

INTERPRETE DI ESPRESSIONI ALGEBRICHE

esame di 'Fondamenti di Informatica II'

ENRICO PAPALINI

#1889524

corso di laurea in Ingegneria Informatica

anno di corso III

Università di Firenze

In questa relazione viene illustrato il progetto di un interprete di espressioni algebriche, cuore fondamentale di un qualsiasi interprete di un linguaggio evoluto.

La filosofia che ha guidato tutto il progetto è stata quella di limitarne la complessità in modo da presentare un programma che mantenesse dimensioni abbastanza circoscritte, senza però rinunciare ad una corretta impostazione del problema.

Si è quindi lavorato in modo rigoroso, progettando una grammatica di tipo LL(1) nella codifica EBNF che descrivesse le comuni espressioni algebriche, includendo le principali funzioni analitiche e la completa gestione delle variabili simboliche, a cui poter assegnare dei valori. Si è poi passati alla fase di programmazione vera e propria che si è risolta nel tradurre il linguaggio in un automa a stati finiti che ne riconoscesse le parole e in un interprete ricorsivo discendente con cui poter fare un'analisi top-down delle varie espressioni algebriche. In seguito si darà un completo resoconto di queste fasi e dei problemi inerenti la gestione delle tabelle dei simboli necessarie.

Per la stesura del programma eseguibile si è scelto il linguaggio ANSI C, in modo da garantire una completa portabilità dei sorgenti su differenti piattaforme hardware e la strutturazione del codice sorgente.

Inoltre, tale scelta ha permesso una completa modularità del progetto: anche se le dimensioni contenute non lo richiedevano espressamente, si è preferito dividere su più files il codice in modo da isolare gruppi di funzioni inerenti compiti specifici (analisi semantica e sintattica, gestione delle variabili, gestione dei simboli 'parole chiave', gestione dell'interfaccia con l'utente). Si è così ottenuto un gruppo di quattro files, che svolgono il compito richiesto di interprete di espressioni e che presentano, come collegamento con l'esterno soltanto poche funzioni, che possono essere integrate in altri progetti, qualora ci fosse bisogno di un interprete di funzioni (ad esempio, in un disegnatore di grafici di funzioni, in un pacchetto di calcolo numerico oppure in un interprete BASIC-like).

Per dare un esempio di tale capacità, sono stati sviluppati due drivers: uno, più spartano, che legge da un file testo un insieme di espressioni e restituisce per ognuna il suo valore; l'altro, un po' più evoluto, che si presenta come un interprete in linea di espressioni algebriche e che potrebbe essere fattivamente utilizzato in un progetto didattico, a partire dalle scuole medie inferiori, visto che mette a disposizione un semplice ambiente in cui definire degli oggetti (le variabili) e manipolarli secondo la loro sintassi (quella classica della matematica elementare).

Verranno adesso analizzate in dettaglio le scelte fatte per lo sviluppo del progetto, ripercorrendole in ordine logico, a partire dalla definizione della grammatica, fino ad arrivare agli ultimi ritocchi dei sorgenti, passando dai problemi di implementazione dello scanner, del parser, delle varie tabelle dei simboli e delle variabili.

I listati completi sono allegati nella Appendice A, oltre che nel dischetto MS-DOS compatibile a corredo. Nelle appendici seguenti sono riportate le grammatiche utilizzate per il parser e per lo scanner.

PROGETTO DELLA GRAMMATICA

Ogni volta che si ha che fare con un linguaggio bisogna impadronirsi della sua grammatica e padroneggiarla. Questo è vero in ogni caso, ed è ancora più vero nel caso dei linguaggi artificiali che si basano proprio sulla rigosità della grammatica che li definisce.

Ed è proprio questo rigore che ha provocato, in un primo momento del progetto dell'interprete di espressioni algebriche, una fase di studio abbastanza complessa nel corso della quale si è potuto apprezzare quanto una corretta stesura della grammatica del linguaggio permetta una traduzione veloce e quasi automatica del linguaggio stesso in un codice eseguibile dal calcolatore.

'Che grammatica scegliere?' è stato il primo dubbio. Ovviamente, le espressioni algebriche non presentavano argomento nuovo e la letteratura informatica pullula di modi per poterle descrivere e manipolare. E' altresì vero che nessun testo ha risposto esaurientemente alla domanda, in quanto la grammatica che si voleva, avrebbe dovuto avere una completa gestione non soltanto delle operazioni elementari, ma anche di qualche funzione analitica indispensabile (si pensi a quelle trigonometriche, oppure a quelle logaritmico-esponenziali), nonché delle operazioni logiche classiche. Inoltre, già a livello di grammatica sembrava opportuno tracciare le basi affinché l'assegnazione e la gestione delle variabili risultasse completamente integrata nella descrizione delle espressioni.

Ecco che, prendendo spunto da quello che avviene nel linguaggio ANSI C e da come le calcolatrici scientifiche programmabili gestiscono le funzioni si è pensato di introdurre già a livello di grammatica operatori, funzioni, assegnazione delle variabili, in modo che, assegnando a ciascuna categoria una produzione corrispondente, si ottenessero le usuali regole di precedenza fra operatori, la loro associatività, indifferentemente per ciascuna categoria.

Il risultato è che un albero sintattico prodotto alla grammatica progettata ha come radice la categoria <espressione> e come nodi figli tutte le altre categorie, incluso quelle logiche e di assegnazione, che in molti linguaggi di alto livello sono distinte dalle espressioni vere e proprie.

Scendendo nei dettagli, le grammatiche progettate (perché di due grammatiche in fondo si tratta, in quanto quella utilizzata dall' analizzatore lessicale è logicamente distinguibile da quella dell' analizzatore sintattico) sono:

- di tipo regolare a sinistra, quella che definisce la forma degli identificatori e delle costanti numeriche (e che è implementata nella funzione `leggi_token` nel modulo `EXPPARSE.c`, la funzione che esegue l'analisi lessicale). Tale forma è ispirata a quella degli identificatori e delle costanti dei linguaggi evoluti, come il C ed il Pascal e non se ne discosta di molto.
- di tipo LL(1) quella che definisce la sintassi delle espressioni (che è implementata nello stesso modulo della precedente). Questo tipo di grammatica è stata preferita alla più

Vediamo un ulteriore esempio:

$$A = B = 10 + 20$$

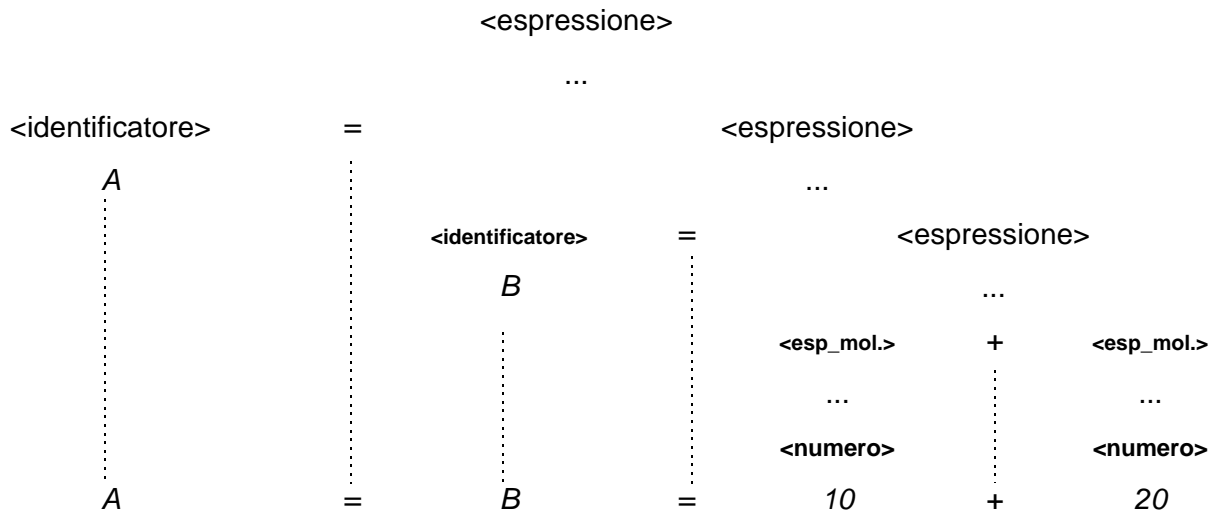


Tabella 2. Albero sintattico dell' espressione $A = B = 10+20$, nota: i tre punti '...' indicano derivazioni omesse per chiarezza.

Osserviamo che anche in questo caso tutto va come ci si immagina. Prima di tutto viene effettuata la somma, che viene assegnata come valore a B che è a sua volta assegnato come valore ad A.

Per concludere questa breve dissertazione sulle grammatiche utilizzate è bene far presente quale sia la precedenza degli operatori e la loro associatività (da sinistra a destra o viceversa). Nella seguente tabella le precedenze sono in ordine decrescente, a lato è riportata l'associatività:

OPERATORI	ASSOCIATIVITÀ	Produzione corrispondente
=	da destra a sinistra	<primario>
()	da sinistra a destra	<parentesi>
+ - / abs cos exp ... tan	da destra a sinistra	<unario>
^ rot	da sinistra a destra	<fattore>
* / mod div	da sinistra a destra	<esp_moltiplicativa>
+ -	da sinistra a destra	<esp_additiva>
== <> < > <= >=	da sinistra a destra	<esp_relazionale>
and or	da sinistra a destra	<espressione>

Tabella 3. Precedenze e associatività degli operatori del linguaggio

Osserviamo che gli operatori unari (+ - /) hanno precedenza maggiore dei corrispettivi binari e associatività invertite.

Inoltre si è scelto di trattare alla stregua di operatori unari le funzioni implementate, in modo tale che non siano necessarie parentesi in espressioni come

SIN A

anche se rimane valida anche la forma classica con le parentesi

SIN (A).

DEFINIZIONE DELLA STRUTTURA DELL' INTERPRETE

Progettate le due grammatiche, si è passati alla definizione della struttura dell' interprete, che per chiarezza è stata sintetizzata nello schema seguente.

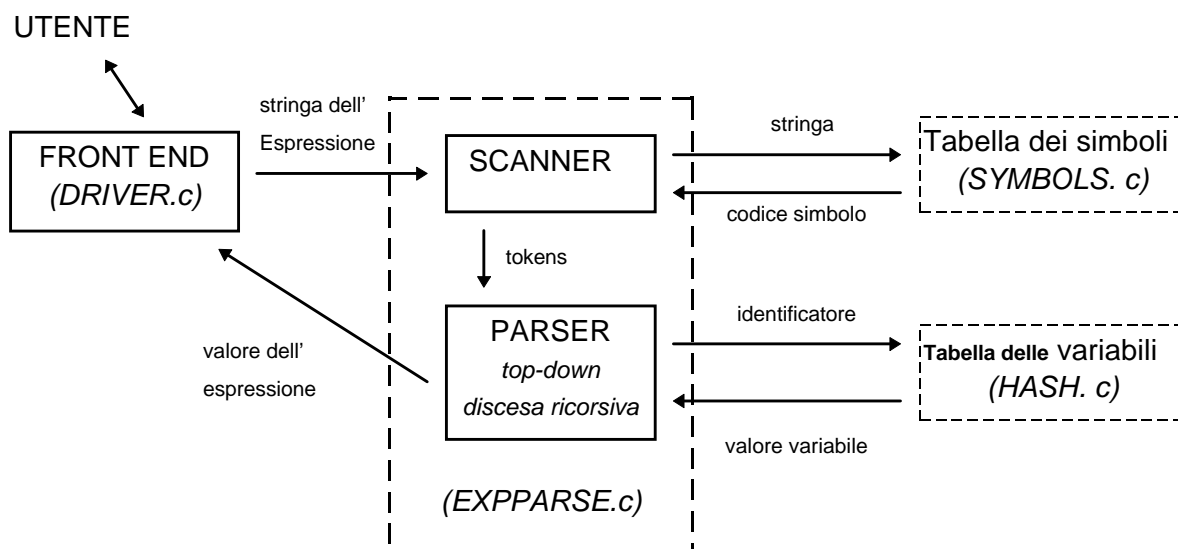


Figura 1. Struttura dell' interprete, fra parentesi in moduli che svolgono il compito indicato

L'utente scambia dati con il driver, unica parte dell' interprete che gli è accessibile. Questo è un' interfaccia con l'interprete vero e proprio che ad ogni chiamata riceve in ingresso una stringa contenente l'espressione immessa dall' utente e restituisce al driver il valore dell' espressione. L'espressione viene 'tokenizzata' dallo scanner, previo ordine del parser.

Lo scanner dialoga con la tabella dei simboli, al fine di capire se un identificatore sia un operatore o una variabile definita dall' utente. Il parser, oltre che con lo scanner, dialoga con la tabella delle variabili, al fine di memorizzare una variabile o reperirne il suo valore.

Vi sono anche delle eccezioni al precedente schema che fanno sì che il driver possa comunicare direttamente con tutti gli altri tre moduli. Queste eccezioni si verificano nei casi di errore (nell' input dato dall' utente, di overflow di calcolo oppure di fine memoria) durante i quali il controllo viene restituito direttamente al driver tramite un salto dal punto in cui è avvenuto l'errore al punto del driver in cui è stata definita la variabile *expbuffer* tramite *setjmp (expbuffer)*.

Nelle successive sezioni verranno spiegate in dettaglio le caratteristiche dei quattro blocchi principali (scanner, parser, tabella dei simboli e tabella delle variabili) e il loro funzionamento.

L' analizzatore semantico, o scanner, è contenuto nel file EXPPARSE.c ed è realizzato tramite la funzione *leggi_token()* che sfrutta due funzioni ausiliarie *is_delim()* (per riconoscere se il carattere attuale è un delimitatore) e *is_in()* (per riconoscere se il carattere attuale è in una certa stringa).

Inoltre utilizza le funzioni di interfaccia col modulo SYMBOLS.c (il gestore della tabella dei simboli), al fine di capire se l' identificatore o, più genericamente, il gruppo di simboli riconosciuti, è o non è un operatore valido (col termine operatore si indica uno qualsiasi di quelli riportati nella tabella delle precedenze a pagina 6, quindi anche una funzione).

In ingresso riceve dal parser un puntatore all'espressione corrente (la stringa passata dal driver nella sua ultima chiamata al parser) e in uscita restituisce il token attuale nella variabile strutturata *TOKEN*, composta dai seguenti tre campi:

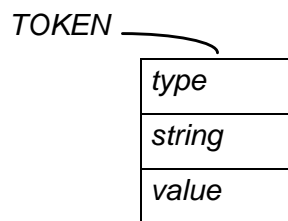


Figura 2. Struttura della variabile *TOKEN*

ove

- *type* contiene il tipo del token attuale:
 - NESSUNO se il token non è stato riconosciuto
 - NUMERO se il token è una costante numerica
 - IDENTIFICATORE se il token è una variabile
 - OPERATORE se il token è una parola chiave operatore.
- *string* contiene, nel caso di costante numerica, il valore di tale costante, nel caso di variabile il nome della variabile.
- *value* contiene invece il valore del token corrispondente all' operatore selezionato, restituito dalla chiamata alla tabella dei simboli.

La struttura dello scanner ricalca quella di un automa a stati finiti, ad esclusione della parte in cui si occupa di riconoscere se si sta trattando con un identificatore di variabile o con un operatore. Delegando tale compito al gestore della tabella dei simboli, che agisce su tutta la stringa già riconosciuta, lo scanner cessa di essere un automa a stati finiti data la presenza di memoria non ottenuta grazie ad uno stato dell' automa. Si è preferito utilizzare questo stratagemma, anziché implementare un' automa standard, per evitare che ci fosse da gestire un elevato numero di stati interni dello scanner.

Nel seguente grafico è rappresentato il funzionamento dello scanner. Si osservi che è stata utilizzata la freccia continua quando si passa da uno stato ad un altro tramite lettura del carattere successivo in input, mentre è stato abbreviato, tramite una freccia tratteggiata, l'impiego della tabella dei simboli (e quindi della memoria estranea all' automa a stati finiti).

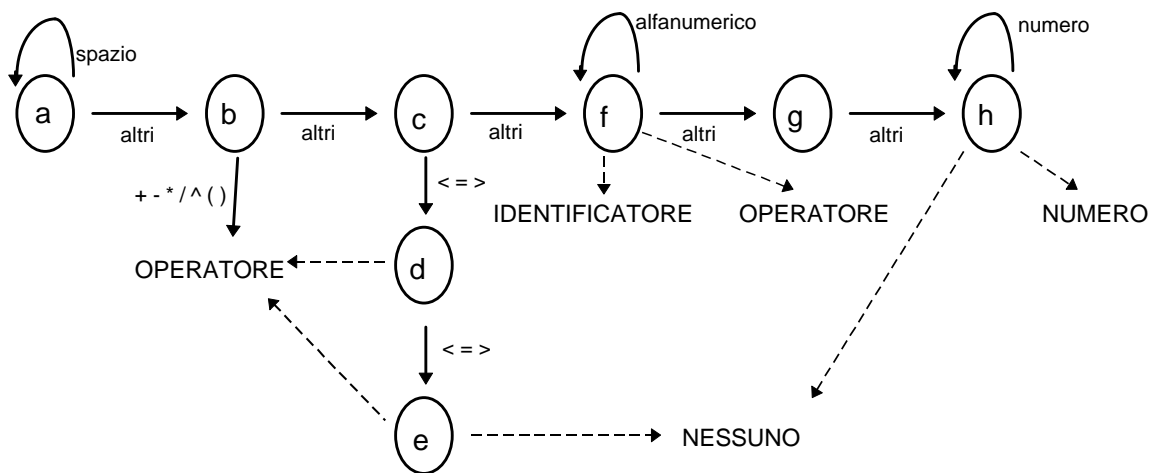


Figura 3. Rappresentazione di massima del funzionamento dello scanner

Lo schema proposto è ovviamente uno schema di massima. Per ulteriori dettagli operativi si rimanda all' analisi del codice sorgente e al paragrafo dedicato alla tabella dei simboli.

L' analizzatore sintattico, o parser, è stato realizzato tramite delle procedure, ognuna rappresentante una ben precisa produzione, come è prassi negli analizzatori sintattici a discesa ricorsiva.

Lo schema di traduzione da una grammatica EBNF a tale struttura procedurale è quello classico, reperibile in un qualsiasi libro di testo riguardanti il progetto di compilatori e interpreti (a riguardo si rimanda alle pagg. 105-127 di G.BRUNO, LINGUAGGI FORMALI E COMPILATORI, UTET Torino) ed è stato seguito in modo fedele. Il sorgente, allegato in appendice A, riporta prima di ogni funzione, la produzione a cui corrisponde.

Più interessante esaminare come è stato possibile realizzare il passaggio dei valori calcolati. Ogni funzione rappresentante una produzione ritorna un valore *double*. In questo modo, il processo di ricorsione automaticamente si occupa di passare alla funzione che ha richiamato quella corrente, il valore parziale calcolato.

Per il calcolo effettivo ci si è affidati a due funzioni, una che gestisce gli operatori ad un unico operando (*operatore_unario()*) ed un'altra che gestisce quelli a due operandi (*operatore_binario()*). A queste vengono passati l'operatore e gli operandi correnti e restituiscono, dopo la corretta elaborazione, il valore dell' operazione. Un'esemplificazione di come operi in realtà questo processo può essere vista direttamente sull' albero sintattico di pagina 4. Ogni volta che viene riconosciuto un operatore viene invocata una delle due procedure di calcolo, passandogli gli operandi. Tali operandi sono passati in forma di chiamate alle funzioni, corrispondenti alle produzioni a cui si riferiscono. Ad esempio, una volta riconosciuta la struttura

A + <esp_moltiplicativa>

la funzione di calcolo viene invocata come

***operatore_binario* (operatore '+', Valore di 'A', <esp_moltiplicativa>)**

sfruttando il fatto che il valore di A è già stato calcolato dallo schema top-down e che il valore dell' <esp_moltiplicativa> verrà calcolato, tramite il meccanismo ricorsivo, prima di effettuare la somma vera e propria.

Sfruttando questa tecnica si ottiene il pregevole risultato di mantenere associatività e precedenze in modo coerente a quanto progettato nella grammatica. Lo schema seguente mostra come tale processo avvenga per tutta l'espressione citata.

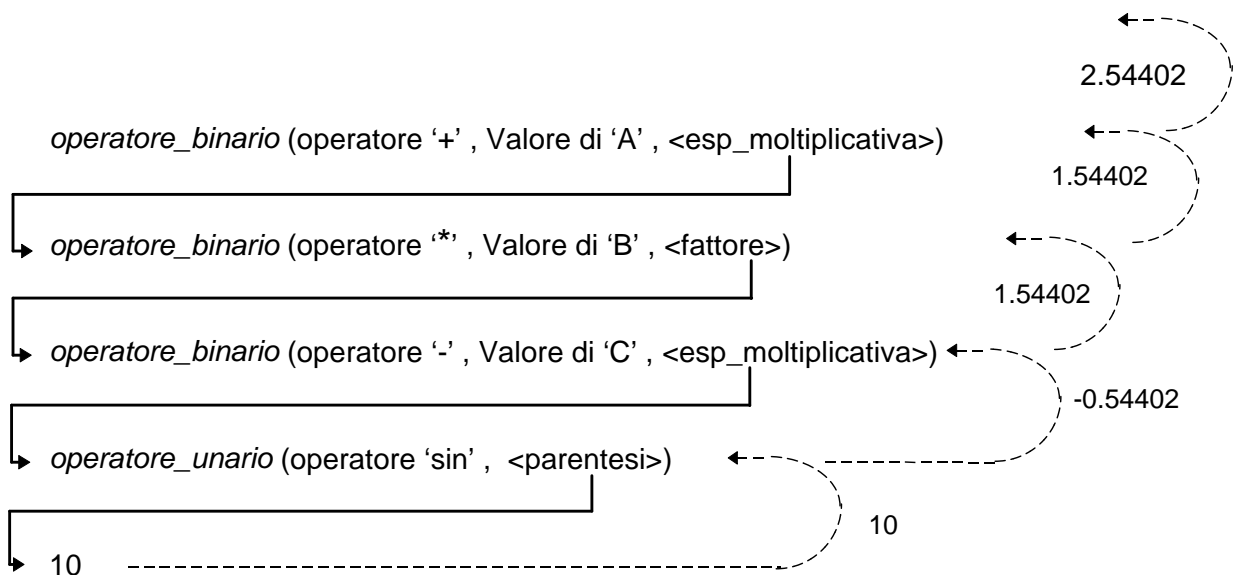


Figura 4. Schema di calcolo ricorsivo delle espressioni

Le frecce a tratto continuo mostrano le chiamate ricorsive, quelle a tratto spezzato i valori di ritorno di ciascun calcolo parziale. Ovviamente, prima dell' esecuzione sono stati assegnati i seguenti valori: $A=B=C=1$, come è mostrato nel frammento qui sotto.

```
->a=b=c=1
=> 1
->a+b*( c - sin 10)
=> 2.54402
```

Dall' esempio di esecuzione riportato appare evidente come lo scopo ultimo di ogni interpretazione sia quello di restituire il valore dell' espressione immessa. Si noti, ad esempio, come anche la prima triplice assegnazione riporti in uscita il valore assegnato alle tre variabili (in questo caso 1). Questo è proprio l'effetto voluto dell' aver parificato assegnazione, relazioni logiche al rango di semplici operatori.

Il parser comunica con lo scanner e con il gestore della tabella dei simboli. Allo scanner chiede, invocandolo, di restituire il token successivo, nel formato già esaminato. Il gestore della tabella delle Variabili è invocato all'interno della funzione che implementa la produzione <primario>. E' a quest'ultimo che è designato il compito di gestire la memorizzazione o il reperimento delle variabili inserite, indicando, nei casi in cui è richiesto il valore di una variabile non assegnata, l'opportuno errore.

Gli altri errori sintattici sono rilevati alla fine di questa produzione, in quanto, essendo l'ultima, se l'esecuzione salta i suoi cicli di controllo, significa che siamo in presenza di un errore sintattico e tale situazione viene segnalata. Il bilanciamento delle parentesi è controllato nella funzione che realizza la produzione <parentesi>, mentre gli errori 'di calcolo', come ad esempio la divisione per zero, sono gestiti all'interno delle procedure di calcolo.

Si è scelta questa via, in verità alquanto semplicistica, ma efficace, di gestire l'errore, perché si ottengono in questo modo dei buoni risultati, ignorando a volte di segnalare l'errore sintattico, ma proprio in quei casi in cui buona parte dell'espressione risulta corretta e quindi interpretabile.

Una gestione più accurata sarebbe stata possibile, ma in un progetto di queste dimensioni, si è reputato sufficiente scegliere questa via. In ogni caso, quando un errore viene rilevato, l'esecuzione viene passata alla posizione del modulo *driver.c* in cui è stata dichiarata la variabile *expbuffer* tramite la funzione *setjmp()*. Nei due esempi di driver realizzati si è scelto di gestire questo salto con due filosofie differenti. Nel primo caso (input da file testo) ogni errore provoca la terminazione prematura dell'esecuzione con la conseguente interruzione della lettura del file contenente le espressioni da elaborare. Nel secondo caso si è invece scelta una linea più permissiva. Trattandosi di un'interfaccia a linea di comando, l'uscita dopo ogni errore sarebbe stata noiosa e alla lunga frustrante: così, il programma semplicemente ignora l'espressione errata e tutto va avanti, dopo aver segnalato l'errore, come se l'espressione non corretta non fosse mai stata digitata.

LA TABELLA DEI SIMBOLI

Il modulo *Symbols.c* contiene la dichiarazione delle tabelle dei simboli e le funzioni atte alla loro gestione.

Tale tabella è stata divisa in tre parti (operatori ad un carattere, operatori a due caratteri e operatori-funzioni alfanumerici) in modo da poter essere più facilmente gestibile nel processo di riconoscimento semantico operato dallo scanner.

Il modulo si presenta con tre funzioni di interfaccia con l'esterno a cui passare una stringa e che restituiscono il valore del token, se tale stringa si è rivelata essere un operatore contenuto in tabella, in caso contrario è restituito il valore nullo.

Queste tre funzioni (*TrovaOperatore?* ()), ove ? sta per 1 o 2 o 3) richiamano la vera funzione di ricerca (*CercaSimbolo()*) che è basata su un algoritmo di ricerca binaria. È possibile utilizzare quest'ultimo in quanto le tabelle dei simboli sono dei vettori statici, in cui ogni elemento è una costante strutturata di due campi, uno che contiene la stringa con l'operatore, l'altro che contiene il valore numerico dell' operatore (quello che verrà registrato in *TOKEN.value* e utilizzato nel parser).

OPERATORI3

[0]	"abs"	T_ValoreAssoluto
[1]	"and"	T_E

[n-1]	"tan"	T_Tangente

Figura 5. Schema della tabella dei simboli OPERATORI3

A causa dell' algoritmo di ricerca utilizzato, le tre tabelle debbono essere tenute in ordine alfabetico, compito del tutto non oneroso se confrontato con i vantaggi in termini di velocità che questo comporta.

LA TABELLA DELLE VARIABILI

Il modulo *Hash.c* contiene le funzioni per gestire la tabella delle variabili. Già il nome del modulo fa intuire la tecnica su cui questa tabella è stata costruita. Il problema dell'intrinseco dinamismo delle variabili è stato risolto tramite una struttura basata su tecniche di hashing.

Dato che le variabili inseribili sono teoricamente un numero molto grande, si è utilizzata una tabella hash con liste di eccedenza realizzate tramite catene lineari.

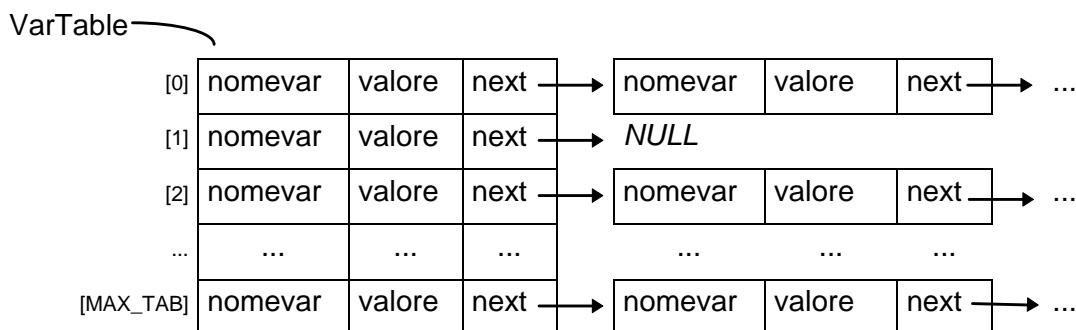


Figura 6. Schema della tabella delle variabili

Ogni elemento è strutturato in tre campi, di cui due informativi (l'identificatore e il suo valore) ed un'altro che punta al prossimo elemento della lista di eccedenza, se è stato allocato.

La funzione primaria per la gestione della tabella è *Valore_Variabile()* a cui viene passato l'identificatore corrente e tramite il settaggio di un opportuno switch si indica alla funzione l'operazione richiesta (se lo switch è nullo si tratta di leggere il valore dell'identificatore dalla tabella, se non è nullo si tratta di scrivere tale valore, che è passato come ulteriore parametro alla procedura). La funzione restituisce in ogni caso il valore della variabile corrente, a meno che si sia chiesto di leggere il valore di una variabile che non esiste: in questo caso si esce con un errore seguendo la prassi già descritta.

La funzione hash utilizzata sposta a sinistra di un bit l'indice hash già calcolato e lo sottopone a XOR con il carattere corrente dell'identificatore. Esauriti i caratteri dell'identificatore, il risultato viene normalizzato alla dimensione della tabella tramite un'operazione di modulo ed inizia la fase di ricerca vera e propria, in cui si scandisce la lista di eccedenza relativa all'indice hash calcolato fino a trovare la variabile. Ulteriori dettagli potranno essere reperiti nei commenti al codice sorgente.

Nello stesso modulo esiste un'ulteriore funzione, detta *Lista_Variabili()*, introdotta nella fase conclusiva del processo, con lo scopo di scandire tutta la tabella e le liste di eccedenza ad essa collegate e restituire nello stream video tutti gli identificatori delle variabili assegnate. Questa funzione è utile soprattutto in un uso in linea dell'interprete, in cui si può avere la necessità di una lista di variabili immesse.

UTILIZZO DELL' INTERPRETE

Veniamo adesso ad esaminare come il progetto descritto possa essere utilizzato nella pratica.

Come già accennato, sono stati messi a punto due driver (presentati nei modulo omonimi), per permettere all' utente di accedere alle funzionalità dell' interprete.

Il primo driver legge un file testo in cui sia presente in ogni riga una espressione algebrica e valuta tali espressioni ad una ad una, richiamando ogni volta l'analizzatore. L'interfaccia è volutamente un po' spartana per restare in linea con la concretezza del progetto, che poco concede agli abbellimenti e ai fronzoli (che continuano ad essere possibili, visto la modularità dell' interprete: basta riprogettare il solo driver per cambiare volto al programma, lasciando sotto di esso il motore base già descritto). Ogni espressione in input è preceduta da una freccia '->', mentre il risultato in output da una freccia di implicazione '=>'.

Vediamo qualche esempio di utilizzo. Mettiamo di aver digitato il seguente file di espressioni:

```
C:\PAPALINI\ENRICO>type prova1.txt
a=10
b=20
c=a*b/790
d=c*a
d+1
d+2*(tan c +12)
e=1
```

ed inseriamolo nel nostro interprete. Il risultato è il seguente:

```
C:\PAPALINI\ENRICO>expinter prova1.txt
-> a=10
=> 10
-> b=20
=> 20
-> c=a*b/790
=> 0.253165
-> d=c*a
=> 2.53165
```

```
-> d+1
=> 3.53165
-> d+2*(tan c +12)
=> 27.0491
-> e=1
=> 1
```

I calcoli sono stati ricontrollati e sono risultati essere corretti. Proviamo adesso a dare in pasto all' interprete un file con degli errori.

```
C:\PAPALINI\ENRICO>type prova2.txt
a = 10
c= (((3+4)*4) rot (4 - 2))*(cos a)
a * b
b= a = 1
c= a*b+2

C:\PAPALINI\ENRICO>expinter prova2.txt
-> a = 10
=> 10
-> c= (((3+4)*4) rot (4 - 2))*(cos a)
=> -4.43995
-> a * b
variabile non assegnata
```

Osserviamo che l'esecuzione si interrompe alla riga in cui è stato trovato l'errore (in questo caso si voleva utilizzare 'b' senza averne specificato il valore). Le righe successive a quella errata vengono ignorate (mentre i calcoli precedenti sono eseguiti correttamente, almeno secondo la riprova fatta con la calcolatrice).

Veniamo adesso a vedere alcuni esempi di utilizzo dell' interprete a linea di comando. Oltre alla gestione degli errori più 'tollerante', sono state introdotte nuove capacità, per rendere l'interfaccia un po' meno ostile. Digitando all' inizio della riga uno dei seguenti segni di interpunzione si accede alla funzione indicata di fianco:

carattere	funzione
?	Help in linea degli operatori e delle funzioni disponibili
:	Lista delle variabili già assegnate
;	ignora il seguito della linea, utile per inserire commenti
!	uscita dal programma

Tabella 4. Facilities dell' interprete

Vediamo un esempio di utilizzo dell' interprete con questa nuova interfaccia:

```
C:\PAPALINI\ENRICO>expinte2
->; eccomi, sono appena entrato
->; diamo un nome a qualche variabile
->base = 10
=> 10
->altezza = 5
=> 5
->area = base * altezza / 2
=> 25
->; chissà quali funzioni sono disponibili ?
->:
----- Interprete di espressioni algebriche -----
                        1995 Enrico Papalini
                   esame di Fondamenti di Informatica II
```

operatori disponibili:

% : Resto Divisione Intera
* : Moltiplicazione
+ : Somma
- : Sottrazione
/ : Divisione
< : Minore
= : Assegnazione
> : Maggiore
^ : Potenza
<= : Minore Uguale
<> : Diverso
== : Uguale
>= : Maggiore Uguale

funzioni disponibili:

abs : Valore Assoluto
and : E
cos : Coseno
div : Divisione Intera
exp : Esponenziale
fac : Fattoriale
int : Parte Intera
ln : Logaritmo Naturale
log : Logaritmo base 10
mod : Resto Divisione Intera
not : Negazione
or : O
rnd : A caso
rot : Radice
sgn : Segno
sin : Seno

```

sqr : Radice Quadrata
squ : Al Quadrato
tan : Tangente
->; adesso so che c'è anche una funzione per i
      numeri casuali (rnd) ma
->; non mi ricordo più che variabili avevo
      assegnato !
->:
      area
      altezza
      base
->; eccole !
-> cateto = 5
=> 5
->altro_cateto = area * 2 /cateto
=> 10
->bse *2
variabile non assegnata
->; ops... mi sono sbagliato, volevo scrivere
->base *2
=> 20
->; ora saluto, ciao!
->!
C:\PAPALINI\ENRICO>_

```

Ulteriori esempi di utilizzo verranno forniti nell' appendice D, tratti da problemi per le scuole medie inferiori.

CODICE SORGENTE

ANSI C

nelle pagine seguenti sono riportati i files :

EP1. h	headerfile del progetto "Interprete di espressioni algebriche"	I
SYMBOLS.c	gestore della tabella dei simboli	III
HASH.c	gestore della tabella delle variabili	V
EXPPARSE.c	scanner e parser delle espressioni algebriche	VII
DRIVER.c	esempio di interfaccia (input da file testo)	XVI
DRIVER2.c	esempio di interfaccia (input da tastiera)	XVII

Grammatica delle espressioni

EBNF

<espressione> ::= *<esp_relazionale>* {'and'|'or'} *<esp_relazionale>*
<esp_relazionale> ::= *<esp_additiva>* {'=='|'<'|'<'>'|'>='|'<=''} *<esp_additiva>*
<esp_additiva> ::= *<esp_moltiplicativa>* {'+'|'-''} *<esp_moltiplicativa>*
<esp_moltiplicativa> ::= *<fattore>* {'*'|'/'|'mod'|'div'} *<fattore>*
<fattore> ::= *<unario>* ['^'|'rot'] *<unario>*
<unario> ::= ['+'|'-'|'!'|'abs'|...|'tan'] *<parentesi>*
<parentesi> ::= '(' *<espressione>* ')' | *<primario>*
<primario> ::= *<numero>* | *<identificatore>* | *<identificatore>* '=' *<espressione>*

Grammatica dello scanner

EBNF

<numero> ::= *<numero>* | *<numero><cifra>* | *<numero>.<numero>* | *<numero>*
<cifra> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<ident.> ::= *<lettera>* | *<lettera><stringa>*
<stringa> ::= *<lettera>* | *<cifra>* | *<lettera><cifra>*
<lettera> ::= 'a' | ... | 'z' | 'A' | ... | 'Z'

RISOLUZIONE DI UN COMPITO DI MATEMATICA (CLASSE TERZA MEDIA)

tratto dal testo Livia Tonolini ALGEBRA Minerva Italica, pag 487

1. Calcola il valore della seguente espressione numerica:

$$\left(2 - 6 + \frac{1}{2}\right) - \left[\left(-\frac{3}{4} + \frac{1}{2}\right) - \left(\frac{3}{4} + \frac{1}{2}\right)\right] - \left[-\left(2 + \frac{4}{5}\right)\right]$$

2. Un mattone ha le dimensioni di 20 cm, 10 cm, 5 cm e il peso specifico di 1,8. Calcola il peso del mattone e calcola la pressione esercitata sul pavimento in ognuno dei tre modi in cui può essere appoggiato il mattone.

3. Determina il valore dell' espressione numerica:

$$\left\{ \left[\left(-\frac{1}{3}\right)^2 + \left(-\frac{1}{2}\right) \cdot \left(-\frac{1}{3}\right) \right]^2 \div \left(-\frac{1}{2} - \frac{1}{3}\right)^2 + \left(\frac{8}{25} - \frac{4}{5}\right)^2 \cdot \left(\frac{1}{4} - \frac{2}{3}\right)^3 \right\} \div \left(-\frac{17}{30}\right)$$

4. Un trapezio isoscele ha le basi che sono l'una i 5/14 dell' altra e l'altezza che misura 24 cm. Il trapezio è la base di un prisma retto, alto 20 cm, il cui volume è di 18240 cm³.

Calcola:

- l'area del trapezio di base,
- il perimetro del trapezio di base,
- l'area della superficie totale del prisma.

[sol. 1. 4/5, 2.1,8 Kg;9,18,36 Pa, 3.-1/6, 4.912 cm²; 136 cm; 4544 cm².]

Risolviamo i quattro questi con l'interprete in linea di comando;

```
-> ; primo esercizio
->(2-6+1/2)-((-3/4+1/2)-(3/4+1/2))-(-(2+4/5))
=> 0.8
-> ; secondo esercizio
->base = 20
=> 20
->altezza = 10
=> 10
->profondita = 5
=> 5
->ps = 1.8
=> 1.8
->volume = base * altezza* profondita
=> 1000
-> ; dato che ps = peso / volume allora...
->peso = ps * volume
=> 1800
-> ; la seconda domanda si risolve calcolando l'area delle tre basi
-> ; e dividendo il peso per ciascuna delle tre superfici
->areal = base * altezza
=> 200
->pressione1 = peso / areal
=> 9
->area2 = base * profondita
variabile non assegnata
->area2 = base * profondita
=> 100
->pressione2 = peso / area2
=> 18
->area3 = altezza * profondita
=> 50
->pressione3 = peso / area3
=> 36

-> ; terzo esercizio
->((squ (-1/3)+(-1/2)*(-1/3))^2 / squ (-1/2-1/3)+squ (8/25-4/5)*(1/4-
2/3)^3)/(-1
7/30)
=> -0.166667
-> ; ed ora risolviamo il quarto esercizio
```

```

->alt_trapezio = 24
=> 24
->alt_prisma = 20
=> 20
->vol_prisma = 18240
=> 18240
->area_trapezio = vol_prisma / alt_prisma
=> 912
->somma_basi = area_trapezio *2 / alt_trapezio
=> 76
->base_minore = somma_basi /19 * 5
=> 20
->base_maggiore = somma_basi - base_minore
=> 56
->cateto = (base_maggiore - base_minore )/2
=> 18
->lato_obliquo = sqr ( squ cateto + squ alt_trapezio)
=> 30
->perimetro = lato_obliquo * 2 + base_minore + base_maggiore
=> 136
->area_prisma = area_trapezio * 2 + lato_obliquo * alt_prisma *2
=> 3024
->; che stolto! Mi sono dimenticato delle aree laterali dovute alle basi
->; del trapezio
->area_prisma = area_prisma + base_minore * alt_prisma + base_maggiore *
    alt_prisma
=> 4544
->; il compito è risolto... e l'ottimo assicurato|
-> |

```

ELENCO DELLE FUNZIONI DISPONIBILI

abs x	valore assoluto di x	
x and y	x e y	restituisce 1 se entrambi gli operandi sono non nulli
cos x	coseno di x	
n div m	divisione intera fra n e m	n e m debbono essere interi, se non lo sono vengono troncati prima della divisione
exp x	esponenziale di x	
fact n	fattoriale di n	n deve essere un intero inferiore a 160
int x	parte intera di x	
ln x	logaritmo naturale di x	
log x	logaritmo in base 10 di x	
n mod m	resto divisione intera fra n e m	n e m debbono essere interi, se non lo sono vengono troncati prima della divisione
not x	negazione di x	se x è non nullo il risultato è nullo, e viceversa
x or y	x oppure y	restituisce 1 se almeno uno dei due operandi è non nullo
rnd n	numero casuale da 1 a n	n deve essere intero, sennò viene troncato
x rot y	$\sqrt[y]{x}$	
sgn x	segno di x	
sin x	seno di x	
sqr x	\sqrt{x}	
squ x	x^2	
tan x	tangente di x	
x + y ... x / y	le quattro operazioni elementari	
x^y	x^y	
x = y	assegna ad x il valore di y	x deve essere un identificatore di variabile
x == y ... x <> y	relazioni logiche fondamentali	se risultano vere, restituisce 1 sennò 0