Lezione 12

La lezione è centrata su un esercizio per impratichirsi sui seguenti argomenti:

- alberi binari rappresentati con puntatori
- alberi binari rappresentati con cursori

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Si vuole scrivere un programma `calcola.c` che elabora espressioni aritmetiche. Il programma acquisisce le espressioni da un file di testo, che contiene un'espressione in ogni riga, con una lunghezza non superiore a 256 caratteri. Le espressioni sono codificate come se fossero espressioni scritte in un linguaggio di programmazione, vale a dire nel seguente formato:

- espressione semplice: un numero reale;
- espressione composta: *operatore*`(`*operando1*`,`*operando2*`)`, dove
  1. *operatore* è il nome di un operatore aritmetico, cioè una stringa qualsiasi fra `somma`, `diff`, `prod`, `div`
  2. *operando1* e *operando2* sono due espressioni aritmetiche, semplici o composte.

Questo formato richiama da vicino la così detta *notazione prefissa* (o *polacca*), nella quale gli operatori precedono gli operandi. La differenza è che nella notazione polacca le parentesi non vengono usate, dato che sono ridondanti. Noi conserviamo le parentesi per maggiore leggibilità. Il file `input01.txt` riporta il seguente esempio di espressione composta:

`prod(somma(5,3),div(10,diff(4,2))`

il cui valore è

$$(5+3) * [10/(4-2)] = 8 * (10/2) = 8 * 5 = 40$$

Il programma deve:

1. calcolare il valore dell'espressione e stamparlo a video;
2. convertire l'espressione nella classica notazione infissa (con i simboli $+$, $-$, $*$ e / inseriti fra gli operandi) e stamparla a video;
3. determinare la *profondità* dell'albero, definita come il numero massimo di nodi lungo un cammino dalla radice alle foglie dell'albero.

Traccia della risoluzione

Prima di eseguire l'esercizio, si possono scorrere interamente i lucidi per acquisire il concetto di albero binario e le basi delle implementazioni di alberi binari con puntatori.

Quindi si può ragionare un po' sul modello del problema. Un'espressione aritmetica composta ha una struttura gerarchica e ricorsiva che si presta benissimo ad essere rappresentata con un albero. Le operazioni aritmetiche qui considerate sono tutte binarie, per cui l'albero può essere binario, anche senza introdurre la convenzione dei puntatori al primo figlio e al fratello successivo. Inoltre, la notazione prefissa consente di costruire l'albero identificando facilmente l'operatore, che corrisponde alla radice di un albero, e gli operandi, che corrispondono ai due sottoalberi sinistro (il primo operando) e destro (il secondo). Le parentesi e le virgole servono solo al lettore umano: il programma le ignora. Il programma dovrà costruire l'albero binario che rappresenta ogni espressione sfruttando il fatto che la notazione prefissa presenta alla lettura prima il nodo padre e poi i suoi due figli, facilitando la costruzione dell'albero stesso. Merita osservare che, quindi, la rappresentazione di partenza dell'espressione corrisponde esattamente a una visita in pre-ordine dell'albero corrispondente.

Il file di partenza, `calcola0.c`, contiene, al solito, le direttive per simulare il tipo `boolean`, la costante simbolica `ROW_LENGTH` per gestire le righe in lettura e scrittura, la funzione `InterpretaLineaComando` che recupera il nome del file dalla linea di comando e lo copia in un'apposita variabile. Contiene anche uno scheletro della funzione principale, `ElaboraEspressioni`, la quale apre il file e lo scorre con un ciclo, caricando ciascuna riga in una stringa in modo da poterla elaborare.

Prima fase (`calcola1.c`) La prima fase dell'esercizio richiede quindi di completare lo scheletro della funzione principale, come sempre in modalità *top-down*, cioè di:

1. leggere l'espressione corrente, che corrisponde a una riga del file (consiglio: `fgets` oppure `fscanf(fp,"%[^\n]\n",Espressione)`: la semplice specifica `"%s"` è valida solo se l'espressione non contiene spazi bianchi);
2. convertire l'espressione corrente da una stringa a un albero binario;
3. calcolare e stampare il valore dell'espressione rappresentata dall'albero;
4. stampare a video in notazione infissa l'espressione rappresentata dall'albero;
5. calcolare la profondità dell'albero;
6. distruggere l'albero.

Lo faremo assumendo di avere a disposizione una libreria per la gestione di alberi binari, della quale è disponibile il file *header* `albero.h`, che adotta un'implementazione a puntatori:

- un `operatore` è dichiarato come un carattere;
- un `operando` è dichiarato come un numero reale;
- un `albero` è dichiarato come un puntatore a un `nodo`;
- una `posizione` è dichiarata come un puntatore a un `nodo`;

- un `nodo` è dichiarato come una struttura contenente un operatore e un operando (questo è uno spreco, perché uno solo dei due campi sarà utilizzato, ma semplifica le cose) e tre posizioni: quella del padre e quelle delle radici del sottoalbero sinistro e destro.

Sono anche definite delle costanti simboliche per rappresentare operatori fittizii (`NO_OP`), operandi fittizii (`NO_VAL`), alberi vuoti (`NO_TREE`) e posizioni fittizie (`NO_NODE`). Il file `albero0.c` fornisce solo i corpi vuoti delle funzioni; quelle che devono restituire un valore ne restituiscono uno fittizio convenzionale.

Seconda fase (`albero.c`) Per poter completare la costruzione dell'albero, occorre definire le funzioni della libreria che gestisce gli alberi binari:

- `albero creaalbero ()` crea e restituisce un albero vuoto;
- `boolean alberovuoto (albero T)` determina se l'albero dato è vuoto o no;
- `void leggenodo (albero T, posizione p, Operatore *pop, Operando *pval)` restituisce l'informazione (operando e operatore) associata ad una data posizione di un dato albero;
- `void scrivenodo (albero T, posizione p, Operatore op, Operando val)` assegna un operando e un operatore a una data posizione di un dato albero;
- `posizione radice (albero T)` restituisce la posizione alla radice dell'albero;
- `posizione figliosinistro (albero T, posizione p)` restituisce la posizione alla radice del sottoalbero sinistro del nodo in una data posizione di un dato albero;
- `posizione figliodestro (albero T, posizione p)` restituisce la posizione alla radice del sottoalbero destro del nodo in una data posizione di un dato albero;
- `posizione padre (albero T, posizione p)` restituisce la posizione del nodo padre di quello in una data posizione di un dato albero;
- `albero costruiscealbero (Operatore op, Operando val, albero Ts, albero Td)` costruisce un albero inserendo nella sua radice una data informazione (operatore e operando) e appendendogli due alberi dati come sottoalbero sinistro e destro;
- `albero cancsottoalbero (albero T, posizione *pp)` distrugge l'intero sottoalbero appeso a una data posizione di un dato albero;
- `void distruggealbero (albero *pT)` distrugge un intero albero.

Le funzioni sono quasi tutte banali. È utile osservare che la distruzione di un intero albero è un caso particolare della cancellazione di un sottoalbero (basta applicarla alla radice), e che per cancellare un sottoalbero bisogna visitarlo rigorosamente in post-ordine, dato che bisogna deallocare prima i sottoalberi e poi il nodo radice; altrimenti, i sottoalberi non sarebbero più raggiungibili. Di conseguenza, tutte le operazioni fondamentali sono in tempo costante $\Theta(1)$ (sempre che le informazioni associate ai nodi abbiano dimensione costante), tranne la cancellazione e la distruzione, che richiedono tempo lineare nel numero n degli elementi dell'albero ($\Theta(n)$).

Terza fase (`calcola2.c`) Nella fase seguente dell'esercizio proseguiamo in modalità *top-down* convertendo l'espressione letta da file in un albero binario con la funzione `ConverteEspressione`. Siccome l'espressione è ricorsiva, conviene che anche la funzione che la manipola lo sia. Avremo quindi, come sempre, due casi:

- *caso base*: se l'espressione è semplice, cioè ridotta a un numero reale (per verificarlo, si può usare `sscanf`), basta costruire un albero con la radice contenente come operando il numero reale e un operatore fittizio e due sottoalberi vuoti;
- *caso ricorsivo*: se l'espressione è composta, occorre dividerla nelle sue tre stringhe componenti:
  1. operatore: la stringa iniziale sino alla prima parentesi aperta, esclusa;
  2. primo operando: la stringa compresa fra la prima parentesi aperta, esclusa, e la virgola che separa i due operandi (che non è in genere la prima virgola!) esclusa;
  3. secondo operando: la stringa finale che parte dalla virgola separatrice, esclusa, fino alla parentesi conclusiva, esclusa.

  Siccome è un'operazione sofisticata, se ne occuperà un'opportuna funzione `ScomponeEspressione`, da realizzare in seguito.

Una volta scomposta l'espressione, l'operatore va tradotto da una stringa (`somma`, `diff`, `prod` o `div`) nel carattere corrispondente (`'+'`, `'-'`, `'*'` o `'/'`), con un'opportuna funzione `IdentificaOperatore`. I due operandi, invece, vanno analizzati ricorsivamente per convertirli in alberi, ancora una volta grazie alla funzione `ConverteEspressione`. Otterremo così un operatore e due alberi, che possiamo fondere in un albero con la funzione di libreria `costruiscealbero`.

Quarta fase (`calcola3.c`) Procediamo quindi a realizzare le funzioni:

- `IdentificaOperatore`, che è una banale serie di confronti in cascata (`if ... else if ... else`) condotti con la funzione `strcmp`;
- `ScomponeEspressione`, che scorre la stringa data copiandone via via i caratteri in tre stringhe ausiliarie.

È possibile fare questo in modo abbastanza elegante, riconoscendo che i tre compiti sono molto simili fra loro: la prima stringa (operatore) termina alla prima parentesi aperta; la seconda (primo operando) termina con una virgola; la terza (secondo operando) termina con una parentesi chiusa. La virgola e la parentesi chiusa che indicano la fine dei due operandi non sono però le prime incontrate lungo la stringa: bisogna ignorare le virgole e le parentesi "annidate" in altre parentesi di livello inferiore. Per esempio, in `prod(somma(5,3),div(10,diff(4,2))` l'operatore è `prod`, mentre i due operandi non sono `somma(5` e 3, bensì `somma(5,3)` e `div(10,diff(3,2)`. Per risolvere questa difficoltà, è sufficiente trovare la prima occorrenza di un carattere separatore che non sia annidata all'interno di una coppia di parentesi. Dunque, si tratta di scorrere la stringa tenendo conto del numero di parentesi aperte che non siano ancora state chiuse: il separatore va accettato solo se questo numero è nullo. La procedura `TrovaCarattereNonAnnidato` fornisce la posizione del separatore. Questa posizione può essere poi usata per dividere l'espressione originale in due parti. La prima parte va copiata (con una funzione `strncpy`, dato che non stiamo copiando un'intera stringa, ma solo una sua porzione) nella stringa di destinazione. La seconda parte deve diventare la nuova stringa da

suddividere. Infatti, la ricerca del separatore successivo non può ripartire dall'inizio dell'espressione, ma deve partire dal carattere successivo al separatore precedente. Un modo un po' criptico, ma molto efficace di fare questo è sostituire `Espressione` con `&Espressione[i+1]`, cioè con il vettore (la stringa) che parte dalla posizione successiva al separatore precedente. Questo è reso possibile dall'equivalenza fra puntatori e vettori e dal fatto che la variabile `Espressione` non è in realtà il vettore statico allocato e riempito nella funzione `ElaboraEspressioni`, ma semplicemente un puntatore nello spazio di memoria locale della funzione `ConverteEspressione`, e quindi si può tranquillamente modificarlo senza alterare il vettore statico originale (che, in quanto vettore statico, non può stare a sinistra in un assegnamento).

Siccome le funzioni qui realizzate si applicano a stringhe con una lunghezza massima predefinita, la loro complessità è in $\Theta(1)$. Questo ci permette di valutare la complessità della conversione di un'espressione in un albero binario. Essa coincide con quella di una visita dell'albero stesso, cioè è proporzionale al numero n di elementi dell'albero ($\Theta(n)$), che è il numero totale di operatori e operandi che compaiono nell'espressione.

Quinta fase (`calcola4.c`) A questo punto, si può procedere a realizzare la funzione `CalcolaValore`, che valuta l'espressione rappresentata dall'albero. Ancora una volta, il modo più semplice è procedere ricorsivamente, avendo:

- un *caso base* corrispondente all'albero vuoto, per il quale si restituisce il valore fittizio `NO_VAL`;
- un secondo *caso base* nel quale la radice non contiene un operatore (cioè contiene l'operatore fittizio `NO_OP`), ma solo un operando, che va restituito;
- un *caso ricorsivo* nel quale occorre prima valutare le due espressioni figlie e poi applicare l'operatore ai loro valori.

Si noti che tutto questo equivale a una visita in post-ordine dell'albero stesso, dato che abbiamo prima valutato i nodi figli e poi ricavato da loro il valore del padre. Avendo realizzato tutte le operazioni su alberi binari (salvo la distruzione) in tempo costante, qualsiasi algoritmo sia riducibile a una visita dell'albero richiede tempo lineare nel numero dei suoi nodi, cioè $\Theta(n)$. Questo vale, ad esempio per la valutazione dell'espressione.

Sesta fase (`calcola5.c`) Passiamo quindi a realizzare la funzione `StampaEspressione` che stampa l'espressione rappresentata dall'albero nella classica notazione infissa, con l'operatore intermedio fra i due operandi. Per evitare ambiguità, dobbiamo aggiungere le parentesi tonde, che nella notazione prefissa di partenza erano presenti, anche se non necessarie: qui invece lo sono. Stamperemo quindi una parentesi aperta al principio dell'intera espressione e una chiusa al termine. Possiamo interpretare le parentesi come istruzioni di navigazione nell'albero: la parentesi aperta fa scendere al livello successivo, quella chiusa fa risalire al precedente. Naturalmente, l'implementazione sarà ancora ricorsiva, con due casi base (albero vuoto e albero ridotto alla sola radice) e un caso ricorsivo, nel quale l'elaborazione del figlio sinistro precede quella dell'operatore nella radice, che precede quella del figlio destro. Il caso di albero vuoto non comporta di fare nulla (quindi, nel codice apparentemente non compare). Il caso di albero ridotto alla sola radice comporta la stampa del valore conservato nella radice, che è un'espressione semplice. Si tratta quindi di una visita in in-ordine. Di conseguenza, la complessità è ancora $\Theta(n)$.

Settima fase (`calcola6.c`) L'ultima fase dell'esercizio realizza il calcolo della profondità dell'albero, anch'esso in modo ricorsivo:

- nel *caso base*, la profondità è per convenzione nulla;
- nel *caso ricorsivo*, la profondità è la maggiore fra le profondità dei due sottoalberi, incrementata di 1 per tener conto del livello aggiuntivo introdotto dalla radice.

Si tratta ancora di una visita, ma in effetti l'elaborazione del nodo radice è talmente banale (sommare 1 al massimo) che non ha molto senso cercare di classificare il tipo di visita. Comunque sia, la complessità è $\Theta(n)$ come in tutti i casi precedenti.

Nota finale: l'implementazione a vettori e indici A questo punto, si potrebbe procedere a creare una nuova libreria di gestione, basata sull'implementazione con vettori e indici interi. La differenze fondamentali sono:[1]

- l'albero è ospitato in un vettore allocato una volta per tutte dalla funzione `creaalbero`; la dimensione non varia durante l'esecuzione (se è insufficiente, si esce con un messaggio di errore) e il suo valore `TREE_SIZE+1` tiene conto degli elementi massimi ospitabili e della testa di una *lista libera*, che ospita gli elementi del vettore non impiegati nell'albero corrente;
- i puntatori sono sostituiti da indici numerici; per accedere ai nodi, bisogna quindi conoscere sia la posizione sia l'albero: il nodo in posizione `p` dell'albero `T` è banalmente `T[p]`
- ogni elemento del vettore contiene l'informazione associata a un nodo e gli indici numerici nel vettore stesso dei nodi radice dei due sottoalberi e del nodo padre;
- la radice è l'elemento del vettore corrispondente a un indice speciale (`RADICE`);
- gli elementi del vettore correntemente non impiegati nell'albero formano una lista libera, la cui testa corrisponde a un indice speciale;
- inserimenti e cancellazioni corrispondono a spostamenti fra albero e lista libera.

Pur essendo molto diversa dalla precedente, la nuova implementazione non richiede nessuna modifica al file `calcola.c`: potenza delle strutture dati astratte.

[1] Quanto segue è stato riadattato dalla lezione sulle liste senza procedere all'effettiva implementazione: si prega di segnalare eventuali errori o incongruenze.