli oggetti.

Gli uccelli e le api

Nel capitolo 1 abbiamo visto i concetti base dell'ereditarietà. La ragione per cui l'ereditarietà è così importante nell'OOP è che l'ereditarietà consente la facile costruzione di **software personalizzato**. Con *software personalizzato* non intendiamo che il software sia creato da zero. Piuttosto, è più come prendere un articolo prodotto in serie e modificarlo per un'applicazione speciale. L'articolo prodotto in serie può essere considerato il prodotto di una **software factory** che produce in modo rapido, economico e affidabile articoli di tipo generale destinati a essere facilmente personalizzati.

Il cuore del paradigma OOP è la creazione di una gerarchia di classi per produrre software in modo rapido, semplice e affidabile. Generalmente, questo software è una modifica del software esistente in modo che i programmatori non sempre "reinventino la ruota".

Sebbene in passato si sia cercato di fornire codice riutilizzabile attraverso meccanismi come le librerie di subroutine, il paradigma di **OOP puro** in linguaggi come Smalltalk porta il concetto di codice riutilizzabile alla sua logica conclusione cercando di costruire *tutto* il software in un sistema come codice riutilizzabile. In Smalltalk tutto è un oggetto, anche le classi. In CLIPS, le istanze dei tipi primitivi come NUMBER, SYMBOL, STRING e così via, nonché le istanze definite dall'utente sono oggetti. Le classi *non* sono oggetti in CLIPS. Ad esempio, il NUMBER 1, il SYMBOL Duck, la STRING "Duck" e l'istanza user-defined Duck sono tutti oggetti.

Il paradigma OOP è molto diverso dall'approccio della libreria di subroutine in cui frammenti di codice di subroutine possono o meno essere utilizzati a seconda dei capricci del programmatore. Il paradigma OOP incoraggia e supporta il codice modulare, i gestori dei messaggi, che può essere facilmente modificato e mantenuto. Questa caratteristica della manutenibilità del codice sta giocando un ruolo sempre più importante con l'aumento delle dimensioni e dei costi dei sistemi.

Una classe in OOP è come una fabbrica di software che contiene le informazioni di progettazione di un oggetto. In altre parole, una classe è come un template che può essere utilizzato per produrre oggetti identici che sono le istanze della classe. L'analogia classica è che una classe è come il progetto di una mucca, e l'oggetto che produce latte, come Elsie, ne è l'istanza.

La sintassi generale del nome di un'istanza è semplicemente un simbolo circondato da parentesi quadre, [ ], come segue.

```
[<name>]
```

Le parentesi in realtà *non* fanno parte del nome dell'istanza, che è un simbolo, come Elsie. Le parentesi vengono utilizzate per racchiudere il nome di un'istanza se esiste il pericolo di ambiguità nell'utilizzo del nome. Ciò può verificarsi nella funzione (send) e quindi le parentesi vengono utilizzate in (send). In caso di dubbio usate le parentesi perché non fa male.

Di seguito sono mostrati alcuni esempi di diversi tipi di oggetti in CLIPS:

SYMBOL	Dorky_Duck
STRING	"Dorky_Duck"
FLOAT	1.0

INTEGER 1
MULTIFIELD (1 1.0 Dorky_Duck "Dorky_Duck")
DUCK [Dorky_Duck]

Le classi **SYMBOL**, **STRING**, **FLOAT**, **INTEGER** e **MULTIFIELD** hanno gli stessi nomi di quelli con cui si ha familiarità dalla programmazione basata su regole in CLIPS. Questi sono chiamati **tipi di oggetto primitivi** perché sono forniti da CLIPS e mantenuti automaticamente secondo necessità. Questi tipi primitivi vengono forniti principalmente per l'uso in **funzioni generiche**. Due **classi composte** sono **NUMBER** che è FLOAT o INTEGER e **LEXEME** che è SYMBOL o STRING. Le classi composte vengono fornite per comodità se il tipo di numero o il tipo di caratteri non ha importanza.

Al contrario, i **tipi di oggetti user-defined** sono quelli definiti tramite classi definite dall'utente. Se si fa riferimento alle classi CLIPS predefinite della Figura 8.6 nel capitolo 8, si vedrà che le classi primitive e user-defined sono la divisione di livello superiore delle classi in CLIPS.

Due funzioni convertono un simbolo in un nome di istanza e *viceversa*. Il **symbol-toinstance-name** converte un simbolo in un nome di istanza, come mostrato di seguito.

```
; Get rid of any old classes
CLIPS> (clear)
CLIPS>
(symbol-to-instance-name Dorky_Duck)
[Dorky_Duck]
CLIPS>
(symbol-to-instance-name
(sym-cat Dorky "_" Duck))
[Dorky_Duck]
CLIPS>
```

Si noti come le funzioni CLIPS standard come (sym-cat), che concatena gli elementi, possono essere utilizzate con il sistema di oggetti di CLIPS.

La funzione opposta, **instance-name-to-symbol**, converte un nome di istanza in un simbolo, come mostrano gli esempi seguenti.

```
CLIPS>
(instance-name-to-symbol [Dorky_Duck])
Dorky_Duck
CLIPS>
(str-cat
(instance-name-to-symbol [Dorky_Duck])
" is a DUCK") "Dorky_Duck is a DUCK"
CLIPS>
```

Dorky Duck

C'è una differenza tra Natura e OOP. In Natura, gli oggetti vengono riprodotti solo da oggetti simili, come gli uccelli e le api (la gallina e l'uovo sono le eccezioni a questa regola). Tuttavia, in OOP le istanze vengono create solo utilizzando il template della classe. In un OOP puro come Smalltalk, viene creata un'istanza di una classe specifica inviando un messaggio alla classe. Infatti, il cuore dell'OOP consiste nell'invio di diversi tipi di messaggi da un oggetto all'altro, anche da un oggetto a se stesso.

Per vedere come funzionano i messaggi, iniziamo inserendo i seguenti comandi per creare una classe DUCK definita dall'utente e verificare che sia inserita. Si noti che non c'è alcun descrittore **role** per la classe DUCK. Se una classe ha un **ruolo concreto**, è possibile creare istanze dirette della classe. Se il ruolo non è specificato, CLIPS determina il ruolo per ereditarietà. Per determinare il ruolo tramite ereditarietà, le classi di sistema si comportano come classi concrete. Pertanto, per default, qualsiasi classe che eredita da USER è una classe concreta e non necessita di essere dichiarata come tale per consentire la creazione di istanze dirette.

Se una classe ha un **role abstract** non è possibile crearne un'istanza diretta. Le classi astratte sono definite solo a scopo di ereditarietà. Ad esempio, si potrebbe definire una classe astratta chiamata PERSON le cui proprietà come nome, indirizzo, età, altezza, peso e così via vengono ereditate dalle classi concrete MAN e WOMAN. Un'istanza diretta di MAN potrebbe essere una persona-uomo chiamata Harold, e un'istanza diretta di WOMAN è una persona-donna chiamata Henrietta.

```
CLIPS> (defclass DUCK (is-a USER))
```

```
CLIPS> (describe-class DUCK)
============================================================
==
************************************************************
**
Concrete: direct instances of this class can be created.
Reactive: direct instances of this class can match defrule patterns.

Direct Superclasses: USER
Inheritance Precedence: DUCK USER OBJECT
Direct Subclasses:
------------------------------------------------------------
--
Recognized message-handlers:
init primary in class USER
delete primary in class USER
create primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
direct-duplicate primary in class USER
message-duplicate primary in class USER
************************************************************
**
============================================================
==
CLIPS>
```

Poiché le classi non sono oggetti in CLIPS, non possiamo inviare un messaggio per creare un oggetto. Invece, la funzione **make-instance** viene utilizzata per creare un oggetto istanza. La sintassi di base è la seguente.

```
(make-instance [<instance-name>] of <class>
<slot-override>)
```

Normalmente, si specifica un nome-istanza. Tuttavia, in caso contrario, CLIPS ne genererà uno utilizzando la funzione gensym*. È inoltre possibile specificare i valori degli slot.

Ora che abbiamo una duck factory, creiamo alcune istanze come segue, dove il nome dell'istanza è tra parentesi. Da notare l'uso della parola chiave "**of**" per separare il nome dell'istanza dal nome della classe. È necessario includere "of" altrimenti si verificherà un errore di sintassi. Inoltre, da notare che le parentesi nel codice indicano il nome di un'istanza, mentre le parentesi nella meta-sintassi come per (make-instance) sopra, significa *opzionale*.

```
CLIPS> (make-instance [Dorky] of DUCK)
[Dorky]
CLIPS> (make-instance [Elsie] of COW)
[PRNTUTIL1] Unable to find class COW.
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS>
```

Dopo che l'istanza è stata creata con successo, CLIPS risponde con il nome dell'istanza. Se *non* è possibile creare un'istanza, CLIPS risponde con un FALSE. Inoltre, come i comandi (rules) e (facts), CLIPS ha una funzione **instances** per stampare le istanze di una classe.

Nel caso di (make-instance), le parentesi attorno al nome dell'istanza sono facoltative per una classe USER-defined. Per esempio, creiamo il cugino di Dorky, Dorky_Duck, senza parentesi come segue.

```
CLIPS> (make-instance Dorky_Duck of DUCK)
; Instance Dorky_Duck is made.
[Dorky_Duck]
CLIPS>
```

Ci sono due regole importanti sulle istanze da tenere a mente.

- In un modulo è possibile utilizzare solo un'istanza con lo stesso nome.
- Una classe non può essere ridefinita se esistono istanze della classe.

Ad esempio, creiamo un clone di Dorky_Duck come segue. (È questo clone malvagio che mette sempre Dorky_Duck nei guai perché nessuno riesce a distinguerli. Anche i bambini hanno spesso cloni come questo, e anche alcuni adulti).

```
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
; Still only one Dorky_Duck
For a total of 2 instances.
CLIPS>
```

Molto rumore sulle istanze

Se viene emesso un comando (reset), tutte le istanze in memoria vengono cancellate.

```
CLIPS> (reset)
CLIPS> (instances)
CLIPS>
```

Proprio come (deffacts) definisce i fatti, esiste anche un **definstances** per definire le istanze quando viene eseguito un (reset). Di seguito (definstances) viene illustrato anche il commento facoltativo tra virgolette doppie dopo il nome dell'istanza, DORKY_OBJECTS.

```
CLIPS>
(definstances DORKY_OBJECTS "The Dorky Cousins"
(Dorky of DUCK)
(Dorky_Duck of DUCK))
CLIPS> (instances)
CLIPS> (reset)
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
CLIPS>
```

L'anatra che scompare

Anche se un (reset) eliminerà tutte le istanze, creerà anche nuove istanze da (definstances). Per eliminare permanentemente un'istanza, la funzione **unmake-instance** eliminerà una o tutte le istanze, a seconda del suo argomento. Per eliminare *tutte* le istanze, utilizzare "*****".

Gli esempi seguenti illustrano il comando (unmake-instance).

```
; Delete all instances
CLIPS> (unmake-instance *)
TRUE
; Check that all are gone
CLIPS> (instances)
; Create new instances again
CLIPS> (reset)
; Check new instances created
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 2 instances.
; Delete a specific instance
CLIPS> (unmake-instance [Dorky])
TRUE
CLIPS> (instances)
[Dorky_Duck] of DUCK
For a total of 1 instance.
CLIPS>
```

Un altro modo per eliminare un'istanza specifica è il **send** di un messaggio **delete**. La sintassi generale della funzione (send) è la seguente.

```
(send <instance-name> <message>)
```

In un comando è possibile specificare un solo nome di istanza e, se si tratta di un nome definito dall'utente, deve essere racchiuso tra parentesi quadre.

Ad esempio, quanto segue farà scomparire Dorky_Duck.

```
; Create new instances again
CLIPS> (reset)
; Check new instances created
CLIPS> (instances)
[Dorky] of DUCK
[Dorky_Duck] of DUCK
```

```
For a total of 2 instances.
CLIPS> (send [Dorky_Duck] delete)
TRUE
CLIPS> (instances)
[Dorky] of DUCK
For a total of 1 instance.
CLIPS>
```

Il "*" in un (send) *non* funzionerà per eliminare tutte le istanze. Il "*" funziona solo con la funzione (unmake). Un'altra alternativa è definire il proprio gestore per l'eliminazione che accetterà il "*" e poi consentirà di inviare i messaggi (send <instance-name> my_delete *).

Un messaggio (send) viene gestito solo da un **oggetto target** che ha un gestore appropriato. CLIPS fornisce automaticamente gestori per print, init, delete e così via per ciascuna classe user-defined. È importante comprendere che il messaggio (send [Dorky_Duck] delete) funziona solo perché *questa istanza è una classe user-defined*. Se si definiscono classi che *non* ereditano da USER come una sottoclasse di INTEGER, si *devono* anche creare gestori appropriati per eseguire tutte le attività desiderate come stampare, creare ed eliminare le istanze. È molto più semplice definire sottoclassi di USER e sfruttare i gestori forniti dal sistema.

Cosa hai mangiato a colazione

La funzione (send) è il cuore dell'operazione OOP poiché è *l'unico modo corretto per comunicare tra gli oggetti*. Secondo il principio dell'incapsulamento degli oggetti, a un oggetto dovrebbe essere consentito di accedere ai dati di un altro oggetto solo inviando un messaggio.

Ad esempio, se qualcuno vuole sapere in cosa consisteva la colazione, generalmente lo chiederà, ovvero invierà un messaggio. Un'alternativa scortese sarebbe aprire la bocca e sbirciare in gola. Se il principio dell'incapsulamento degli oggetti non viene seguito, qualsiasi oggetto può hacherare le parti private di altri oggetti, con risultati potenzialmente disastrosi.

Un'utile applicazione di (send) è stampare informazioni su un oggetto. Finora tutti gli esempi di oggetti visti non hanno struttura. Tuttavia, proprio come deftemplate dà la struttura ad un pattern di regole, gli slot danno la struttura oggetto. Sia per deftemplate che per gli oggetti, uno slot è una posizione denominata in cui è possibile archiviare i dati. Tuttavia, a differenza degli slot del deftemplate, gli oggetti ottengono i propri slot dalle classi e le classi utilizzano l'ereditarietà. Pertanto, le informazioni negli slot degli oggetti possono essere effettivamente ereditate dagli oggetti delle sottoclassi. Uno slot **unbound** è uno slot a cui non sono assegnati valori. Tutti gli slot *devono* essere bound.

Come semplice esempio, creiamo un oggetto con degli slot per contenere informazioni personali per poi inviargli messaggi. I seguenti comandi configureranno innanzitutto l'ambiente CLIPS con i costrutti appropriati. Gli slot denominati *sound* e *age* inizialmente non contengono dati, ovvero valori *nil*.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(slot sound)
(slot age))
CLIPS>
(definstances DORK_OBJECTS
(Dorky_Duck of DUCK))
CLIPS> (reset)
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound nil)
(age nil)
CLIPS>
(send [Dorky_Duck] put-sound quack)
quack
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(age nil)
CLIPS>
```

Si noti che gli slot vengono stampati nell'ordine definito nella classe. Tuttavia, se l'istanza eredita slot da più di una classe, gli slot delle classi più generali verranno stampati per primi.

Il valore di uno slot viene modificato utilizzando il messaggio put-. Per default, CLIPS crea un gestore put per ogni slot di una classe per gestire i messaggi put-. Notare il trattino "-" alla fine di "put-". Il trattino è una parte essenziale della sintassi del messaggio poiché separa il "put" dal nome dello slot. In un (send) è consentito solo un "put-". Pertanto, per modificare più slot o lo stesso slot di più istanze, è necessario inviare più messaggi. Invece di farlo manualmente, è possibile scrivere una funzione per eseguire più invii o utilizzare la funzione **modify-instance**.

Il valore di uno slot può essere impostato da uno slot-override in una make-instance. Come esempio,

```
CLIPS>
(make-instance Dixie_Duck of DUCK
(sound quack) (age 2))
[Dixie_Duck]
CLIPS> (send [Dixie_Duck] print)
[Dixie_Duck] of DUCK
(sound quack)
(age 2)
CLIPS>
```

Il messaggio complementare a “put-” è *get-* che ottiene i dati da uno slot, come mostrato nell'esempio seguente. Se un *put-* ha successo restituisce il nuovo valore, mentre se *get-* ha successo restituisce i dati appropriati. Se *put-* o *get-* non hanno esito positivo, verrà restituito un messaggio di errore. Gli esempi seguenti mostrano come funziona.

```
; No slot color
CLIPS>
(send [Dorky_Duck] put-color white)
[MSGFUN1] No applicable primary message-handlers found for put-color.
FALSE
CLIPS> (send [Dorky_Duck] get-age)
nil
; Value put in age
CLIPS> (send [Dorky_Duck] put-age 1)
1
; Check value is correct
CLIPS> (send [Dorky_Duck] get-age)
1
CLIPS>
```

A differenza del messaggio *put-*, il messaggio *get- restituisce* il valore di uno slot. Poiché viene restituito il valore di *get-*, può essere utilizzato da un'altra funzione, assegnato a una variabile e così via. Al contrario, un valore stampato non può essere utilizzato da un'altra funzione, assegnato a una variabile e così via perché il valore va al dispositivo di output standard. Un modo per aggirare questo problema è stampare su un file e poi leggere i dati dal file. Sebbene questa non sia una soluzione elegante, funziona. Un altro modo è scrivere il proprio gestore di messaggi print che restituisca anche valori.

Un punto molto importante sugli slot è che non è possibile modificare gli slot di una classe aggiungendo slot, eliminando slot o modificando le caratteristiche degli slot. L'unico modo per modificare una classe è (1) eliminare tutte le istanze della classe e (2) utilizzare una (defclass) con lo stesso nome di classe e gli slot desiderati. Questa situazione è analoga alla modifica di una regola caricando una nuova regola con lo stesso nome.

Etichetta di classe

Dopo aver visto gli slot e le istanze, è il momento di parlare dell'**etichetta della classe**. Il termine *etichetta* si riferisce a una serie di linee guida per fare qualcosa.

A differenza della programmazione procedurale standard, il paradigma OOP è **orientato alle classi**. Ogni oggetto è intrinsecamente correlato a una classe e quella classe fa parte di una gerarchia di classi. Invece di concentrarsi innanzitutto sulle azioni, il programmatore OOP considera la gerarchia complessiva delle classi o l'**architettura delle classi** e il modo in cui i messaggi verranno inviati tra gli oggetti. Pertanto, le azioni nei programmi procedurali consueti vengono eseguite esplicitamente, mentre le azioni

vengono eseguite implicitamente nell'OOP. In entrambi i casi, il risultato finale è lo stesso. Tuttavia, il sistema OOP può essere verificato, convalidato e mantenuto più facilmente. L'uso corretto delle classi è riassunto nelle seguenti *Rules of Class Etiquette*:

1. La gerarchia delle classi dovrebbe essere in incrementi logici specializzati utilizzando i link is-a.
2. Una classe non è necessaria se ha una sola istanza.
3. Una classe non dovrebbe avere il nome di un'istanza e viceversa.

La prima regola scoraggia la creazione di una singola classe per l'applicazione. Se una singola classe è adeguata, probabilmente non c'è bisogno dell'OOP. Creando classi in incrementi, si può verificare, convalidare e gestire più facilmente il codice. Inoltre, le gerarchie di classi incrementali possono essere facilmente inserite nelle librerie di classi per facilitare notevolmente la creazione di nuovo codice. Questo concetto di librerie di classi è analogo alle librerie di subroutine per le azioni. È possibile utilizzare solo i link is-a poiché questa è l'unica relazione supportata dalla versione 6.3 di CLIPS.

La seconda regola incoraggia l'idea che le classi siano intese come template per produrre più oggetti dello stesso tipo. Ovviamente si può iniziare con zero o una istanza. Tuttavia, se non ci sarà mai bisogno di più di un'istanza in una classe, si dovrebbe considerare di modificare la superclasse per adattarla all'istanza invece di definire una nuova sottoclasse. Se tutte le classi hanno una sola istanza, è probabile che l'applicazione semplicemente non sia adatta all'OOP e che la codifica in un linguaggio procedurale potrebbe essere la soluzione migliore.

La terza regola significa che le classi non dovrebbero avere il nome di istanze e *viceversa*, per evitare confusione.

Altre funzionalità

Esistono numerose funzioni predicato utili per gli slot. Se si usano queste funzioni predicato per verificare i valori appropriati per le funzioni, il programma sarà più robusto contro i crash anomali. In generale, se una funzione non restituisce TRUE, restituisce FALSE. Le funzioni slot fornite da CLIPS includono:

class-slot-exists: Restituisce TRUE se lo slot della classe esiste.
slot-existp: Restituisce TRUE se lo slot dell'istanza esiste.
slot-boundp: Restituisce TRUE se lo slot specificato ha un valore.
instance-address: Restituisce l'indirizzo macchina in cui è archiviata l'istanza specificata **instance-name**: Restituisce il nome dato a un'istanza.
instancep: Restituisce TRUE se il suo argomento è un'istanza.
instance-addressp: Restituisce TRUE se il suo argomento è un indirizzo di istanza.
instance-namep: Restituisce TRUE se il suo argomento è un nome di istanza.
instance-existp: Restituisce TRUE se l'istanza esiste.
list-definstances: Elenca tutte le definstances.
ppdefinstances: Stampa formattata della definstance.
watch instances: Consente di osservare [watch] le istanze create ed eliminate.
unwatch instances: Disattiva l'osservazione [watch] delle istanze.
save-instances: Salva le istanze in un file.
load-instances: Carica le istanze da un file.
undefinstances: Elimina la definstance nominata.

Capitolo 10

Facet affascinanti

Se vuoi avere classe, comportati, vestiti e parla come i tuoi amici.

In questo capitolo vedremo di più sugli slot e su come specificarne le caratteristiche utilizzando le **facet**. Proprio come gli slot descrivono le istanze, le facet descrivono gli slot. L'uso delle facet è una buona ingegneria del software perché c'è una maggiore possibilità che CLIPS segnali un valore illegale piuttosto che rischiare un errore a runtime o un crash. Esistono molti tipi di facet utilizzabili per specificare gli slot, come riepilogato nell'elenco seguente:

default e **default-dynamic**: Impostano i valori iniziali per gli slot. **cardinality**: Numero di valori multicampo.
access: Accesso read-write, read-only, initialize-only.
storage: Slot locale nell'istanza o slot condiviso nella classe.
propagation: Eredita o non eredita gli slot.
source: Eredità composita o esclusiva.
override-message: Indica il messaggio da inviare per l'override dello slot.
create-accessor: Crea il gestore put e get.
visibility: Pubblica o privata solo per la definizione della classe.
reactive: Modifica il trigger di pattern-matching dello slot.

Per motivi di spazio, nel resto di questo capitolo descriveremo più in dettaglio solo alcune facet. Per ulteriori dettagli, consultare il *CLIPS Reference Manual*.

Uno slot denominato Default

Il **default facet** imposta il valore di default di uno slot quando un'istanza viene creata o inizializzata, come mostrato nell'esempio seguente.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(slot sound (default quack))
(slot ID)
(slot sex (default male)))
CLIPS> (make-instance Dorky_Duck of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID nil)
(sex male)
CLIPS>
```

Come si vede, i valori predefiniti per il sesso e lo slot audio sono stati impostati sui valori delle facet di default. Dopo la parola chiave default può esserci qualsiasi espressione CLIPS valida che non coinvolga una variabile. Ad esempio, l'espressione di default dello slot audio è il simbolo quack. Le funzioni possono essere utilizzate nella **espressione facet** come verrà mostrato nel prossimo esempio.

Questo facet di default è un **default statico** perché il valore della sua espressione facet viene determinato quando la classe viene definita e non viene mai modificato a meno che la classe non venga ridefinita. Per esempio, impostiamo il valore di default dello slot ID sulla funzione (gensym*) che restituisce un nuovo valore non presente nel sistema ogni volta che viene chiamata.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(slot sound (default quack))
(slot ID (default (gensym*)))
(slot sex (default male)))
CLIPS> ; Dorky_Duck #1
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
```

```
[Dorky_Duck] of DUCK
(sound quack)
(ID gen1)
(sex male)
CLIPS> ; Dorky_Duck #2
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen1)
(sex male)
CLIPS>
```

Come si vede, l'ID è sempre gen1 poiché (gensym*) viene valutata solo una volta e non più quando viene creata la seconda istanza. Notare che i valori (gensym*) potrebbero essere diversi sul proprio computer se è già stata chiamata (gensym*) poiché aumenta di uno ogni volta che viene chiamato e *non* viene resettata da un (clear). La funzione (gensym*) viene resettata al suo valore iniziale solo se si riavvia CLIPS.

Supponiamo ora di voler tenere traccia delle diverse istanze di Dorky_Duck che sono state create. Invece di utilizzare il default statico, possiamo utilizzare la facet chiamata **default dinamico** che valuterà la sua espressione di facet ogni volta che viene creata una nuova istanza. Notare la differenza tra l'esempio seguente e quello precedente.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(slot sound (default quack))
(slot ID (default-dynamic (gensym*)))
(slot sex (default male)))
CLIPS> ; Dorky_Duck #1
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen2)
(sex male)
CLIPS> ; Dorky_Duck #2
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(ID gen3)  ; Note ID is different
(sex male) ; from Dorky_Duck #1
CLIPS>
```

In questo esempio che utilizza il default dinamico, l'ID della seconda istanza, gen3, è diverso dalla prima istanza, gen2. Al contrario, per l'esempio precedente di default statico, i valori ID erano gli stessi, gen1, poiché (gensym*) veniva valutato solo una volta quando la classe veniva definita anziché per ogni nuova istanza in caso di default dinamico.

Proprietà cardinali

La **cardinalità** di uno slot si riferisce a uno dei due tipi di campi che uno slot può contenere: (1) campo singolo o (2) multicampo. Il termine *cardinalità* si riferisce a un conteggio. Uno slot a campo singolo associato [bound] contiene un solo campo, mentre uno slot a campo multiplo associato [bound] può contenere zero o più campi. Lo slot bound a campo singolo e lo slot bound multicampo contengono ciascuno *un* valore. Tuttavia, l'unico valore multicampo può contenere più campi. Ad esempio, (a b c) è un singolo valore multicampo con tre campi. La stringa vuota "" è un valore a campo singolo, proprio come lo è "a b c". Al contrario, uno slot non-bound non ha valore.

Per analogia con le variabili a campo singolo e multicampo, si pensi a uno slot come alla casella di posta. A volte si può trovare un singolo messaggio di posta indesiderata senza busta. Invece, un'etichetta con l'indirizzo è stata appena attaccata su un pezzo di carta piegato indirizzato a "Residente". Altre volte si trova una busta con più annunci al suo interno. Il singolo pezzo di posta indesiderata senza busta è come un valore a campo singolo mentre la busta con più annunci è come il valore multicampo. Se il distributore di posta indesiderata commette un errore e invia una busta senza nulla all'interno, ciò

corrisponde alla variabile multicampo vuota. (Ora che ci penso, se la busta della posta indesiderata è vuota, hai davvero ricevuto posta indesiderata?)

Un **facet multiplo** con la parola chiave **multislot**, viene utilizzato per archiviare un valore multicampo come mostrato nell'esempio seguente.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(multislot sound (default quack quack)))
CLIPS> (make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack quack)
CLIPS>
```

È possibile accedere a un valore multicampo utilizzando get- e put- come mostrato negli esempi seguenti, che mostrano come tenere traccia dei quack.

```
CLIPS>
(send [Dorky_Duck] put-sound
quack1 quack2 quack3)
(quack1 quack2 quack3)
CLIPS> (send [Dorky_Duck] get-sound)
(quack1 quack2 quack3)
CLIPS>
```

È possibile utilizzare funzioni CLIPS standard come nth$ per ottenere l'ennesimo campo di un valore multislot. L'esempio seguente mostra come scegliere un certo quack.

```
CLIPS> (nth$ 1 (send [Dorky_Duck] get-sound))
quack1
CLIPS> (nth$ 2 (send [Dorky_Duck] get-sound))
quack2
CLIPS> (nth$ 3 (send [Dorky_Duck] get-sound))
quack3
CLIPS>
```

Altre funzionalità

CLIPS ha diverse **funzioni slot multicampo** come mostrato nell'elenco seguente.

slot-replace$: Sostituisce l'intervallo specificato.

slot-insert$: Inserisce l'intervallo specificato.

slot-delete$: Elimina l'intervallo specificato.

Uno slot multicampo senza valori, ad esempio il valore multicampo vuoto (), può essere assegnato a uno slot con una facet (multiple). Notare che esiste una differenza tra uno slot con un valore multicampo vuoto () e uno slot non bound. Se si pensa a un valore multicampo vuoto come l'analogo di un bus vuoto, si vede che c'è una differenza tra nessuna persona (slot non bound) e un bus senza persone (valore multicampo vuoto, ()).

Una **create-accessor** indica a CLIPS se creare gestori **put-** e **get-** per uno slot. Per default, tutti gli slot hanno un "create-accessor" **read-write**, quindi non è necessario specificare questo facet per creare handler. Se si definiscono gli handler, si deve utilizzare ?NONE col facet create-accessor. Gli altri tipi di facet per *create accessor* sono **read** e **write**.

Una **storage facet** definisce uno dei due posti in cui viene memorizzato il valore di uno slot: (1) in un'istanza o (2) nella classe. Un valore di slot memorizzato nell'istanza è detto **local** perché il valore è noto solo all'istanza. Pertanto, possono esistere istanze diverse con valori di slot locali diversi. Al contrario, un valore di slot memorizzato in una classe è detto **shared** perché è lo stesso per *tutte* le istanze della classe.

Un valore locale è specificato dal **local facet**, che è il default per uno slot. Un valore shared è specificato dalla **shared facet** e *tutte* le istanze con questo tipo di slot avranno il valore dello slot modificato automaticamente se *uno* cambia. Una **access facet** definisce uno dei tre tipi di accesso per uno slot, sia che si utilizzino i gestori creati da CLIPS sia che si definiscano i propri. Il tipo di default, **read-write**, consente sia di leggere che di scrivere il valore dello slot. Gli altri tipi sono **readonly** e **initialize-only**.

Un altro modo per impostare i valori dell'istanza è con la funzione **initialize-instance**. È possibile richiamare un'istanza (initialize-instance) in qualsiasi momento per resettare i valori di default e conservare i valori negli slot non(default).

Un (reset) può essere considerato come una **inizializzazione a freddo** poiché tutti i valori negli slot non-(default) vengono cancellati, mentre una (initialize-instance) può essere considerata una **warm-initialization** poiché i valori non(default) vengono mantenuti. Naturalmente, solo le (definstances) possono essere inizializzate a freddo poiché le non(definstances) verranno semplicemente eliminate. Inoltre, gli slot-override possono essere utilizzati in (initialize-instance) come mostra l'ultimo esempio.

Due funzioni predicato sono progettate per l'uso con le access facet. Entrambe queste funzioni predicato restituiscono un messaggio di errore se lo slot o l'istanza specificata non esiste. **slotwritablep** è una funzione predicato che restituisce TRUE se uno slot è scrivibile e FALSE se non lo è. La funzione predicato **slot-initablep** restituisce TRUE se lo slot è inizializzabile e FALSE in caso contrario. Il termine *inizializzabile* significa che *non* è read-only.

La **inherit facet**, che è il default, specifica che le istanze indirette di una classe possono ereditare questo slot dalla classe. Come ricorderete, le istanze indirette di una classe sono le istanze delle sue sottoclassi, mentre le istanze dirette sono quelle definite specificatamente per la classe. Le istanze indirette di una classe sono istanze dirette delle sottoclassi in cui sono definite. Ad esempio, [Dorky_Duck] è un'istanza diretta di DUCK e un'istanza indiretta di USER che è la superclasse di DUCK. La **no-inherit facet** specifica che solo un'istanza diretta ha lo slot di classe.

È importante rendersi conto che la facet (no-inherit) proibisce solo l'ereditarietà dalla classe e non dalle sue superclassi. Ciò significa che un'istanza può ancora ereditare dalle superclassi della classe (no-inherit).

La **composite facet**. facet afferma che le facet che non sono *esplicitamente* definite nella classe di precedenza più alta prendono le loro facet dalla classe di precedenza immediatamente più alta. Se la facet non è impostata esplicitamente nella classe con precedenza più alta *ed è anch'essa composita*, CLIPS prova quella immediatamente superiore e così via finché la facet non viene definita o non ci sono più classi. Se la classe immediatamente superiore non è composita, CLIPS non effettua ulteriori controlli. L'opposto della facet composita è la **exclusive facet**, che è il default. Per ulteriori informazioni, consultare il *CLIPS Reference Manual*.

Capitolo 11

Gestire i gestori

Ci sono due passaggi per imparare a usare una pala: (1) scoprire quale parte è il manico e (2) cosa fare con il manico.

Gli handler sono essenziali in OOP perché supportano l'incapsulamento degli oggetti. L'unico modo corretto in cui gli oggetti possono rispondere ai messaggi è disporre di un gestore [handler] appropriato per ricevere il messaggio ed eseguire l'azione appropriata. In questo capitolo vedremo come i messaggi vengono interpretati dagli oggetti. Vedremo come modificare i gestori di messaggi esistenti e come scriverne di propri.

L'io interiore

Finora abbiamo visto la struttura statica delle classi attraverso la gerarchia di ereditarietà e gli slot. Tuttavia, la parte dinamica di un oggetto è determinata dai suoi gestori dei messaggi, o handler in breve, che ricevono messaggi ed eseguono azioni in risposta. I gestori sono responsabili delle proprietà dinamiche di un oggetto, che ne determinano il comportamento. Abbiamo già utilizzato un gestore molte volte: *print* in a (send).

Polimorfismo; è una delle cinque caratteristiche generalmente accettate di un vero linguaggio OOP. Ad esempio, lo stesso tipo di messaggio, (send <instance-name> print), può avere azioni diverse a seconda della classe dell'oggetto che lo riceve. Gli oggetti DUCK possono stampare suono, ID ed età, mentre gli oggetti DUCKLING stampano solo suono ed età.

Nei linguaggi senza polimorfismo, si deve definire una funzione, (send-duckprint) per i tipi duck e un'altra funzione, (send-duckling-print) per il tipo duckling. Tuttavia, in OOP, indipendentemente dal numero di classi definite, lo stesso messaggio (send <instance-name> print), stamperà gli slot dell'oggetto. Ciò migliora notevolmente l'efficienza dello sviluppo del programma poiché non è necessario definire nuove funzioni per ogni nuovo tipo.

Il polimorfismo può essere portato agli estremi facendo sì che lo stesso messaggio faccia cose completamente diverse. Ad esempio, si potrebbe definire un gestore di messaggi di stampa che esegue la *print* gli oggetti di una determinata classe e il *delete* degli oggetti di un'altra classe. Un'altra possibilità estrema sarebbe quella di non far stampare nulla dal gestore di stampa. Invece, eliminerebbe oggetti di una classe, salverebbe oggetti di un'altra classe, aggiungerebbe oggetti di un'altra classe e così via.

Questo uso estremo del polimorfismo creerebbe la Torre di Babele di un programmatore e renderebbe molto difficile la comprensione del codice poiché sarebbe tutto dipenderebbe dal run-time. Definire gestori di messaggi con lo stesso nome in classi diverse che fanno cose completamente diverse va contro il Principio del Minimo Stupore [Principle of Least Astonishment].

Un altro esempio di polimorfismo potrebbe essere definito utilizzando un gestore di messaggi per "+". Se un messaggio per "+" viene inviato a stringhe o simboli, ovvero oggetti LEXEME, questi verranno concatenati a causa di un gestore definito per la classe LEXEME. Se il "+" viene inviato a numeri ordinari, il risultato è un'addizione a causa del gestore di sistema predefinito per l'addizione dei numeri. Se il "+" viene inviato a numeri complessi, definiti come una sottoclasse di USER chiamata COMPLEX, un gestore definito per la classe COMPLEX eseguirà l'addizione di numeri complessi. Pertanto, il “+” esegue tipi di operazioni correlate che corrispondono alla nostra intuizione e non ci sorprendono.

Oltre ai gestori predefiniti come *print*, se ne possono definire dei propri gestori. Iniziamo scrivendo un gestore per aggiungere numeri tramite messaggi.

Come si può vedere dalla Fig. 8.6 del capitolo 8, la classe NUMBER ha le sottoclassi INTEGER e FLOAT. Trattandosi di classi predefinite, sarebbe naturale eseguire calcoli numerici inviando messaggi a numeri. Proviamolo in questo modo.

```
CLIPS> (clear)
CLIPS> (send 1 + 2)
[MSGFUN1] No applicable primary message-handlers found for +.
FALSE
CLIPS>
```

Bene, come vede, questo esempio non ha funzionato. Il motivo è implicito nel messaggio di errore. Proviamolo ottenendo maggiori informazioni sulla classe INTEGER poiché quella era l'oggetto target del messaggio di stampa.

```
CLIPS> (describe-class INTEGER)
============================================================
==
************************************************************
**
Abstract: direct instances of this class cannot be created.

Direct Superclasses: NUMBER
Inheritance Precedence: INTEGER NUMBER PRIMITIVE OBJECT
Direct Subclasses:
************************************************************
**
============================================================
==
CLIPS>
```

Il problema è che la classe INTEGER non ha un gestore per "+". In effetti, non ha alcun gestore poiché non ne è elencato nessuno. Come ricorderete, la classe USER e le sue sottoclassi hanno sempre print, delete e altri gestori definiti automaticamente da CLIPS. Se vogliamo inviare messaggi di stampa a un oggetto come 1 della classe INTEGER, dovremo creare il nostro gestore di stampa.

Prima di scrivere questo gestore, rispondiamo a una domanda riguardo a come INTEGER può avere istanze. Poiché INTEGER è una classe astratta, ci si potrebbe chiedere come può avere istanze come 1, 2, 3, ecc. Sebbene non sia possibile creare istanze dirette di una classe astratta, è possibile utilizzare istanze esistenti. Nel caso della classe di sistema predefinita INTEGER sono disponibili come oggetti tutti i numeri interi fino al massimo consentito. Allo stesso modo, stringhe e simboli sono disponibili per le classi astratte definite dal sistema STRING e SYMBOL, e così via per le altre classi predefinite.

Di seguito è riportata la definizione di un gestore per la classe NUMBER che gestirà l'addizione tramite messaggi. Il gestore è definito per NUMBER anziché per INTEGER perché vorremmo gestire anche oggetti FLOAT. Invece di definire lo stesso gestore per FLOAT e per INTEGER, è più semplice definire semplicemente un gestore per la superclasse NUMBER. *Se un gestore per un messaggio non è definito nella classe dell'oggetto, CLIPS prova tutti i gestori nella lista di precedenza dell'ereditarietà.* Poiché “+” non è definito per INTEGER, CLIPS prova successivamente con NUMBER, trova il gestore applicabile e restituisce il risultato di 3.

```
CLIPS>
; ?arg is argument of handler
(defmessage-handler NUMBER + (?arg)
; Function addition of handler
(+ ?self ?arg))
CLIPS> (send 1 + 2)
3
CLIPS> (send 2.5 + 3)
5.5
CLIPS> (send 2.5 + 2.6)
5.1
CLIPS> (describe-class NUMBER)
============================================================
==
************************************************************
**
Abstract: direct instances of this class cannot be created.

Direct Superclasses: PRIMITIVE
```

```
Inheritance Precedence: NUMBER PRIMITIVE OBJECT
Direct Subclasses: INTEGER FLOAT
-------------------------------------------------------------
--
Recognized message-handlers:
+ primary in class NUMBER
*************************************************************
**
=============================================================
==
CLIPS>
```

La variabile **?self** è una variabile speciale in cui CLIPS memorizza l'**istanza attiva**. *?self* è una parola riservata che non può essere inclusa esplicitamente in un argomento dell'handler, né può essere associata [bound] a un valore diverso. L'istanza attiva è l'istanza a cui è stato inviato il messaggio. Nel nostro esempio, tutte le classi predefinite come NUMBER, INTEGER e FLOAT sono tutte sottoclassi della classe PRIMITIVE. Ciò è in contrasto con USER, che è l'altra sottoclasse principale di OBJECT.

Come altro esempio, scriviamo un handler per concatenare stringhe, simboli o entrambi.

```
CLIPS>
; ?arg is argument of handler
(defmessage-handler LEXEME + (?arg)
; Function concatenation of handler
(sym-cat ?self ?arg))
; SYMBOL + STRING
CLIPS> (send Dorky_ + "Duck")
Dorky_Duck ; SYMBOL result
CLIPS>
```

Si noti che il gestore è definito per la classe LEXEME poiché questa è una superclasse sia di SYMBOL che di STRING. In questo caso, il gestore restituisce un SYMBOL poiché viene utilizzato (sym-cat).

Questo esempio illustra anche perché le parentesi potrebbero essere necessarie in un (send). Come mostrato in questo esempio, il messaggio va al SYMBOL *Dorky_* senza parentesi. Con le parentesi, il messaggio va a un oggetto *[Dorky_]* di una classe user-defined. Qui assumiamo che *[Dorky_]* potrebbe essere un oggetto di una classe definita dall'utente, come DUCK.

Falli pagare a caro prezzo

La vera utilità dei gestori è con le sottoclassi di USER poiché si possono definire istanze di queste classi. Per vedere come funzionano gli handler in questo caso, configuriamo prima l'ambiente come segue.

```
CLIPS> (clear)
CLIPS>
(defclass DUCKLING (is-a USER)
(slot sound (default quack))
(slot age (visibility public)))
CLIPS>
(defclass DUCK (is-a DUCKLING)
(slot sound (default quack)))
CLIPS>
(definstances DUCKY_OBJECTS
(Dorky_Duck of DUCK (age 2))
(Dinky_Duck of DUCKLING (age .1)))
CLIPS> (reset)
CLIPS>
```

Come semplice esempio, scriviamo un gestore che stamperà gli slot dell'istanza attiva. Possiamo utilizzare la funzione **ppinstance** per stampare gli slot dell'istanza attiva. Questa funzione *non* restituisce un valore e viene utilizzata solo per l'effetto collaterale della stampa sul dispositivo standard. Inoltre, può essere utilizzata solo dall'interno di un handler poiché è nota solo l'istanza attiva. Di seguito è mostrato un gestore USER-defined chiamato print-slots che stampa gli slot dell'istanza attiva utilizzando (ppinstance).

```
CLIPS>
(defmessage-handler USER print-slots ()
(ppinstance))
CLIPS> (send [Dorky_Duck] print-slots)
[Dorky_Duck] of DUCK
(age 2)
(sound quack)
```

```
CLIPS>
```

Sebbene in questo caso il gestore possa essere definito solo per DUCK, un gestore definito per USER verrà chiamato per tutte le sottoclassi di USER, non solo DUCKLING e DUCK. Pertanto, l'handler print-slots funzionerà per *tutte* le sottoclassi che possiamo definire di USER.

Naturalmente è possibile lasciarsi trasportare e definire tutti i gestori di messaggi come gestori USER. Tuttavia, è buono stile e migliora l'efficienza definire i gestori il più vicino possibile alla classe o alle classi a cui sono destinati. L'efficienza è migliorata perché CLIPS non deve continuare a cercare in molte classi per trovare il gestore applicabile.

Muoversi attorno

Esaminiamo ora i gestori di messaggi in modo più dettagliato. Definiremo un gestore per stampare un'intestazione quando un oggetto riceve un messaggio per stampare se stesso. Il gestore dei messaggi viene definito utilizzando il costrutto defmessage-handler come segue.

```
CLIPS>
(defmessage-handler USER print before ()
(printout t "*** Starting to print ***"
crlf))
CLIPS> (send [Dorky_Duck] print)
*** Starting to print ***
[Dorky_Duck] of DUCK
(age 2)
(sound quack)
CLIPS>
```

Il motivo per cui alla fine viene stampata un'intestazione anziché un trailer alla fine ha a che fare con il **tipo di gestore**. Un gestore di tipo **before** viene utilizzato *prima* del messaggio print. Per creare un trailer, utilizzare il gestore del tipo **after** come mostrato nell'esempio seguente.

```
CLIPS>
(defmessage-handler USER print after ()
(printout t "*** Finished printing ***"
crlf))
CLIPS> (send [Dorky_Duck] print)
*** Starting to print ***
[Dorky_Duck] of DUCK
(age 2)
(sound quack)
*** Finished printing ***
CLIPS>
```

Il formato generale di un gestore di messaggi è il seguente.

```
(defmessage-handler <class-name>
<message-name> [handler-type]
[comment]
(<parameters>* [wildcard-parameter])
<action>*)
```

Anche se in un gestore possono essere presenti più azioni, *viene restituito solo il valore dell'ultima azione*. Notare che questo è proprio come una (deffunction).

Poiché [Dorky_Duck] appartiene alla classe DUCK, una sottoclasse di USER, possiamo sfruttare il gestore **print** che è predefinito da CLIPS per la classe USER. Tutte le sottoclassi di USER possono trarre vantaggio dai gestori di USER, il che fa risparmiare la fatica di scrivere gestori per ogni classe definita. Si noti come il concetto di ereditarietà da USER alla sua sottoclasse DUCK semplifica lo sviluppo del programma consentendo il riutilizzo del codice esistente, ovvero il gestore di stampa di USER.

Le parentesi vuote () che seguono il tipo di gestore *before* indicano che non sono presenti né parametri né parametri jolly. In altre parole, il gestore dell'intestazione non accetta argomenti e quindi le parentesi sono vuote, ma obbligatorie. Si noti che sebbene possano essere utilizzati più parametri, può esserci un solo carattere jolly.

Considerazioni principali

Come si vede, il gestore del trailer è uguale al gestore dell'intestazione, tranne per il fatto che viene utilizzato un tipo di gestore *after* e il testo dell'azione è diverso. Pertanto, un tipo di gestore before esegue la sua attività prima del gestore di tipo **primary** e un gestore

after esegue la sua attività dopo il gestore primary. Un *primary* è destinato a svolgere il compito principale. Un tipo di gestore around ha lo scopo di configurare l'ambiente per il resto dei gestori. I tipi before e after sono destinati ad attività [task] minori come l'inizializzazione di variabili o la stampa, mentre il primary svolge l'attività principale. L'elenco seguente riassume il ruolo della classe e il valore restituito di ciascun tipo di gestore:

around: Configura l'ambiente per altri handler. Restituisce un valore.
before: Lavoro ausiliario prima del primary. Nessun valore restituito.
primary: Esegue l'attività principale del messaggio. Restituisce un valore.
after: Lavoro ausiliario dopo il primary. Nessun valore restituito.

I tipi di handler sono elencati nell'ordine in cui vengono normalmente chiamati durante l'esecuzione di un messaggio. A seconda del tipo di gestore, CLIPS sa quando eseguirlo. Cioè, un gestore **around** inizia prima di qualsiasi handler before. Un handler before viene eseguito prima di qualsiasi gestore primary, seguito dai gestori after. L'eccezione a questa esecuzione sequenziale dell'handler è il gestore del tipo around. Se viene definito un gestore around, inizierà l'esecuzione prima di tutti gli altri, eseguirà le azioni specificate e poi completerà le sue azioni *dopo* che tutti gli altri tipi di gestori avranno terminato. Presto vedremo un esempio dettagliato dell'esecuzione di questi gestori.

La **class role** descrive lo scopo previsto di ciascun tipo. Il valore restituito descrive se il tipo di gestore è generalmente inteso per un **valore restituito** o semplicemente per fornire un utile effetto collaterale come la stampa**.** Questa considerazione dipenderà dal gestore. Ad esempio, molti gestori primary user-defined possono essere scritti per restituire un valore come risultato di alcuni calcoli numerici o operazioni su stringhe. Un'eccezione alla restituzione di un valore di ritorno utile è un gestore primary di stampa la cui attività principale è l'effetto collaterale della stampa e non ha un valore di ritorno.
Di seguito viene mostrato l'elenco dei gestori primary predefiniti di USER e il relativo class role. Per ereditarietà, questi sono disponibili per tutte le sottoclassi di USER.

init: Inizializza un'istanza.
delete: Elimina un'istanza.
print: Stampa un'istanza.
direct-modify: Modifica direttamente gli slot.
message-modify: Modifica gli slot utilizzando i messaggi put.
direct-duplicate: Duplica un'istanza senza utilizzare messaggi put-.
message-duplicate: Duplica un'istanza utilizzando i messaggi.

Questi gestori primary sono predefiniti e non possono essere modificati a meno che non si modifichi il codice sorgente di CLIPS. Tuttavia, è possibile definire i tipi di gestori *before*, *after* e *around* per questi primary. Si è già visto un esempio di modifica dei tipi di gestore before e after per il gestore print di USER. Ora diamo un'occhiata ad alcuni esempi di definizione dei tipi di gestore before e after per il gestore primary **init .**

```
CLIPS>
(defmessage-handler USER init before ()
(printout t
"*** Starting to make instance ***"
crlf))
CLIPS>
(defmessage-handler USER init after ()
(printout t
"*** Finished making instance ***"
crlf))
CLIPS> (reset)
*** Starting to make instance ***
*** Finished making instance ***
*** Starting to make instance ***
*** Finished making instance ***
CLIPS>
(make-instance Dixie_Duck of DUCK
(age 1))
*** Starting to make instance ***
*** Finished making instance ***
[Dixie_Duck]
CLIPS> (instances)
[Dorky_Duck] of DUCK
```

```
[Dinky_Duck] of DUCKLING
[Dixie_Duck] of DUCK
For a total of 3 instances.
CLIPS>
```

Il potere della fede

Il parametro self è utile perché può essere utilizzato per *leggere* un valore di slot nel modulo

```
?self:<slot-name>
```

Può anche essere usato per *scrivere* un valore di slot utilizzando la funzione bind. La notazione "?self:" è più efficiente dell'invio di messaggi ma può essere utilizzata solo dall'interno di un handler sulla sua istanza attiva. Al contrario, le funzioni **dynamic-get-** e **dynamic-put-** possono essere utilizzate dall'interno di un handler per leggere e scrivere un valore di slot dell'istanza attiva. Sebbene sia possibile utilizzare i messaggi dall'interno di un gestore, come il seguente, non è efficiente.

```
(send ?self dynamic-get-<slot>)
(send ?self dynamic-put-<slot>)
```

Come esempio di dynamic-get-, cambiamo il nostro esempio come segue.

```
CLIPS>
(defmessage-handler USER print-age primary ()
(printout t "*** Starting to print ***"
crlf
"Age = " (dynamic-get age)
crlf
"*** Finished printing ***"
crlf))
CLIPS> (send [Dorky_Duck] print-age)
*** Starting to print ***
Age = 2
*** Finished printing ***
CLIPS>
```

?self:age può essere utilizzato solo in una classe e nelle sue sottoclassi che ereditano lo slot age. ? self:<slot-name> viene valutato in modo *statico* ramite ereditarietà. Ciò significa che se una sottoclasse ridefinisce uno slot, un gestore di messaggi della superclasse fallirà se tenta di accedere direttamente allo slot utilizzando ? self:<slot-name>.

Al contrario, dynamic-get- e dynamic-put- sono utilizzabili da superclassi e sottoclassi perché queste controllano gli slot *dinamicamente*. Affinché una superclasse possa fare riferimento dinamicamente a uno slot, tuttavia, la facet ddi visibilità dello slot deve essere pubblico. L'esempio seguente *non* funzionerebbe se DUCK venisse cambiato in USER.

```
CLIPS>
(defmessage-handler DUCK print-age primary ()
(printout t "*** Starting to print ***"
crlf
"Age = " ?self:age crlf
"*** Finished printing ***"
crlf))
CLIPS> (send [Dorky_Duck] print-age)
*** Starting to print ***
Age = 2
*** Finished printing ***
CLIPS>
```

Come esempio di utilizzo di una funzione dynamic-put- in un handler, supponiamo di voler aiutare Dorky_Duck a ritrovare un po' della sua giovinezza. L'esempio seguente mostra come è possibile modificare la sua età utilizzando un handler. Questo esempio illustra anche come un valore può essere passato a un gestore tramite la variabile ?change.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(slot sound (default quack))
(slot age))
CLIPS>
(defmessage-handler DUCK lie-about-age
(?change)
(bind ?new-age (- ?self:age ?change))
(dynamic-put age ?new-age)
(printout t "*** Starting to print ***"
crlf
"I am only " ?new-age
```

```
crlf
"*** Finished printing ***"
crlf))
CLIPS>
(make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 2)
*** Starting to print ***
I am only 1
*** Finished printing ***
CLIPS>
```

Come si vede, la convinzione di Dorky_Duck è così forte che l'età modificata viene inserita nel suo slot age. Notare come l'handler utilizza la variabile ?new-age per memorizzare l'età modificata, che viene poi inserita nello slot age dell'istanza.

La dura verità

Un **demone** è un handler che viene eseguito ogni volta che viene eseguita un'**azione di base** come l'inizializzazione, l'eliminazione, il recupero o l'inserimento su un'istanza. Una regola non può essere considerata un demone perché non è sicuro che verrà eseguita solo perché i suoi pattern a sinistra [LHS] sono soddisfatti. L'unica cosa certa è che una regola verrà *attivata* quando il suo LHS sarà soddisfatto, non che verrà eseguita.

Non esiste una parola chiave esplicita per demone poiché è solo un concetto. Gli handler before e after che stampavano le stringhe possono essere considerati demoni di stampa. Questi handler hanno aspettato un messaggio (send [Dorky_Duck] print-age) per attivare la loro azione. Prima il gestore before ha stampato la sua stringa, poi ha stampato il gestore primary e infine ha stampato il gestore after. Un demone è il gestore before che attende un messaggio di stampa. Il secondo demone è il gestore after che attende che il print primary finisca di stampare.

La stampa non è considerata un'azione di base perché non esiste alcun valore di ritorno associato a (send <instance> print). Il messaggio di stampa viene inviato solo per l'effetto collaterale della stampa. Al contrario, un messaggio (send <instance> get-<slot>) restituirà un valore che può essere utilizzato da altro codice. Allo stesso modo, l'inizializzazione, l'eliminazione e il put hanno tutti un effetto su un'istanza e quindi sono considerate azioni di base come get.

I demoni vengono facilmente implementati utilizzando i gestori before e after poiché questi verranno eseguiti prima e dopo il loro gestore primary. L'implementazione di demoni come questa è detta **implementazione dichiarativa** perché non è necessaria alcuna azione esplicita da parte dell'handler per essere eseguita. Cioè, CLIPS eseguirà sempre un handler before prima del suo gestore primary ed eseguirà sempre un handler after dopo il suo gestore primary. In un'implementazione dichiarativa del demone, il normale funzionamento di CLIPS farà sì che i demoni vengano attivati quando sarà giunto il loro momento. Pertanto, l'implementazione dichiarativa è implicita nel normale funzionamento.

L'opposto dell'esecuzione implicita è l'**implementazione imperativa** in cui le azioni sono programmate esplicitamente. L'handler around è molto comodo da usare per i demoni imperativi. L'idea di base dell'handler around è la seguente.

1. Inizia prima di qualsiasi altro handler.
2. Chiama l'handler successivo utilizzando **call-next-handler** per passare gli stessi argomenti o **override-next-handler** per passarne di diversi.
3. Continua l'esecuzione al termine dell'ultimo handler.
4. Al termine di qualsiasi altro handler around, before, primary, o after, il gestore around riprende l'esecuzione.

La parola chiave *call-next-handler* viene utilizzata per chiamare il/i successivo/i gestore/i. Si dice che un handler sia *shadowed* da uno **shadower** se deve essere chiamato dallo shadower da una funzione come call-next-handler. L'handler call-next-handler può essere utilizzato più volte per chiamare gli stessi handler.

Una funzione predicato chiamata **next-handlerp** viene utilizzata per verificare l'esistenza di un gestore prima che venga effettuata la chiamata. Se non esiste alcun gestore, next-handlerp restituirà FALSE.

L'esempio seguente illustra l'handler around attraverso un demone veritiero che dice a Dorky_Duck ogni volta che mente sulla sua età.

```
CLIPS>
(defmessage-handler DUCK lie-about-age around
(?change)
(bind ?old-age ?self:age)
(if (next-handlerp) then
(call-next-handler))
(bind ?new-age ?self:age)
(if (<> ?old-age ?new-age) then
(printout t "Dorky_Duck is lying!"
crlf
"Dorky_Duck is lying!"
crlf
"He's really " ?old-age
crlf)))
CLIPS>
(make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 1)
*** Starting to print ***
I am only 2
*** Finished printing ***
Dorky_Duck is lying!
Dorky_Duck is lying!
He's really 3
CLIPS>
```

Sebbene Dorky_Duck possa ancora mentire sulla sua età, il demone dice la verità.

Notare l'argomento ?change. Anche se l'handler around non utilizza ?change, il primary lieabout-age chiamato dal call-next-handler ne ha bisogno per modificare l'età. Pertanto, ? change deve essere passato al primary dall'handler around Se si tralascia ?change, verrà visualizzato un messaggio di errore.

l'handler call-next-handler passa *sempre* gli argomenti dello shadower al gestore shadowed. È possibile passare *diversi* argomenti a un gestore shadowed utilizzando la funzione override-next-handler come mostrato nell'esempio seguente.

```
CLIPS>
(defmessage-handler DUCK lie-about-age around
(?change)
(bind ?old-age ?self:age)
(if (next-handlerp) then
; Divide age in half!
(override-next-handler (/ ?change 2)))
(bind ?new-age ?self:age)
(if (<> ?old-age ?new-age) then
(printout t "Dorky_Duck is lying!"
crlf
"Dorky_Duck is lying!"
crlf
"He's really " ?old-age
crlf)))
CLIPS>
(make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 1)
*** Starting to print ***
I am only 2.5
*** Finished printing ***
Dorky_Duck is lying!
Dorky_Duck is lying!
He's really 3
CLIPS>
```

È importante tenere presente che il valore restituito da call-next-handler e override-next-handler è quello degli handler shadowed.

Di seguito sono riportate le Regole di Invio dei Messaggi:

1. Tutti gli handler around iniziano l'esecuzione. Poi gli handler before, primary e after partono e finiscono, seguiti dal completamento dell'esecuzione dell'around.
2. Gli handler around, before e primary vengono chiamati in ordine dalla classe di precedenza più alta a quella più bassa.

3. Gli handler after vengono chiamati dalla classe di precedenza più bassa a quella più alta. 4. Ciascun handler around deve chiamare esplicitamente il successivo handler shadowed.
4. I primary con precedenza più alta devono esplicitamente chiamare i primary con precedenza più bassa (shadowed) se devono essere eseguiti.

Tuttavia, si tenga presente il seguente prerequisito per qualsiasi handler dei messaggi:

Deve essere presente almeno un handler primary applicabile.

Poiché solo i gestori around e primary possono restituire valori e i gestori around shadows, ne consegue che il valore restituito di un (send) sarà il valore restituito around. Se non c'è intorno, il valore restituito sarà quello del primary con la precedenza più alta.

L'elenco seguente riepiloga il valore restituito dei tipi di handler.

around: Ignora o acquisisce il valore restituito del successivo around o primary più specifico. **before**: Ignora. Solo l'effetto collaterale.

primary: Ignora o acquisisce il valore restituito dello scope primary più generale. **after**: Ignora. Solo l'effetto collaterale.

Tiriamo le somme

Finora abbiamo discusso dell'ereditarietà utilizzando solo i collegamenti is-a. Come visto, questo tipo di relazione di ereditarietà è utile per definire classi sempre più specializzate. Cioè, si inizia definendo le classi più generali come sottoclasse di USER, poi si definiscono classi più specializzate con più slot nei livelli inferiori della gerarchia delle classi.

Normalmente, si progettano nuove classi come specializzazioni di quelle esistenti. Questo paradigma è **Ereditarietà per Specializzazione**. Come ricorderete dall'esempio del quadrilatero della Figura 8.7 del capitolo 8, il livello più alto è il quadrilatero e i livelli inferiori sono trapezio, parallelogramma, rettangolo e quindi quadrato lungo un percorso di ereditarietà. Il trapezio è una classe speciale di quadrilatero, il parallelogramma è una classe speciale di trapezio, il rettangolo è una classe speciale di parallelogramma e il quadrato è una classe speciale di rettangolo.

L'ereditarietà può essere utilizzata anche per creare classi più complesse. Tuttavia, questo non è così diretto in CLIPS. Ad esempio, la classe base per la geometria è un POINT contenente un singolo slot point1. Una LINE può quindi essere definita aggiungendo point2 a POINT. Un TRIANGLE si definisce aggiungendo point3 a LINE, e così via per i QUADRILATERALS, PENTAGONS, ecc.

```
(defclass POINT (is-a USER)
(multislot point1))

(defclass LINE (is-a POINT)
(multislot point2))

(defclass TRIANGLE (is-a LINE)
(multislot point3))
```

Notare come ogni classe è una specializzazione della classe madre ereditando i punti della superclasse e quindi aggiungendo un nuovo punto.

Il paradigma opposto è l'**Ereditarietà per Generalizzazione** in cui classi più generali vengono costruite a partire da classi semplici. Ad esempio, una LINE è considerata composta da due punti. Un TRIANGLE è formato da tre linee. Un QUADRILATERAL è formato da quattro linee e così via. Questo sarebbe un buon paradigma per costruire un sistema di disegno object-oriented in cui nuovi oggetti potrebbero essere costruiti a partire da oggetti più semplici.

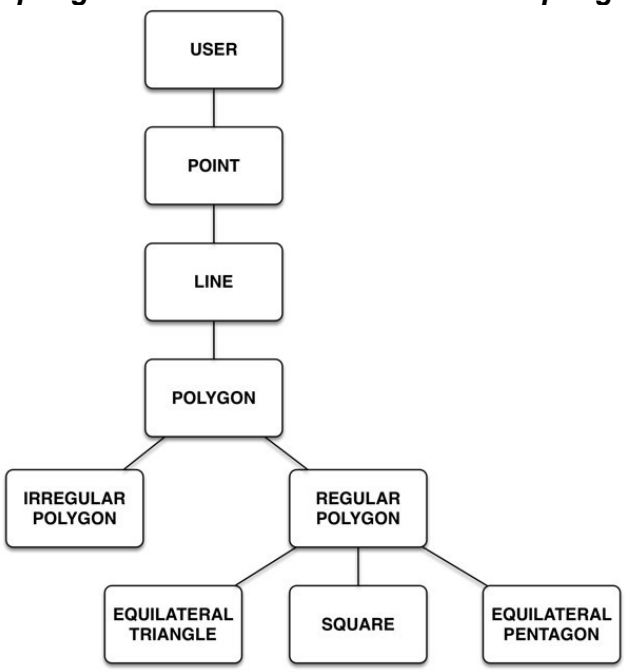
C'è una differenza sottile ma importante tra questo esempio di specializzazione e la generalizzazione. Nella specializzazione, le nuove classi vengono create aggiungendo slot specializzati che sono dello stesso tipo degli slot delle superclassi. Quindi, POINT, LINE e TRIANGLE hanno tutti slot di tipo punto.

Al contrario, la generalizzazione si sviluppa utilizzando nuovi tipi di slot definiti per ciascuna classe. La classe POINT ha uno slot di tipo punto. LINE ha due slot di tipo punto.

TRIANGLE ha tre slot di tipo linea. QUADRILATERAL ha quattro slot di tipo linea e così via. La generalizzazione è utile per la **sintesi**, che significa una costruzione. L'opposto della sintesi è l'**analisi**, che significa scomposizione o semplificazione. Il modello per l'analisi è la specializzazione.

La Figura 11.1 illustra uno schema di ereditarietà per i poligoni in cui le classi vengono costruite per ereditarietà. In questo caso il collegamento tra classi sarebbe “is-made-of”. Quindi una LINE è "is-madeof" POINT. Un POLYGON "is-made-of" LINE, e così via.

Fig. 11.1 Gerarchia di poligoni che mostra alcune classi di poligoni regolari



Collegamenti come “is-made-of” possono essere simulati in CLIPS mediante definizioni di slot appropriate anche se nella versione 6.3 sono supportati solo i collegamenti is-a. Come esempio di generalizzazione, creiamo una classe LINE come generalizzazione di una classe POINT. La classe POINT fornirà istanze che hanno una posizione. Per rendere realistico questo esempio, assumeremo un numero arbitrario di dimensioni definendo la posizione come (multiple). Pertanto, un punto unidimensionale avrà un valore nello slot di position, un punto bidimensionale avrà due valori e così via. La definizione della classe POINT è molto semplice.

```
CLIPS> (clear)
CLIPS>
(defclass POINT (is-a USER)
(multislot position
(propagation no-inherit)))
CLIPS>
```

La facet (no-inherit) viene utilizzata per impedire a una LINE di ereditare uno slot position. Invece, una LINE sarà definita da due punti chiamati slot point1 e slot point2. Questi due slot definiranno la linea ed è estraneo avere per ereditarietà uno slot position aggiuntivo.

La definizione della classe LINE è un po' più complessa. La ragione della maggiore complessità è che i dettagli dell'implementazione sono inclusi nella (defclass) poiché la Versione 6.3 supporta solo le relazioni is-a.

```
(defclass LINE (is-a POINT)
(slot point1
(default-dynamic
(make-instance (gensym*) of POINT))
(propagation no-inherit))
(slot point2
(default-dynamic
(make-instance (gensym*) of POINT))
(propagation no-inherit))
(message-handler find-point)
(message-handler print-points)
```

```
(message-handler find-distance)
(message-handler print-distance))
```

Da notare che i gestori di messaggi di LINE sono **forward-declared** per documentazione.

In questo momento ci si chiederà perché POINT e LINE non sono entrambi definiti come sottoclassi di USER poiché tutti i loro slot hanno facet (no-inherit). Poiché tutti gli slot di POINT, LINE e la classe TRIANGLE da definire in seguito hanno facet (no-inherit), tutte queste classi potrebbero essere definite come sottoclassi dirette di USER anziché definire LINE come sottoclasse di POINT e TRIANGLE come sottoclasse di LINE.

Tuttavia, il punto centrale di questo esempio è illustrare l'ereditarietà per Generalizzazione, che è un concetto logico che non è direttamente supportato dalla Versione 6.3. Pertanto, la definizione di LINE come sottoclasse di POINT e TRIANGLE come sottoclasse di LINE viene effettuata per documentazione del concetto logico di Ereditarietà per Generalizzazione. Certo, un commento potrebbe essere aggiunto da (defclass LINE (is-a USER)) e da (defclass TRIANGLE (is-a USER)) affermando che stiamo cercando di implementare l'ereditarietà per generalizzazione, ma vedere il codice a posto è una documentazione migliore. Se l'ereditarietà per generalizzazione sarà mai supportata direttamente da CLIPS, queste istruzioni (defclass) ne faciliteranno la conversione.

Il motivo per includere (make-instance (gensym*)) negli slot LINE è fornire l'ereditarietà dalla classe POINT. Con l'ereditarietà standard per specializzazione, è possibile solo uno slot position di LINE perché POINT ha un solo slot position. Non è possibile che sia lo slot point1 che lo slot point2 di LINE ereditino lo slot position di POINT. Il valore effettivo dello slot di ciascuna LINE sarà un valore gensym*. Ogni valore gensym* sarà il nome di istanza di un'istanza di punto. È poi possibile accedere alla posizione del punto tramite il valore gensym*. Pertanto, i valori gensym* agiscono come puntatori a istanze diverse.

Questa tecnica di **accesso indiretto** ai valori (gensym*) è analoga all'utilizzo di un puntatore per accedere a un valore in un linguaggio procedurale. Pertanto, i diversi slot di LINE possono ereditare indirettamente gli stessi slot di POINT. È comodo da usare (gensym*) perché non ci interessa quali siano i nomi dei puntatori di LINE, non più di quanto ci interessi quali siano gli indirizzi dei puntatori in un linguaggio procedurale.

Gli esempi seguenti mostrano come accedere ai punti per punti unidimensionali nelle posizioni 0 e 1.

```
CLIPS>
(definstances LINE_OBJECTS
(Line1 of LINE))
CLIPS> (reset)
CLIPS>
(send (send [Line1] get-point1)
put-position 0)
(0)
CLIPS>
(send (send [Line1] get-point2)
put-position 1)
(1)
CLIPS> (send [Line1] print)
[Line1] of LINE
(point1 [gen1])
(point2 [gen2])
CLIPS>
(send (send [Line1] get-point1)
get-position)
(0)
CLIPS>
(send (send [Line1] get-point2)
get-position)
(1)
CLIPS>
```

Capito come funziona l’accesso indiretto, definiamo alcuni gestori per evitare la difficoltà di inserire messaggi annidati (send), come negli ultimi due casi. Definiamo un gestore chiamato find-point per stampare un valore point specificato e un gestore chiamato print-points per stampare i valori di entrambi i punti LINE come segue. L'argomento di find-point sarà 1 per point1 o 2 per point2.

```
CLIPS>
(defmessage-handler LINE find-point (?point)
```

```
(send (send ?self
(sym-cat "get-point" ?point))
get-position))
CLIPS>
(defmessage-handler LINE print-points ()
(printout t "point1 "
(send ?self find-point 1)
crlf
"point2 "
(send ?self find-point 2)
crlf))
CLIPS> (send [Line1] find-point 1)
(0)
CLIPS> (send [Line1] find-point 2)
(1)
CLIPS>
```

Per l'uso reale, sarebbe meglio fornire il rilevamento degli errori in modo che siano consentiti solo 1 o 2.

Come si vede, il gestore funziona bene per i punti unidimensionali. Può essere testato per punti bidimensionali come segue. Assumeremo che il primo numero per ciascun punto sia il valore X e il secondo numero sia il valore Y. Cioè, point1 ha il valore X 1 2 di Y.

```
CLIPS>
(send (send [Line1] get-point1)
put-position 1 2)
(1 2)
CLIPS>
(send (send [Line1] get-point2)
put-position 4 6)
(4 6)
CLIPS> (send [Line1] print)
[Line1] of LINE
(point1 [gen1])
(point2 [gen2])
CLIPS>
(send (send [Line1] get-point1)
get-position)
(1 2)
CLIPS>
(send (send [Line1] get-point2)
get-position)
(4 6)
CLIPS>
```

Come previsto, il gestore funziona correttamente anche per i punti bidimensionali.

Ora che abbiamo le posizioni dei due punti, è facile trovare la distanza tra i punti, ovvero la lunghezza della linea. La distanza può essere determinata definendo un nuovo handler chiamato find-distance che utilizza il teorema di Pitagora per calcolare la distanza come radice quadrata della somma dei quadrati. Poiché non sono state fatte ipotesi riguardo al numero di dimensioni, la funzione (nth$) viene utilizzata per individuare ciascun valore multicampo fino al numero massimo di coordinate, memorizzato nella variabile ?len.

```
CLIPS>
(defmessage-handler LINE find-distance ()
(bind ?sum 0)
(bind ?index 1)
(bind ?len
(length$ (send ?self find-point 1)))
(bind ?Point1 (send ?self find-point 1))
(bind ?Point2 (send ?self find-point 2))
(while (<= ?index ?len)
(bind ?dif (- (nth$ ?index ?Point1)
(nth$ ?index ?Point2)))
(bind ?sum (+ ?sum (* ?dif ?dif)))
(bind ?index (+ ?index 1)))
(bind ?distance (sqrt ?sum)))
CLIPS>
(defmessage-handler LINE print-distance ()
(printout t "Distance = "
(send ?self find-distance)
crlf))
CLIPS> (send [Line1] print-distance)
Distance = 5.0
CLIPS>
```

I valori 1, 2 per point1 e 4, 6 per il point2 sono stati scelti per un facile controllo dell'handler poiché queste coordinate definiscono un triangolo 3–4–5. Come si vede, la distanza è 5.0, come previsto.

Le mappe del tesoro

Ora che le classi POINT e LINE sono state definite mediante ereditarietà generalizzata, perché fermarsi ora? Continuiamo con la prossima classe più semplice che può essere definita da una linea: il triangolo. Di seguito sono mostrate le tre defclass richieste.

```
CLIPS> (clear)
CLIPS>
(defclass POINT (is-a USER)
(multislot position
(propagation no-inherit)))
CLIPS>
(defclass LINE (is-a POINT)
(slot point1
(default-dynamic
(make-instance (gensym*) of POINT))
(propagation no-inherit))
(slot point2
(default-dynamic
(make-instance (gensym*) of POINT))
(propagation no-inherit)))
CLIPS>
(defclass TRIANGLE (is-a LINE)
(slot line1
(default-dynamic
(make-instance (gensym*) of LINE))
(propagation no-inherit))
(slot line2
(default-dynamic
(make-instance (gensym*) of LINE))
(propagation no-inherit))
(slot line3
(default-dynamic
(make-instance (gensym*) of LINE))
(propagation no-inherit)))
CLIPS>
```

Si noti che le facet (no-inherit) in TRIANGLE non sono tecnicamente necessarie poiché non è definita alcuna sottoclasse di TRIANGLE. Il motivo per cui sono state incluse le facet (no-inherit) è dovuto alla **defensive-programming**, che è analoga alla guida difensiva. Se viene aggiunta un'altra sottoclasse, da qualcuno, la (defclass) di TRIANGLE deve essere modificata per includere le facet (no-inherit). Questo non è uno stile corretto perché significa che il codice esistente di cui è stato eseguito il debug deve essere modificato. Per migliorare il codice esistente di cui è stato eseguito il debug, non lo si dovrebbe dover modificare. È meglio pianificare in anticipo i miglioramenti.

Successivamente definiremo un'istanza del triangolo e controlleremo le istanze generate. Tieni presente che i valori gen potrebbero essere diversi da quelli mostrati a meno che non si abbia appena avviato o riavviato CLIPS o non si si sia utilizzato (gensym*) o (gensym) dall'inizio.

```
CLIPS>
(definstances TRIANGLE_OBJECTS
(Triangle1 of TRIANGLE))
CLIPS> (reset)
CLIPS> (instances)
[Triangle1] of TRIANGLE
[gen3] of LINE
[gen4] of POINT
[gen5] of POINT
[gen6] of LINE
[gen7] of POINT
[gen8] of POINT
[gen9] of LINE
[gen10] of POINT
[gen11] of POINT
For a total of 10 instances.
CLIPS>
```

All'inizio ci si potrebbe sorprendere da tutti i valori gensym* creati. Tuttavia, sono tutti necessari. Innanzitutto, è stato creato gen3 per lo slot line1, che richiedeva gen4 e gen5 per gli slot point1 e point2 ad esso associati per ereditarietà. In secondo luogo, gen6 è

stato creato per lo slot line2, che richiedeva gen7 e gen8 per i suoi slot point1 e point2. Infine, è stato creato gen9 per lo slot line3, che richiedeva gen10 e gen11 per gli slot point1 e point2 ad esso associati. Gli slot per [Triangle1] e uno dei suoi valori del puntatore, [gen3], sono mostrati di seguito.

```
CLIPS> (send [Triangle1] print)
[Triangle1] of TRIANGLE
(line1 [gen3])
(line2 [gen6])
(line3 [gen9])
CLIPS> (send [gen3] print)
[gen3] of LINE
(point1 [gen4])
(point2 [gen5])
CLIPS>
```

Ora inseriamo le coordinate X, Y di [Triangle1] come segue.

```
CLIPS>
(send (send (send [Triangle1] get-line1)
get-point1)
put-position -1 0)
(-1 0)
CLIPS>
(send (send (send [Triangle1] get-line1)
get-point2)
put-position 0 2)
(0 2)
CLIPS>
(send (send (send [Triangle1] get-line2)
get-point1)
put-position 0 2)
2)
CLIPS>
(send (send (send [Triangle1] get-line2)
get-point2)
put-position 1 0)
0)
CLIPS>
(send (send (send [Triangle1] get-line3)
get-point1)
put-position 1 0)
(1 0)
CLIPS>
(send (send (send [Triangle1] get-line3)
get-point2)
put-position -1 0)
(-1 0)
CLIPS>
```

I valori memorizzati sono i seguenti.

```
CLIPS> (send [Triangle1] print)
[Triangle1] of TRIANGLE
(line1 [gen3])
(line2 [gen6])
(line3 [gen9])
CLIPS> (send [gen3] print)
[gen3] of LINE
(point1 [gen4])
(point2 [gen5])
CLIPS> (send [gen4] print)
[gen4] of POINT
(position -1 0)
CLIPS> (send [gen5] print)
[gen5] of POINT
(position 0 2)
CLIPS>
```

Come si vede, il puntatore line1 [gen1] punta a point1 e point2 con i loro puntatori [gen2] e [gen3]. Questi ultimi due puntatori puntano infine ai valori effettivi di (-1 0) e (0 2) che definiscono line1 di [Triangle1]. È analogo a Long John Silver che trova uno scrigno del tesoro con una mappa, [gen1], che conduce a un altro scrigno con due mappe, [gen2] e [gen3], che portano ai due tesori sepolti nelle posizioni specificate da [gen2] e [gen3].

I valori memorizzati per ciascuna riga di [Triangle1] possono essere recuperati da un singolo comando utilizzando messaggi nidificati come il seguente.

```
(send (send (send [Triangle1] get-line1)
get-point1)
get-position)
```

Sebbene questi comandi funzionino, non è molto divertente digitarli a meno che non si venga pagati a ore e non sia necessaria la praticità nella battitura. Come intuito dai gestori LINE che abbiamo definito nella sezione precedente, è possibile definire i gestori TRIANGLE come segue.

```
CLIPS>
(defmessage-handler TRIANGLE find-line-point
(?line ?point)
(send (send (send ?self
(sym-cat "get-line"
?line))
(sym-cat "get-point" ?point))
get-position))
CLIPS>
(defmessage-handler TRIANGLE print-line
(?line)
(printout t "point1 "
(send ?self find-line-point
?line 1)
crlf
"point2 "
(send ?self find-line-point
?line 2)
crlf))
CLIPS> (send [Triangle1] print-line 1)
point1 (-1 0)
point2 (0 2)
CLIPS>
```

Usare questi gestori è molto più conveniente che digitare i messaggi nidificati.

A questo punto, si potrebbe essere tentati di definire un gestore chiamato find-line che restituisca entrambi i valori in punti della linea specificata. Ricordiamo che find-line-point richiede la specificazione sia della linea che di uno dei due punti che definiscono la linea. Allora perché non inviare semplicemente due messaggi nello stesso gestore per restituire entrambi i valori in punti della linea specificata? Di seguito viene mostrato il gestore di find-line e ciò che restituisce per line1 di [Triangle1].

```
CLIPS>
(defmessage-handler TRIANGLE find-line (?line)
(send (send (send ?self
(sym-cat "get-line"
?line))
get-point1)
get-position)
(send (send (send ?self
(sym-cat "get-line"
?line))
get-point2)
get-position))
CLIPS> (send [Triangle1] find-line 1)
(0 2)
CLIPS>
```

Come si può vedere, il gestore restituisce solo l'ultimo valore del messaggio (0 2). Pertanto, il primo valore di (-1 0) non viene restituito da CLIPS. Questo è come il caso delle deffunction che restituiscono solo l'ultima azione. Un modo per aggirare questo problema e restituire entrambi i valori in punti della linea è mostrato di seguito.

```
CLIPS>
(defmessage-handler TRIANGLE find-line (?line)
(create$
(send
(send (send ?self
(sym-cat "get-line"
?line))
get-point1)
get-position)
(send
(send (send ?self
(sym-cat "get-line"
?line))
get-point2)
get-position)))
CLIPS> (send [Triangle1] find-line 1)
(-1 0 0 2)
CLIPS>
```

Si noti che la funzione (create$) è stata utilizzata per combinare entrambi i valori in punti in un unico valore multicampo (-1 0 0 2) che è stato poi restituito.

Altre funzionalità

Molte altre funzioni sono utili con i gestori. Ne seguono alcune.

undefmessage-handler: Elimina un gestore specificato.
list-defmessage-handlers: Elenca i gestori.
delete-instance: Opera sul gestore attivo.
message-handler-existp: Restituisce TRUE se il gestore esiste, altrimenti FALSE.

Le **funzioni di raggruppamento** raggruppano gli elementi COOL in una variabile multicampo.

get-defmessage-handler-list: Raggruppa i nomi delle classi, i nomi dei messaggi e i tipi (diretti o ereditati).
class-superclasses: Raggruppa tutti i nomi delle superclassi (diretti o ereditati).
class-subclasses: Raggruppa tutti i nomi di sottoclassi (diretti o ereditati).
class-slots: Raggruppa tutti i nomi degli slot (definiti esplicitamente o ereditati).
slot-existp: Restituisce TRUE se lo slot della classe esiste, altrimenti FALSE.
slot-facets: Raggruppa i valori del facet slot specificati di una classe.
slot-sources: Raggruppa i nomi degli slot delle classi che contribuiscono a uno slot nella classe specificata.

La funzione **preview-send** è utile nel debug poiché visualizza la sequenza di tutti i gestori che *potenzialmente* potrebbero essere coinvolti nell'elaborazione di un messaggio. Il motivo del termine *potenzialmente* è che i gestori shadow non verranno eseguiti se lo shadower non utilizza call-next-handler o override-next-handler.

Gli handler sono organizzati in una **precedenza del message-handler** che determina come vengono chiamati. Il processo di determinazione di quali gestori devono essere chiamati e in quale ordine è chiamato **message dispatch**. Ogni volta che un messaggio viene inviato a un oggetto, CLIPS organizza l'invio del messaggio per gli handler applicabili di quell'oggetto, che possono essere visualizzati dal comando (previewsend). Gli handler applicabili sono tutti i gestori in tutte le classi lungo il percorso di ereditarietà dell'oggetto che possono rispondere al tipo di messaggio. Seguono altre funzioni utili con il pattern matching degli oggetti in base alle regole.

object-pattern-match-delay: Ritarda il pattern matching delle regole fino a dopo la creazione, la modifica o l'eliminazione delle istanze.
modify-instance: Modifica l'istanza utilizzando l'override degli slot. Il pattern matching di oggetto viene posticipata a dopo le modifiche.
active-modify-instance: Modifica i valori dell'istanza concorrente all'oggetto object pattern matching con il messaggio *direct-modify*.
message-modify-instance: Modifica i valori dell'istanza. Posticipa l'oggetto pattern matching finché non vengono modificati tutti gli slot.
active-message-modify-instance: Modifica i valori dell'istanza concorrente con l'oggetto pattern matching utilizzando *message-modify*.

Capitolo 12

Domande e risposte

Il modo migliore per imparare è porsi domande; il modo migliore per scusarsi è rispondere a tutti.

In questo capitolo vedremo come creare un pattern matching sulle istanze. Un modo è con le regole. Inoltre, CLIPS ha una serie di funzioni di query per il matching delle istanze. Inoltre, è possibile utilizzare fatti di controllo e slot demoni per il pattern matching.

Lezioni oggetto

Una delle nuove funzionalità della versione 6.0 è la capacità delle regole di realizzare il pattern matching sugli oggetti. L'esempio seguente mostra come il valore dello slot sound dello viene abbinato [match] a una regola.

```
CLIPS> (clear)
CLIPS>
(defclass DUCK (is-a USER)
(multislot sound (default quack quack)))
CLIPS>
(make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS>
(make-instance [Dinky_Duck] of DUCK)
[Dinky_Duck]
CLIPS>
(defrule find-sound
?duck <- (object (is-a DUCK)
(sound $?find))
=>
(printout t "Duck "
(instance-name ?duck)
" says " ?find crlf)) CLIPS> (run)
Duck [Dinky_Duck] says (quack quack)
Duck [Dorky_Duck] says (quack quack)
CLIPS>
```

L'object-pattern oggetto elemento condizionale e seguito dalle classi e dagli slot con cui effettuare la corrispondenza [matching]. Dopo is-a e lo slot-name possono esserci espressioni di vincolo che coinvolgono ?, $?, &, e |.

Inoltre, è possibile specificare i nomi delle istanze per il pattern matching. L'esempio seguente mostra come viene soddisfatta [match] una sola istanza della classe DUCK utilizzando il **name constraint**, **name**, delle istanze. Notare che *name* è una parola riservata e non può essere utilizzata come nome di slot.

```
CLIPS>
(defrule find-sound
?duck <- (object (is-a DUCK)
(sound $?find)
(name [Dorky_Duck]))
=>
(printout t "Duck "
(instance-name ?duck)
" says " ?find crlf))
CLIPS> (run)
Duck [Dorky_Duck] says (quack quack)
CLIPS>
```

Oggetti nel database

Consideriamo il seguente tipo generale di problema. Dati alcuni casi, quanti soddisfano una condizione specifica? Ad esempio di seguito sono mostrate le defclasses e definstances di Joe's Home che mostrano le varie tipologie di sensori e gli elettrodomestici ad essi collegati. Si noti che viene definita una classe astratta DEVICE poiché sia SENSOR che APPLIANCE ereditano il tipo e la posizione degli slot comuni.

```
CLIPS> (clear)
CLIPS>
(defclass DEVICE (is-a USER)
(role abstract)
; Sensor type
```

```
(slot type (access initialize-only))
; Location
(slot loc (access initialize-only)))
CLIPS>
(defclass SENSOR (is-a DEVICE)
(role concrete)
(slot reading)
; Min reading
(slot min (access initialize-only))
; Max reading
(slot max (access initialize-only))
; SEN. APP.
(slot app (access initialize-only)))
CLIPS>
(defclass APPLIANCE (is-a DEVICE)
(role concrete)
; Depends on appliance
(slot setting)
; off or on
(slot status))
CLIPS>
(definstances ENVIRONMENT_OBJECTS
(T1 of SENSOR
(type temperature)
(loc kitchen)
(reading 110) ; Too hot
(min 20)
(max 100)
(app FR))
(T2 of SENSOR
(type temperature)
(loc bedroom)
(reading 10)  ; Too cold
(min 20)
(max 100)
(app FR))
(S1 of SENSOR
(type smoke)
(loc bedroom)
(reading nil) ; Bad sensor nil reading
(min 1000)
(max 5000)
(app SA))
(W1 of SENSOR (type water)
(loc basement)
(reading 0)   ; OK
(min 0)
(max 0)
(app WP))
(FR of APPLIANCE
(type furnace)
(loc basement)
(setting low) ; low or high
(status on))
(WP of APPLIANCE
(type water_pump)
(loc basement)
(setting fixed)
(status off))
(SA of APPLIANCE
(type smoke_alarm)
(loc basement)
(setting fixed)
(status off)))
CLIPS>
```

Supponiamo che vengano poste le seguenti domande o query. Quali sono tutti gli oggetti nel database? Come sono sistemati tutti gli oggetti? Quali sono le relazioni tra gli oggetti? Quali sono tutti i device? Cosa sono tutti i sensori? Quali sono tutti gli elettrodomestici? Quale sensore è collegato a quale apparecchio? Esistono sensori il cui tipo è la temperatura? Quali sensori di tipo temperatura hanno una lettura compresa tra il minimo e il massimo? Una domanda ancora più elementare è presente un *qualsiasi* sensore o meno.

Il **query system** di COOL è un insieme di sei funzioni utilizzabili per il pattern matching di un **instance-set** e per l'esecuzione di azioni. Un *instance-set* è un insieme di istanze, come le istanze di SENSOR. Le istanze in instance-set possono provenire da più classi

che non devono essere correlate. In altre parole, le classi non devono appartenere allo stesso percorso di ereditarietà.

Il seguente elenco dal *CLIPS Reference Manual*, riassume le**funzioni query** predefinite che possono essere utilizzate per l'accesso all'instance-set.

any-instancep: Determina se uno o più instance-sets soddisfano una query.

find-instance: Restituisce il primo instance-set che soddisfa una query.

find-all-instances: Raggruppa e restituisce tutti gli instance-set che soddisfano una query.

do-for-instance: Esegue un'azione per il primo instance-set che soddisfa una query.

do-for-all-instances: Esegue un'azione per ogni instance-set che soddisfa una query non appena vengono trovate.

delayed-do-for-all-instances: Raggruppa tutti le instance-set che soddisfano una query e poi esegue l'iterazione di un'azione su questo gruppo.

Ne prenderò uno qualsiasi

La funzione **any-instancep** è una funzione predicato che restituisce TRUE se esiste un'istanza che soddisfa il pattern matching e FALSE in caso contrario. Di seguito è riportato un esempio di questa funzione di query utilizzata con le classi e le istanze SENSOR e APPLIANCE. La funzione di query determina se è presente una *qualsiasi* istanza nella classe SENSOR.

```
CLIPS> (reset)
CLIPS> (instances)
[T1] of SENSOR
[T2] of SENSOR
[S1] of SENSOR
[W1] of SENSOR
[FR] of APPLIANCE
[WP] of APPLIANCE
[SA] of APPLIANCE For a total of 7 instances.
; Function returns TRUE because
; there is a SENSOR instance
CLIPS> (any-instancep ((?ins SENSOR)) TRUE)
TRUE
; Evaluation error—Bad!
; No DUCK class
CLIPS> (any-instancep ((?ins DUCK)) TRUE)
[PRNTUTIL1] Unable to find class DUCK.
CLIPS>
```

Il formato di base di una funzione di query prevede un **instance-set-template** per specificare le istanze e le relative classi, una **instance-set-query** come condizione booleana che le istanze devono soddisfare e **actions** per specificare le azioni da intraprendere. La funzione predicato **class-existp** restituisce TRUE se la classe esiste e FALSE altrimenti.

La combinazione del nome dell'istanza seguita da una o più **restrizioni di classe** è detta una **instance-set-member-template**. Le funzioni di query possono generalmente essere utilizzate come qualsiasi altra funzione in CLIPS.

Ci sono due passaggi coinvolti nel tentativo di soddisfare una query. Innanzitutto, CLIPS genera tutti i possibili instance-set che corrispondono [match] all'instance-set-template. In secondo luogo, la instance-setquery booleana viene applicata a tutte gli instance-set per vedere quali, se presenti, soddisfano la query. Gli instance-set vengono generati da una semplice **permutazione** dei membri in un template, in cui i membri più a destra vengono modificati per primi. Notare che una permutazione non è la stessa cosa di una **combinazione** perché l'ordine è importante in una permutazione ma non in una combinazione.

La funzione **find-all-instances** restituisce un valore multicampo di tutte le istanze che soddisfano la query o un valore multicampo vuoto per nessuna. La funzione di query **do-for-instance** è simile a find-instance tranne per il fatto che esegue una singola **azione distribuita** quando la query è soddisfatta. La funzione **do-for-all-instances** è simile alla funzione do-for-instance tranne per il fatto che esegue le sue azioni per ogni instance-set che soddisfa la query.

Decisioni progettuali

A differenza delle regole che vengono attivate solo quando i loro pattern sono soddisfatti, le deffunctions vengono esplicitamente chiamate e poi eseguite. Solo perché una regola è attivata non significa che verrà eseguita. Le deffunctions sono di natura completamente procedurale perché una volta chiamate per nome, il loro codice viene eseguito in modo procedurale, istruzione per istruzione. Inoltre, in una deffunction non viene utilizzato alcun pattern matching che coinvolga vincoli per decidere se le sue azioni devono essere eseguite. Invece, qualsiasi argomento che corrisponde [match] al numero previsto dall'elenco degli argomenti della deffunction soddisferà la deffunction e causerà l'esecuzione delle sue azioni.

L'idea di base delle deffunction come codice procedurale denominato è portata a un livello molto maggiore con **defgenerics** e i **defmethods** che ne descrivono l'implementazione. Una defgeneric è come una deffunction ma molto più potente perché può svolgere compiti diversi a seconda dei vincoli e dei tipi di argomenti. La capacità di una funzione generica di eseguire azioni diverse a seconda delle classi dei suoi argomenti è detto **overload** del nome della funzione.

Utilizzando correttamente l'overload degli operatori, è possibile scrivere codice più leggibile e riutilizzabile. Ad esempio, un defgeneric per la funzione “+” può essere definito con diversi defmethod. L'espressione,

```
(+ ?a ?b)
```

potrebbe aggiungere due numeri reali rappresentati da ?a e ?b, o due numeri complessi, o due matrici, o concatenare due stringhe, e così via, a seconda che sia definito un defmethod per le classi argomento. CLIPS fa questo riconoscendo prima il tipo degli argomenti e poi chiamando il defmethod appropriato definito per quei tipi. Un defmethod sovraccaricato [overloaded] separato per "+" verrebbe definito per ogni set di tipi di argomenti ad eccezione dei tipi di sistema predefiniti come i numeri reali. Una volta definito il defgeneric, è facile riutilizzarlo in altri programmi.

Qualsiasi **nome di funzione** definita dal sistema o esterna può essere sovraccaricata [overloaded] utilizzando una funzione generica. Si noti che una deffunction non può essere "overloaded". Un uso appropriato di una funzione **generica** consiste nell'overload di una nunzione con nome. Se l'overloading non è richiesto, è necessario definire una deffunction o una funzione esterna.

La sintassi delle defgeneric è molto semplice, consiste solo del nome legale del simbolo CLIPS e di un commento opzionale.

```
(defgeneric <name> [<comment>])
```

Come semplice esempio di funzioni generiche, si consideri il seguente tentativo in CLIPS di confrontare due stringhe utilizzando la funzione “>”.

```
CLIPS> (clear)
CLIPS> (> "duck2" "duck1")
[ARGACCES5] Function > expected argument #1 to be of type integer or float
CLIPS>
```

Non è possibile eseguire questo confronto con la funzione “>” perché prevede tipi NUMBER come argomenti.

Tuttavia, è facile definire un (defgeneric) che esegue l'overload di ">" per accettare i tipi STRING e i tipi NUMBER. Ad esempio, se gli argomenti di “>” sono di tipo STRING, il defgeneric eseguirà un confronto di stringhe, lettera per lettera iniziando da sinistra finché i codici ASCII non differiscono. Al contrario, se gli argomenti di “>” sono di tipo NUMBER, il sistema confronta il segno e la grandezza dei numeri. Il “>” user-defined per il tipo STRING è un **metodo esplicito**, mentre una funzione esterna definita dal sistema o dall'utente come “>” per il tipo NUMBER è un **metodo implicito** .

La tecnica dell'overloading del nome di una funzione in modo che il metodo che la implementa non sia noto fino al momento dell'esecuzione è un altro esempio di binding dinamico. Qualsiasi oggetto riferimento a un nome o indirizzo può essere associato in fase di esecuzione in CLIPS alle funzioni anche tramite il binding dinamico.

Alcuni linguaggi come Ada hanno un tipo di overloading più restrittivo in cui il nome della funzione deve essere noto in fase di compilazione anziché in fase di esecuzione. Il *dynamic binding* a run-time è la meno restrittiva poiché i metodi possono essere creati durante l'esecuzione dall'istruzione (build). Tuttavia, si deve fare attenzione nell'usare (build) poiché la creazione dinamica di costrutti è spesso difficile da debuggare. Inoltre, il codice risultante potrebbe essere difficile da verificare e convalidare poiché si dovrà interrompere l'esecuzione per esaminare il codice. Il binding dinamico è una caratteristica di un vero linguaggio di programmazione orientato agli oggetti.
Di seguito è riportato un esempio di defgeneric, “>”, per i tipi STRING e il relativo metodo.

```
; Header declaration. Actually unnecessary
CLIPS> (defgeneric >)
CLIPS>
(defmethod > ((?a STRING) (?b STRING))
(> (str-compare ?a ?b) 0))
; The overload ">" works correctly
; in all three cases.
CLIPS> (> "duck2" "duck1")
TRUE
CLIPS> (> "duck1" "duck1")
FALSE
CLIPS> (> "duck1" "duck2")
FALSE
CLIPS>
```

(defgeneric) agisce come una **dichiarazione di un header** per dichiarare il tipo della funzione overloaded. In questo caso non è effettivamente necessario utilizzare un defgeneric perché CLIPS deduce implicitamente il nome della funzione dal nome del defmethod, che è il primo simbolo che segue “defmethod”. L'header è una *dichiarazione forward* necessaria se i metodi (defgeneric) non sono ancora stati definiti, ma altro codice come defrules, defmessage-handlers e così via si riferiscono al nome (defgeneric).

Altre funzionalità

Rispetto alle deffunction, un metodo ha un **indice del metodo** opzionale. Se non si fornisce questo indice, CLIPS fornirà un numero di indice univoco tra i metodi per quella funzione generica, visualizzabile col comando **list-defmethods**. **method body** può essere stampato utilizzando il comando **ppdefmethod**. Un metodo può essere rimosso con una chiamata alla funzione **undefmethod**.

La classificazione [ranking] dei metodi determina la **precedenza del metodo** di una funzione generica. È la precedenza del metodo che determina l'ordine di elencazione dei metodi. I metodi con precedenza più alta vengono elencati prima di quelli con precedenza più bassa. Anche il metodo con la precedenza più alta verrà provato per primo da CLIPS.

Un **metodo shadow** è quello in cui un metodo deve essere chiamato da un altro. Il processo mediante il quale CLIPS sceglie il metodo con la massima precedenza è chiamato **generic dispatch**. Per ulteriori informazioni, consultare il *CLIPS Reference Manual*.

Informazioni di supporto

Domande e informazioni

L'URL della pagina Web CLIPS è http://www.clipsrules.net.

Le domande riguardanti CLIPS possono essere postate in uno dei numerosi forum online, incluso il
CLIPS Expert System Group, http://groups.google.com/group/CLIPSESG/, il SourceForge CLIPS Forums, http://sourceforge.net/forum/?group_id=215471 e su Stack Overflow, http://stackoverflow.com/questions/tagged/clips.

Richieste relative all'uso o all'installazione di CLIPS possono essere inviate tramite posta elettronica a support@clipsrules.net.

Codice Sorgente ed eseguibili di CLIPS

Gli eseguibili e il codice sorgente di CLIPS sono disponibili su http://sourceforge.net/projects/clipsrules/ files.

Documentazione

I manuali di riferimento CLIPS e la guida per l'utente sono disponibili in formato PDF (Portable Document Format) all'indirizzo http://www.clipsrules.net/?q=Documentation.

Expert Systems: Principles and Programming, 4a edizione, di Giarratano e Riley (ISBN 0-534-38447-1) viene fornito con un CD-ROM contenente i file eseguibili CLIPS 6.22 (DOS, Windows XP e Mac OS), la documentazione e il codice sorgente. La prima metà del libro è orientata alla teoria e la seconda metà tratta la programmazione basata su regole utilizzando CLIPS. È pubblicato da Course Technology.