

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

Gruppo 14

Marco Bucchiarone
Emanuele Tonetti
Francesco Zilli

Esercizio 1

La funzione Haskell `boundedMaximum(n [BST t])` data una lista di BST `t` trova, se esiste

$$\max_n t = \max(\forall x \in t \mid x < n).$$

La soluzione all'esercizio è implementata nel modulo `BoundMax` (`BoundMax.hs`) che viene importato nel `Main`. Nel `Main` la funzione viene chiamata fornendo gli argomenti inseriti a tastiera a runtime (usando le funzioni importate tramite `System.Environment`).

La `boundedMaximum` viene costruita sulla `foldl1`: `MaxNT` individua il massimo tra i minoranti nella lista costruita dalla funzione di visita del BST `getmin` che sfrutta la struttura dati considerata. Il modulo `BoundMax` è testabile nell'interprete di GHC tramite i test case forniti nel file `queryEs1.txt`, il quale comprende anche una rappresentazione grafica di alcuni degli alberi usati per una maggiore leggibilità.

Esercizio 2

Parte a

Dalla sintassi concreta per alberi pesati con numero arbitrario di figli fornita è stata costruita la sintassi astratta *polimorfa*. L'implementazione proposta è suddivisa tra i seguenti file, di cui si dà una breve descrizione:

Data.hs contiene la definizione dei tipi di dato per la sintassi astratta polimorfa e per i token

Lexer.x analizzatore lessico-grafico che produce token per dati di tipo `Int` e `Double`;

ParserI.y parser dedicato alla restituzione di alberi pesati di numeri soli
`Int`;

ParserD.y parser dedicato alla restituzione di alberi pesati di numeri `Double`;

Symm.hs

TestSymm.hs

In entrambi i parser è fornita, nella sezione di codice opzionale, la funzione `detWeight` che calcola il peso dei nodi individuali; poichè il peso viene definito ricorsivamente come la distanza massima di una foglia dal nodo considerato incrementato di 1 (ad ogni foglia viene assegnato peso pari a 0).

In entrambi i parser è implementata, come codice aggiuntivo, la funzione `detWeight` che determina per ogni singolo nodo il rispettivo peso (sommando 1 al peso massimo di eventuali figli), il quale sarà fornito nel risultato.

La funzione `isSymm` utilizza due sotto-funzioni:

- `isEq`: stabilisce l'equivalenza strutturale di due alberi;
- `trasp`: la trasposta di un albero in maniera ricorsiva (invertendo i suoi figli);

`isSymm` si basa sull'idea che una foglia è sempre simmetrica a se stessa, mentre un nodo lo è solamente se equivale al suo trasposto.

- `isEq`: stabilisce l'equivalenza strutturale di due alberi;
- `trasp`: la trasposta di un albero in maniera ricorsiva (invertendo i suoi figli);

`isSymm` si basa sull'idea che una foglia è sempre simmetrica a se stessa, mentre un nodo lo è solamente se equivale al suo trasposto. La funzione `isSymm` è combinata al parser per interi nel file sorgente `TestSymm`; qui viene stampato a video l'albero, il rispettivo albero trasposto ed il risultato della funzione `isSymm` che indicherà se l'albero è effettivamente simmetrico. All'eseguibile risultante dalla compilazione è possibile fornire in input il file `"examples.txt"` per effettuare svariati test case. (eseguire `make demo` per veloce riscontro)

- breve descrizione della grammatica $w=wR$ con esempi
- breve descrizione della grammatica di $L(G)$ con esempi, di cui palindromo solo "a", "b"
- descrizione della sottrazione $P - L(G)$ con dimostrazione
- descrizione dell'insieme risultante con esempi (" $[$ " e/o " $ab[ba$ ") (G)