di Raffaello De Masi

Il Debug

Come in ogni trattazione di linguaggi che si rispetti, la parte finale va riservata all'illustrazione delle tecniche di debug. Anche qui non ci esimeremo da questo obbligo, ma ci preme far rilevare alcune differenze che esistono (poteva non essere) tra il nostro e gli altri linguaggi.
Eseguire il debug di un programma è sempre una operazione complessa, per una serie di circostanze e di motivi peraltro ben noti a tutti. Dipende, come al solito, dalla «educazione» del programmatore, dal suo ordine, dal linguaggio che utilizza, dalle tecniche di redazione del programma da lui usate. In Prolog, per la sua stessa natura, il debug è sfrondato di un gran numero di passaggi; come già accadeva in LISP, la caratteristica particolare dei linguaggi di I.A., in cui per caso o a ragion veduta l'interattività è spinta al massimo grado, facilita grandemente il compito di «spulciare» il programma. A ciò si raggiunge una serie di facility, proprie del linguaggio, che rendono, per quanto possibile, il calvario del debug una operazione più sopportabile

Che scopi ha il debug

Se si è arrivati a programmare in prolog, certo si ha poco bisogno di rispondere a questa domanda; ciononostante lo facciamo lo stesso illudendoci che ci sia stato qualche lettore che abbia proprio scelto il nostro per cominciare a programmare. Ciò premesso, possiamo affermare senza tema di smentite che scrivere un programma immediatamente funzionante secondo i nostri desideri è più raro dell'araba fenice. Molto più facilmente il programma non va, o per semplici errori di output, non rispondenti alle nostre esigenze o ai nostri gusti, o perché il programma si «impunta» in questo o quel passaggio. D'altro canto anche programmatori professionisti a tempo pieno raramente possono affermare che le loro creature siano esenti da errori (bug-free) e lo dimostra anche l'uso di fior fiore di pacchetti, anche della più bell'acqua, che presto o tardi mostrano, nell'uso continuato, qualche piccola (o grande) falla c'è sempre e bisogna porci rimedio (non a caso uno degli socpi delle registration card, incluse nei package, è anche quello di raggiungere gli utenti quando il costruttore si accorge di un bug imprevisto; in questo ho dovuto purtroppo verificare che, nella maggior parte dei casi, i distributori italiani evitano di applicare, per quanto possibile, questa pur corretta prassi commerciale; l'esatto inverso l'ho verificato con una utility acquistata in America, che mi è stata aggiornata per ben due volte con un costo, anche solo di spese postali, da parte del produttore superiore al costo stesso del programma). La tecnica più bruta è quella di stampare il nostro bel listatone, e con pazienza mirandolesca, mettersi a cercare l'errore; o far ricorso a certe tecniche e tool, specifici del linguaggio adottato, che ci facilitano, anche enormemente, il compito.

Ma, come diceva Cesare, per battere il nemico occorre prima conoscerlo; allora vediamo innanzitutto qual è, poi potremo dire come sconfiggerlo.

Gli errori sintattici e logici

In linea generale, indipendentemente dal linguaggio adottato e dal calcolatore utilizzato, ci sono due tipi fondamentali d'errore; quelli sintattici e quelli logici. I primi sono comparabili, con un traslato, agli errori d'ortografia e, formalmente, sono definibili semplicemente come errori che avvengono quando il linguaggio non comprende una istruzione; il caso più semplice e diffuso è rappresentato da un errore di battitura in un comando, ma lo stesso può avvenire nella definizione di una variabile o di una procedura. Nella maggior parte dei casi il compilatore evidenzia nell'apposita finestra una serie di messaggi d'errore, e, comunque, la compilazione non avviene. Elenco degli errori alla mano, si ripercorre il listato e si eseguono le opportune correzioni; se, dopo questa operazione, la compilazione ha successo, almeno dal punto di vista sintattico si può avere una ragionevole certezza che non ci sia niente di errato nel programma; tecnicamente ci troviamo di fronte a un programma Prolog eseguibile.

Ma non è detto che un programma eseguibile (vale a dire scritto correttamente) sia capace di dare i risultati desiderati. Esiste il grosso scoglio della correttezza logica; tanto per capirci se di due variabili è stata eseguita la somma invece che il prodotto, non esiste debugger di questo mondo capace di rilevare l'errore; in altri termini la responsabilità (e la fatica) di rendere un programma corretto dal punto di vista logico ricade tutta sulle spalle del programmatore. In altri termini il programma sarà compilato, ma eseguirà quello che effettivamente gli abbiamo scritto di fare, invece di quello che era nelle nostre intenzioni.

Esistono comunque una serie di regolette che permettono di alleviare, almeno in parte, la frustante fatica del debug; si tratta di regole di comportamento, più che altro e possono essere così riassunte:

- costruire, la procedura principale attraverso una serie di procedure più semplici e più facili da testare; eseguire, appunto, un test dei predicati e delle clausole con valori semplici e di cui già si conosce il risultato (è ovvio che se una serie di operazioni dovrebbe dare il valore 100 ma non lo fa, a maggior ragione l'errore si dovrebbe ripetere per

calcoli da cui ci si aspetta risultati come 1234.56789;
- eseguire test di valori limite, o valori singolari; si tratta di una tecnica purtroppo un poco disattesa, ma sovente molto utile. Si tratta infatti di testare casi anomali per verificare che tutto funzioni alla perfezione; un esempio potrebbe essere il fattoriale, dove risulta talora utile verificare che, nel caso di fattoriale di 1 il risultato sia davvero 1;
- verificare l'affidabilità di funzioni multiargomento eseguendo sperimentazioni con diversi valori per ogni argomento; ad esempio capita spesso che il risultato sia valido usando un certo valore, ma che sia del tutto errato in altre condizioni. Sovente la tecnica di tenere fisso il valore di tutte le variabili e di variarne una sola con diversi valori dà risultati addirittura insperati;
- inserire dei breakpoint nel programma attravero cui eseguire un tracing dei valori delle variabili; ma di questo parliamo più a lungo di seguito.

Le tecniche di Tracing del programma

Quando, pur avendo eseguite tutte le verifiche precedentemente indicate il programma si rifiuta di dare risultati accettabili, occorre andare avanti nel debug con tecniche più specialistiche. Il Prolog è dotato di uno strumento di debug estremamente potente e facile da usare: la funzione di Tracing.

In linguaggi procedurali, come il Basic o il C, la funzione di Trace monitorizza i valori delle differenti variabili nel programma, evidenziando i valori da esse assunti sia nei passaggi intermedi che nella soluzione finale. Ma, in un linguaggio descrittivo come il Prolog, il valore e il contenuto delle variabili sono, come abbiamo più volte evidenziato, poco significativi. È invece molto più importante tenere il conto delle chiamate alle procedure, del ritorno da esse, dell'utilizzazione delle basi di dati e, in questo caso, delle variabili che ad esse sono collegate.

Il Tracing in Turbo Prolog è molto simile almeno nella forma a quello visto in altri linguaggi. Quando si chiede a Turbo Prolog di «tracciare» lo sviluppo di un programma egli fornisce una serie di notizie, nel Trace Window, circa l'andamento delle procedure. La funzione di Trace permette di passare, inoltre, in una fase di «single-step», di un passo (riga di programma) alla volta; il programma esegue una riga e si ferma, mostrando, ove necessario, le variazioni interne nel Trace Window; premendo F10 si passa al passo successivo. È possibile, altresì, stampare il contenuto della finestra con il comando Control-P o, più generalmente, con la combinazione CTRL-Ptr Scr. Comunque, per tracciare l'esecuzione di un programma, occorre notificare la direttiva al sistema. Ciò avviene, appunto, settando una delle due direttive del compilatore, Trace e ShortTrace, ma prima di parlare di queste due opzioni, due parole sulla ottimizzazione del compilatore.

Turbo Prolog maneggia il compilatore in maniera sostanzialmente diversa da quella di altri linguaggi. Non è qui il caso di parlare di questo tipo particolare di azione, dato anche che tutto quanto avviene è trasparente all'utente; facciamo solo cenno a una tecnica che dimostrerà quanto complesso sia il lavoro del linguaggio su quanto noi inseriremo attraverso un semplice listato.

La tecnica della ricursione non è qualcosa di innato in un linguaggio, come non lo sono le procedure, i metacomandi e così via. Ad esempio, nel caso di una procedura ricorsiva, il linguaggio non è in possesso di una mente sovrumana che gli permette di adottare la ricursione tal quale; ogni passaggio ricorsivo può essere anche scritto come un loop, che si esegue fino a che non si verifica una condizione particolare, ad esempio, nel caso del fattoriale, finché il contatore non raggiunge il valore di 1. Il Prolog non possiede tool efficienti per realizzare questo loop, cosa che, per esempio, può essere realizzata agevolmente in Basic. Ciononostante, summa dell'assurdo, la struttura ricorsiva scritta in Prolog, viene trasformata internamente, dal linguaggio, in un più comune loop, e così compilata. Si tratta di un processo di ottimizzazione del tutto trasparente all'utente, e fa intravedere cosa ci sia affettivamente in quel 20-30% in più che differenzia un codice compilato da un sorgente.

Ritorniamo alla differenza tra Trace e ShortTrace; premesso che si tratta di direttive che vanno inserite direttamente nel programma (ovviamente nel punto da cui si desidera l'esecuzione del Tracing, e, prevedibilmente, nella maggior parte dei casi, all'inizio), diremo, semplificando notevolmente il discorso, che le due tecniche si differenziano solo nel caso di procedure ricorsive (o più in generale, dove esiste ottimizzazione).

Qualunque sia la tecnica prescelta per la ottimizzazione, il Tracing può essere applicato a tutto il programma o solo a certe parti di esso; in questo caso il settaggio e la cancellazione della direttiva avvengono attraverso due comandi di funzione opposta [trace(off)] e [trace(on)].

In ogni caso, una volta eseguito il Tracing, il risultato viene visualizzato nella finestra relativa; i messaggi sono rappresentati essenzialmente dal contenuto degli argomenti, e da una serie di label, che evidenziano le azioni che av-

call	indica una chiamata ad una procedura
return	la procedura chiamata ha eseguito la sua funzione e il controllo è stato restituito alla routine principale
redo	la procedura chiamata è stata richiamata dopo che un tentativo precedente era fallito
fail	la procedura chiamata, con i relativi argomenti, non è stata coronata da successo

Figura a - I messaggi di Tracing, con i loro significati.

nome	Dbase	Dterm	Size	Domi-flowpattern
goal	no	yes	86	—
var_a	yes	no	255	symbol-symbol-i,i
var_b	yes	no	133	symbol-symbol-i,i
run	no	yes	250	41
segno	no	no	120	symbol-o
genere-x	no	no	255	symbol-i
genere_y	no	no	125	symbol-i
positivo	no	yes	700	symbol-symbol-i,i
negativo	no	no	400	symbol-symbol-i,i
regola	no	si	250	—
fatto_1	no	yes	1024	symbol-symbol-i,i,i,i
fatto_2	no	yes	444	symbol-symbol-i,i

Figura b - Un esempio di output prodotto da Diagnostic.

vengono durante l'esecuzione del programma stesso. Queste label sono riassunte nella figura a.

Seguendo passo passo le informazioni con la lista dei messaggi di Tracing, è possibile, osservando le chiamate, i parametri ad esse affidati e quelli restituiti, le inaspettate (o anche i risultati di alcune messe ad arte) combinazioni di FAIL-REDO, con una cerca sicurezza, diagnosticare errori logici e porvi rimedio; anche in questo caso sarà utile e interessante verificare risultati già testati prima di passare ad altri.

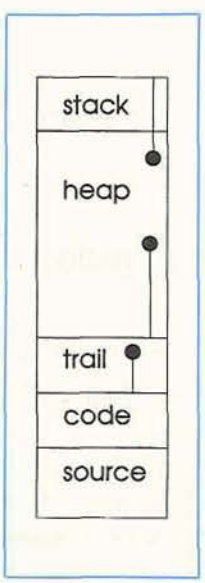
Per concludere, il debug è una pratica sempre estremamente tediosa e frustrante, sovente la scoperta di errori, magari banali, richiede un lungo ed estenuante periodo di tempo alla tastiera; purtroppo non è possibile farne a meno. Per questo, forse, i programmatori professionisti hanno cominciato a credere ad una mostruosa creatura originaria, il cosiddetto Bug Impossibile, capace di sfuggire alle ricerche più accurate e alle tecniche di caccia più raffinate. Ma se vi è capitato di spendere ore senza cavare un ragno dal buco, c'è una tecnica che non ha niente di nazionale, e che il più delle volte ha ragione dei più infami nemici; è qualcosa che ricordo funzionava anche al liceo, con i più infami, circonvoluti e intraducibili passi greci da tradurre; mettete tra voi e il bug un poco di tempo, preferibilmente una notte, senza pensare al problema; nella maggior parte dei casi quando riaffronterete il problema questo si scioglierà d'incanto; come procedimento non ha niente di scientifico, lo riconosco, ma funziona!

Le direttive del compilatore

Prima di concludere questa puntata (e, nello stesso momento, la trattazione del Prolog), ci preme parlare di qualcosa di cui abbiamo accennato qua e là in maniera piuttosto vaga, ma che il lettore avrà capito avere una importanza non trascurabile; le direttive del compilatore. C'è da dire che, in molti casi, non si accede a queste direttive; comunque appena si passa dallo stadio dilettantistico a quello più professionale, e la mole dei programmi cresce di conseguenza, molte delle direttive comprese nel compilatore divengono davvero utili, per non dire indispensabili.

Le direttive possono essere poste, all'interno del programma, in qualsiasi punto; per talune di esse questa «dislocabilità» è essenziale, ma la maggior parte coinvolge tutto il blocco del programma, per cui la soluzione migliore è quella di porle all'inizio del file, non foss'altro per pulizia di programmazione e facilità di reperimento, all'occorrenza.

Turbo Prolog include un'altra direttiva del compilatore che può essere utile nel debug, [diagnostic], che produce, al momento dell'esecuzione del programma, una lista del genere di quella mostrata in figura b). Capire cosa effettivamente accade e cosa significano i messaggi è relativamente semplice; la prima colonna contiene il nome di ogni predicato presente nel programma; i predicati sono elencati secondo il loro ordine di dichiarazione. La seconda colonna indica se tutte le presenze sono correlate con fatti anziché con regole (in altri termini se tutte le occorrenze sono realmente dei predicati di database). La terza colonna è forse quella più utile; essa indica se il predicato è deterministico (vale a dire, in termini più semplici, se produce una e una sola soluzione) o non deterministico (se è capace di produrre più di una soluzione attraverso il backtracking. La colonna successiva indica la grandezza della procedura dopo la compilazione, infine la stampa diagnostica mostra i domini che sono usati dal predicato, e la cosiddetta «rete di flusso». Basti sapere che i simboli [i] e [o] evidenziano rispettivamente se il parametro è o no rappresentato da un valore noto (per la verità non si capisce bene da che cosa provenga l'abbreviazione).



Figigura c ► La mappa di memoria in Turbo Prolog (da Dan Shafer, op. citata).

Due direttive del compilatore intervengono direttamente in runtime; la prima si riferisce in particolare alle situazioni di «warning» quando, nelle condizioni normali, il programma si ferma; in queste stesse condizioni di normalità la pressione di ESCAPE interrompe la compilazione, mentre quella di F10 permette di proseguire. La presenza della direttiva [nowarnings] permette di bypassare questa condizione, e si rivela utile in due condizioni particolari: quando una variabile compare una sola volta in una clausola, e quando una variabile, per qualche motivo, viene a superare i suoi limiti.

Un'altra direttiva piuttosto pratica, [nobreak] permette di escludere la possibilità, da parte dell'utente, di fermare il programma con la universale combinazione CTRL-C e CTRL-BREAK. Ovviamente è una procedura da usare con un minimo di cautela se si tien conto che se, durante l'uso, il flusso va a finire in un loop senza fine, l'unica possibilità di escape è quella di spegnere la macchina.

Un altro utile blocco di direttive è quello che consente di adottare, in maniera davvero efficiente ed estesa, la tecnica di programmazione modulare. Le modalità sono simili a quelle già viste in altri linguaggi, C in primis. La sintassi è data da un operatore, [include], che maneggia un solo argomento, il nome del file DOS da includere, appunto. A supervisione di questa tecnica ce n'è un'altra, dal nome di [project], che manipola un cosiddetto «project file», che è un file speciale che raccoglie gli [include] dei diversi file messi insieme. Ogni blocco, ovviamente, è rappresentato da un file singolarmente editabile e testabile; l'unica particolarità di questa direttiva ovviamente, è che deve comparire come primo comando nel file principale stesso.

E parliamo di un'altra direttiva, ancora, dal complesso nome [check_cmpio], che pare nasconda chissà quale messaggio interplanetario. Si tratta so-

lo, invece, dell'abbreviazione di «check compound I/O», che crea un messaggio d'avvertimento durante la compilazione se predicati sono usati in flussi incrociati (un flusso incrociato si verifica se uno o più parametri sono usati in input e uno o più altri sono usati in output dalla stessa procedura). Un'ultima procedura consente di manipolare predicati non deterministici, attraverso l'uso del comando [check_determ] che, in runtime, ferma l'esecuzione del programma appena viene incontrato un predicato non deterministico, appunto.

Occorre, prima di chiudere, parlare della manipolazione della memoria e della relativa mappa. Essa è visualizzata in figura c e mostra l'heap, lo stack e il trail che possono essere modificabili nella grandezza. La funzione dello stack è nota; del tutto trasparente all'utente, contiene riferimenti e indirizzamenti nelle operazioni di loop, nel lancio di procedure, nelle operazioni ricorsive. In default il linguaggio riserva 16 K per la regione che conterrà il codice. Può accadere, anche per le esigenze di spazio specifiche del compilatore stesso, che sovente questi limiti siano un poco stretti; occorre pertanto provvedere a un ampliamento di queste frontiere.

Per motivi non noti, la Borland utilizza una misura personale per lo spazio da allocare, rappresentata da «paragrafo». Esso consta di 16 byte e l'ampliamento dell'area di codice avviene secondo un semplice comando del tipo:

code = xxxxx

dove xxxx è il numero dei paragrafi da utilizzare (100, nel nostro caso vale 1600 byte). L'area di trail invece, è ben poco usata e, a quanto ci è dato di vedere, confinata alla realizzazione di programmi estremamente complessi; anche qui per mantenere chiare le cose, Borland ha scelto un sistema di misura personale, rappresentato dalla word, che è costituita, anch'essa da 16 byte; il formato della dichiarazione è sempre lo stesso:

trail = xxxx

Conclusioni

Conclusioni, stavolta, significa conclusioni generali. In questi mesi siamo partiti dai primi concetti, inusuali, di programmazione, per giungere, nocchieri in gran tempesta, per dirla alla Dante, a manipolare direttive e goal alla maniera di Magellano (o, meglio, per restare in ambiente, alla Bowman). Ci spiace solo dopo l'eccellente release 2 del Turbo Prolog, che la Borland pare abbia deciso di «mollare» il pacchetto, che peraltro era giunto ad una avanzata specializzazione, tanto da poter gareggiare, ad armi pari, con quotati ambienti di più alto lignaggio. Ci spiace di tutto questo e ci auguriamo che Borland ritorni sulle sue decisioni; vogliamo solo sperare che il tutto non dipenda da semplici considerazioni commerciali (credo d'altro canto che le previsioni di mercato per questo pacchetto non abbiano mai ipotizzato le code ai negozi); si tratta di prodotti che, d'altro canto, rappresentano una etichetta di prestigio, di carisma, che non può basare la sua controparte solo su un banale (anche se importante) numero di pacchetti venduti. D'altro canto la stessa Microsoft produce il suo Fortran e probabilmente ci perde, ma non per questo ne interrompe la produzione. Perciò speciamo (anche se la nostra è solo una vocina nella tempesta) che Borland non diventi, coll'andar del tempo «quella che fa il turbo Pascal». Non sembra anche a voi?