Progetto di Linguaggi e Compilatori 1 – Parte 1
A.A 2015-16

GRUPPO 7

Graca Sanela 114201
Markovski Vlatko 112122
Michelli Luca 109532

Esercizio 1

Abbiamo fatto la funzione f che, dati un vettore e una matrice, calcola il prodotto tra il vettore e la matrice, e poi moltiplica il risultato ottenuto per lo stesso vettore trasposto.

Le matrici sono implementate mediante QuadTrees propri. Le matrici devono essere di dimensione 2^n x 2^n, dove n deve essere il valore *nexp*. Noi assumiamo che il programma venga lanciato passando in input una matrice il cui valore *nexp* sia effettivamente l'esponente del quadTree. Il vettore invece è implementato come una lista.

Non poniamo limiti per la dimensione del vettore. Se non è compatibile per la moltiplicazione con la matrice (cioè non è lungo quanto il lato della matrice), verrà segnalato un errore e il programma terminerà.

Per la risoluzione, abbiamo scomposto il problema in questo modo: la funzione *f* (dopo aver controllato le dimensioni), chiama una funzione ausiliaria *f1*, passandogli, inoltre, un intero i=0 che farà da contatore in *f1*.

La funzione *f1* farà la moltiplicazione tra il vettore e la colonna i-esima della matrice, fino a che i non sarà uguale alla lunghezza della matrice. Poi *f* farà un'ulteriore moltiplicazione tra il vettore così ottenuto e il vettore di input (in questo modo, mettendolo come secondo argomento, sarà considerato come trasposto).

La funzione *mul* è quella che effettivamente svolge la moltiplicazione tra due vettori; uno sarà sempre il vettore in input, mentre l'altro sarà una colonna della matrice (*f1* passerà ad ogni iterazione l' i-esima colonna).

Per poter moltiplicare le colonne della matrice con il vettore, queste vengono prima convertite in liste. La funzione *toList* si occupa di fare questo. Le colonne vengono invece estratte dalla funzione *getColonne*, che si occupa di estrarre l' i-esima colonna. Per far ciò, *getColonne* costruirà la lista della colonna prendendo prima l'elemento in posizione *i*, e poi si sposterà sempre di $i+2^n$ ($n=nexp$), fino alla fine della lista dei nodi del *quadTree*.

Oltre al file *Esercizio1.hs*, che contiene le funzioni appena descritte, abbiamo anche fatto il file *Test.txt*, che contiene le query di test.

Esercizio 2

Nella cartella Esercizio2 si trovano altre due cartelle che generano il parser e il lexer. Nella cartella CreazioneBnfc abbiamo generato la grammatica, e tramite bnfc abbiamo creato il parser, il lexer e il makefile con il demo per l'esecuzione. Non siamo riusciti a mantenere il valore dell'altezza dei nodi. Per questo motivo non siamo riusciti ad implementare la funzione Adjust.
Nella cartella CreazioneManuale abbiamo cercato di generare il parser e il lexer manualmente sperando di riuscire a fare quello che con il bnfc non ci era riuscito. Quindi abbiamo costruito le funzioni nella sintassi astratta, ma non siamo riusciti a utilizzarle per la grammatica che abbiamo creato.

Nonostante ciò, siamo riusciti a progettare la sintassi astratta polimorfa, creando però un solo parser (che comprende sia gli interi, che i numeri in virgola mobile, come anche tutti e due nello stesso albero) nella creazione con il bnfc; a differenza di quello creato manualmente, in cui abbiamo fatto due parser separati.

Punto b.
L(G) è il linguaggio creato da tutte le stringhe che se lette al contrario risultano essere diverse da se stesse.
Esempio. “a a a (a a a)” rovesciata risulterà: “)a a a(a a a”
“a a a” rovesciata risulterà: “a a a”
Come possiamo vedere dall'esempio, la prima stringa, che nella grammatica rappresenta un albero, risulta diversa da se stessa rovesciata. La seconda, invece, risulta la stessa. Da ciò possiamo concludere che qualsiasi stringa nella grammatica che rappresenti un albero senza i sottoalberi risulterà non appartenente al linguaggio, mentre tutte le altre (ossia quelle che hanno almeno un sottoalbero e quindi le parentesi) verranno accettate. Quindi l'unico insieme che risulta w=w(r) e non appartiene a L(G) è {a}*.

Il nostro linguaggio risulta essere: L(G)\ {a}*