Progetto Linguaggi Formali e Traduttori

Federico Paschetta

January 2022

1 DFA

Nella cartella "DFA" del file .zip inviato sono presenti i programmi .java degli esercizi riguardanti gli automi deterministici (da 1.2 a 1.10), con il disegno di ciascun automa. Per quanto riguarda l'esercizio 1.7 è presente anche una variante NFA dell'automa proposto, in cui l'automa ha già il nome da confrontare salvato e, dunque, ha soltanto tre stati.

2 Lexer

Nelle cartelle "Lexer 2.1" e "Lexer 2.2-2.3" sono presenti i Lexer e le classi dei relativi esercizi, con due soluzioni (una commentata) riguardanti il riconoscimento delle costanti numeriche.

3 Parser

Nelle cartelle "Parser 3.1" e "Parser 3.2" sono presenti i Parser e le classi dei relativi esercizi, insieme al file "Grammatica.txt" per ciascuna cartella, che ha produzioni, first, follow e insiemi guida di ciascuna grammatica.

4 Valutatore

Nella cartella "Valutatore 4.1" sono presenti il Valutatore e le classi necessarie alla sua esecuzione.

5 Translator

Nelle ultime quattro cartella sono presenti i traduttori e le classi dei relativi esercizi.

5.1 Translator 5.1

La base di tutte le diverse versioni è il Translator dell'esercizio 5.1. Per rendere più compatto ed efficiente il codice IJVM ho eliminato (messo sotto forma di commento) "emitLabel" di L0 e il suo conseguente "GOto L0" nella procedura "prog", poiché rappresentava soltanto un'etichetta vuota al fondo del programma. Allo stesso modo, inoltre, ho eliminato i "GOto" nel metodo "stat" nei case "assign", "print" e "read", poiché, anche in questo caso, non alterando le due chiamate il flusso di esecuzione, il programma continua "a cascata" come se ci fosse il "GOto". Per quanto riguarda i due principali problemi dell'applicazione della grammatica in linguaggio IJVM questo è ciò che ho applicato:

5.1.1 Assign a più variabili

Ho aggiunto tra le istruzioni concesse nel file "Instruction.java" "dup" e "pop", poiché il Translator, non conoscendo a priori a quante variabili assegnare il valore che segue "assign", ha sempre bisogno di averne in cima allo stack una copia, che viene duplicata ogni volta prima di un "istore", che, a sua volta, la consuma. Quando l'elenco di variabili è finito, si esegue un "pop" in modo da svuotare la pila della copia duplicata in eccesso.

5.1.2 Chiamate uguali, ma diversi comportamenti

Le variabili "idlist" e "exprlist" compaiono più volte nel corpo delle produzioni associate ad "assign", "print", "read" e agli operatori matematici, ma necessitano di un trattamento diverso per permettere di essere tradotte nella maniera corretta. Prendendo come esempio "idlist": "assign", infatti, come detto prima, ha bisogno di fare "dup", riconoscere l'ID, chiamare "idlistp" e fare "pop", mentre "read" ha bisogno di una "invokevirtual" e una "istore" per ogni variabile che compare in "idlist". Per questo motivo in "idlist" e in "idlistp" (e allo stesso modo in "exprlist" e "exprlistp") ho aggiunto un intero, definito al momento dell'invocazione del metodo nel corpo della produzione "stat", che determina quale case ha invocato la funzione e, dunque, in quale modo gestirla.

5.2 Translator 5.2

Per sviluppare il Translator 5.2 ho modificato il metodo relativo alla variabile "bexpr", rendendolo "bexprlist", in cui ho inserito i due operatori binari e l'operatore unario. La procedura "bexprlist" prende due attributi ereditati anziché uno e utilizza una label d'appoggio nelle diverse maniere per svolgere le operazioni booleane nella maniera adatta.

5.2.1 Translator 5.2.2

Ho provato ad implementare una modifica al Translator 5.2, secondo il principio dell'esercizio 5.3, quindi eliminando i "GOto" da "bexpr" e limitandoli al minor

numero possibile. Questa soluzione è più efficace per l'operatore "and", necessita un solo "GOto" per l'operatore "or", mentre non è efficace per "!".

5.3 Translator 5.3

Il Translator 5.3 differisce dal 5.1 per il controllo delle operazioni booleane, infatti, svolgendo l'operazione booleana opposta a quella richiesta nel testo, si può risparmiare una chiamata "GOto" ad ogni if o while. Nelle istruzioni IJVM, infatti qualsiasi istruzione if è seguita da una label a cui saltare qualora fosse vera. Ponendo la chiamata if opposta (ad esempio if-icmpge quando viene riconosciuto ¡) il programma salta alla label contenente il ramo "false" e prosegue direttamente con il codice relativo al ramo "true".