Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica (Classe L-31)

Compilazione di un linguaggio funzionale in Java

Laureando
Massimo Pavoni

Matricola 124377

Relatore
Prof. Luca Padovani

A.A. 2023/2024

Indice

Elenco dei codici

Elenco delle figure

Elenco delle tabelle

1. Introduzione

Lorem ipsum dolor sit amet, consectetur adipisci elit, sed do eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrum exercitationem ullamco laboriosam, nisi ut aliquid ex ea commodi consequatur. Duis aute irure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Nel Capitolo 1 illustreremo prima le motivazioni che ci hanno spinto a perseguire l'obiettivo descritto e quindi la struttura della tesi.

1.1 Motivazione

1.2 Obiettivi

1.3 Struttura della Tesi

2. Funx

Questo capitolo descrive brevemente i linguaggi funzionali e le scelte effettuate durante l'ideazione del linguaggio usato per il progetto: **Funx**.
Il nome nasce dall'unione dei due termini anglosassoni *functional* e *expression*; viene quindi pronunciato ['fʌnɪk's] in inglese, o comunque [fàn·èx] in italiano.

2.1 Linguaggi funzionali

Nonostante molti linguaggi non si possano confinare all'interno di una sola specie, parlando di linguaggi di programmazione si fa spesso riferimento a due grandi categorie: linguaggi imperativi e linguaggi dichiarativi.

I primi hanno caratteristiche direttamente legate al modello di calcolo di *John Von Neumann*, a sua volta non dissimile dalla macchina di *Alan Turing*. Questi linguaggi sono usati per scrivere codice che segue una precisa sequenza di istruzioni, la quale descrive più o meno esplicitamente i passi necessari per risolvere il problema affrontato.
Appartengono alla famiglia dei linguaggi di programmazione imperativi sia linguaggi procedurali come `Fortran`, `Cobol` e `Zig`, sia i linguaggi orientati agli oggetti, tra cui `Kotlin`, `C#` e `Ruby`.

Per lo scopo del progetto, tuttavia, è fondamentale analizzare i linguaggi dichiarativi: tali linguaggi sono generalmente di altissimo livello, e permettono allo sviluppatore di concentrarsi sull'obiettivo da raggiungere piuttosto che sui dettagli implementativi.
Fanno parte di questa categoria linguaggi di interrogazione come `SQL`, linguaggi di marcatura quali `HTML` e `XML`, linguaggi logici come `Prolog` e soprattutto i linguaggi funzionali: `Lisp`, `Clojure`, `Elixir`, `OCaml` e `Haskell` sono alcuni esempi.

Alla base di ogni linguaggio funzionale vi è il **lambda calcolo**: un sistema formale definito dal matematico *Alonzo Church* (supervisore di *Alan Turing* durante il dottorato), equivalente alla macchina di Turing, ma fondato sul concetto di funzione pura.

La grammatica del lambda calcolo verrà presentata poco più avanti (sezione 2.3), ma le regole che ne governano il funzionamento e il modo in cui queste vengano utilizzate per ridurre le espressioni ad una forma normale esulano dai fini di questo documento.
È invece rilevante elencare le principali qualità che un linguaggio funzionale comunemente matura grazie al lambda calcolo:

- **funzioni come entità di prima classe**: le funzioni possono essere passate come argomenti e restituite come risultato di altre funzioni;
- **immutabilità**: le variabili utilizzate sono immutabili;
- **purezza**: le funzioni sono libere da effetti collaterali (non modificano lo stato del programma) e restituiscono sempre lo stesso output per input identici;
- **ricorsione**: la ricorsione è il meccanismo più idiomatico per esprimere l'iterazione su una struttura dati.

2.2 ML, Haskell e Funx

Nonostante le funzioni pure tipiche di un linguaggio funzionale siano un concetto molto attraente dal punto di vista della correttezza della computazione, i vincoli così imposti possono risultare stringenti a tal punto da rendere difficile, se non impossibile, la scrittura di programmi che interagiscano con il mondo reale.

Per questo motivo, molti linguaggi funzionali permettono invece di utilizzare particolari funzioni impure o di effettuare almeno operazioni di input/output. Inoltre, molti linguaggi prevalentemente imperativi adottano ormai da tempo alcune caratteristiche tipiche dei linguaggi funzionali (e.g. `Rust`, il linguaggio più amato dagli sviluppatori secondo gli ultimi sondaggi di *Stack Overflow*, eredita molto dal linguaggio con cui era scritto il suo primo compilatore, `OCaml`, ed è dotato quindi di funzioni di prima classe, immutabilità di default, strutture dati algebriche, ecc.).

`ML` è un linguaggio funzionale sviluppato negli anni '70 presso l'Università di Edimburgo, costituente la base per moltissimi dei linguaggi sviluppati in seguito. `ML` permette effettivamente l'uso di funzioni impure, ma fra i suoi discendenti vi è `Haskell`, uno dei pochi linguaggi completamente puri.
`Haskell` si avvale di un pattern di programmazione chiamato *monadi* (vedi *Notions of computation and monads*, [Mog91]) per gestire le operazioni di input/output e altre operazioni impure, garantendo comunque la purezza delle funzioni.

Nell'ideare **Funx** l'ispirazione viene proprio da `Haskell`, ma è presente la possibilità di dichiarare un'unica funzione impura (il cosiddetto *main*) per permettere di visualizzare a schermo un risultato. Il linguaggio non è quindi allo stesso livello di purezza di `Haskell`, e naturalmente non supporta molte delle funzionalità più avanzate di quest'ultimo (come le *classi di tipi* e il *pattern matching*), ma ne mutua altre comunque interessanti, tra cui l'uso di alcuni operatori infissi e il *polimorfismo parametrico*.

2.3 Sintassi

La sintassi di **Funx** risulta molto simile a quella di `Haskell`, con poche differenze dovute a tre principali motivi:

- libera scelta di nomi e simboli per le parole chiave;
- necessità di successiva traduzione in `Java`;
- difficoltà e scarso valore all'interno del progetto dell'implementazione di un parser dipendente dall'indentazione.

A prescindere da ciò, il cuore del linguaggio è lo stesso di ogni altro linguaggio derivato dal lambda calcolo: la sua definizione si può agilmente comprendere visualizzando la grammatica del lambda calcolo e confrontandola con quella (leggermente semplificata) di **Funx**, facendo attenzione alle regole aggiuntive.

Espressione	E	$::=$	x	variabile
		$\mid$	$E_l\ E_r$	applicazione
		$\mid$	$\lambda x\ .\ E$	astrazione

Figura 2.1: Grammatica del lambda calcolo

Le tre regole presenti in Figura 2.1 indicano le tre componenti indispensabili del lambda calcolo:

- **variabile**: simbolo rappresentante un parametro;
- **applicazione**: applicazione di funzione ad un argomento (entrambi espressioni);
- **astrazione**: definizione di una funzione anonima, con un solo input x (variabile vincolata) e un solo output E (espressione, potenzialmente un'altra astrazione).

Modulo	M	::=	$nome \cdot L$	dichiarazione del modulo
Dichiarazione	D	::=	$?(schema\ di\ tipo) \cdot id = E$	dichiarazione di funzione
Espressione	E	::=	c	costante
		|	x	variabile
		|	$E_l\ E_r$	applicazione
		|	$\lambda x\ .\ E$	astrazione
		|	L	let
		|	$if\ E\ then\ E\ else\ E$	if
Let	L	::=	$let \cdot D\ D* \cdot in\ E$	let

Figura 2.2: Grammatica di Funx

È facile constatare la presenza delle ulteriori produzioni per la definizione del modulo corrente (informazione inclusa a prescindere dal fatto che il linguaggio ad ora non supporti l'importazione di moduli esterni che non siano la libreria standard) e di funzioni con nome: lo *schema di tipo* è un'informazione opzionale relativa al tipo della funzione e di cui si parlerà più approfonditamente nella sezione **??**.
Per quanto riguarda invece le espressioni, vengono introdotte tre nuove regole:

- **costante**: rappresenta un valore letterale, come un numero o una stringa;
- **let**: permette di avere dichiarazioni locali utilizzabili all'interno di un'espressione;
- **if**: non è altro che la più classica istruzione condizionale controllata da un'espressione booleana.

2.4 Zucchero sintattico

Con lo scopo di rendere il codice più leggibile, conciso e semplice, **Funx** introduce alcuni zuccheri sintattici (del tutto simili a quelli di `Haskell`). Oltre all'indispensabile per non dover fare parsing dell'indentazione e a quelli più comuni per l'arrichimento del lambda calcolo, sono riportati in Tabella 2.1 anche tutti gli operatori simbolici supportati al momento (assieme alla notazione per indicarne l'associatività e precedenza).

Zucchero	**Sostituzione**
`if b then e1 else e2 fi`	`if b then e1 else e2`

`let`
`f1 = e1`
`f2 = e2`
`in e3`
	`let f1 = e1 · f2 = e2 in e3`

`f3 = e3`
`with`
`f1 = e1`
`f2 = e2`
`out`
	`f3 = let f1 = e1 · f2 = e2 in e3`

`\x -> e`	`λx . e`
`\x y -> e`	`λx . λy . e`
`f x y = e`	`f = λx . λy . e`
`e1 . e2` `infixr 9`	`compose e1 e2`
`e1 / e2` `infixl 7`	`divide e1 e2`
`e1 % e2` `infixl 7`	`modulo e1 e2`
`e1 * e2` `infixl 7`	`multiply e1 e2`
`e1 + e2` `infixl 6`	`add e1 e2`
`e1 - e2` `infixl 6`	`subtract e1 e2`
`e1 > e2` `infix 4`	`greaterThan e1 e2`
`e1 >= e2` `infix 4`	`greaterThanEquals e1 e2`
`e1 < e2` `infix 4`	`lessThan e1 e2`
`e1 <= e2` `infix 4`	`lessThanEquals e1 e2`
`e1 == e2` `infix 4`	`equalsEquals e1 e2`
`e1 != e2` `infix 4`	`notEquals e1 e2`
`!!e` `prefix 4`	`not e`
`e1 && e2` `infixr 3`	`if e1 then e2 else False`
`e1 || e2` `infixr 2`	`if e1 then True else e2`
`e1 $ e2` `infixr 0`	`apply e1 e2`

Tabella 2.1: Zuccheri sintattici

Come già accennato, il Capitolo 5 illustrerà come l'albero sintattico astratto (AST) di un programma viene ottenuto, annotato e tradotto in Java; la sezione **??** in particolare esporrà il motivo della traduzione degli operatori booleani binari in if.

Un esempio di programma e corrispondente AST sono presentati nel Codice 2.1 e Figura 2.3; seppur superflua, l'indentazione è inclusa per maggiore chiarezza, e le annotazioni di tipo sono state omesse poiché approfondite nel Capitolo 3.

```
1 main = equalsEquals 499751156776108032 . multiply (tetration 3 3) . tetration 2 $ 4
2
3 tetration a n = if n == 0 then 1 else power a $ tetration a (n - 1) fi
4    with
5        power b e = if e == 0 then 1 else b * power b (e - 1) fi
6    out
```

Codice 2.1: Esempio di programma

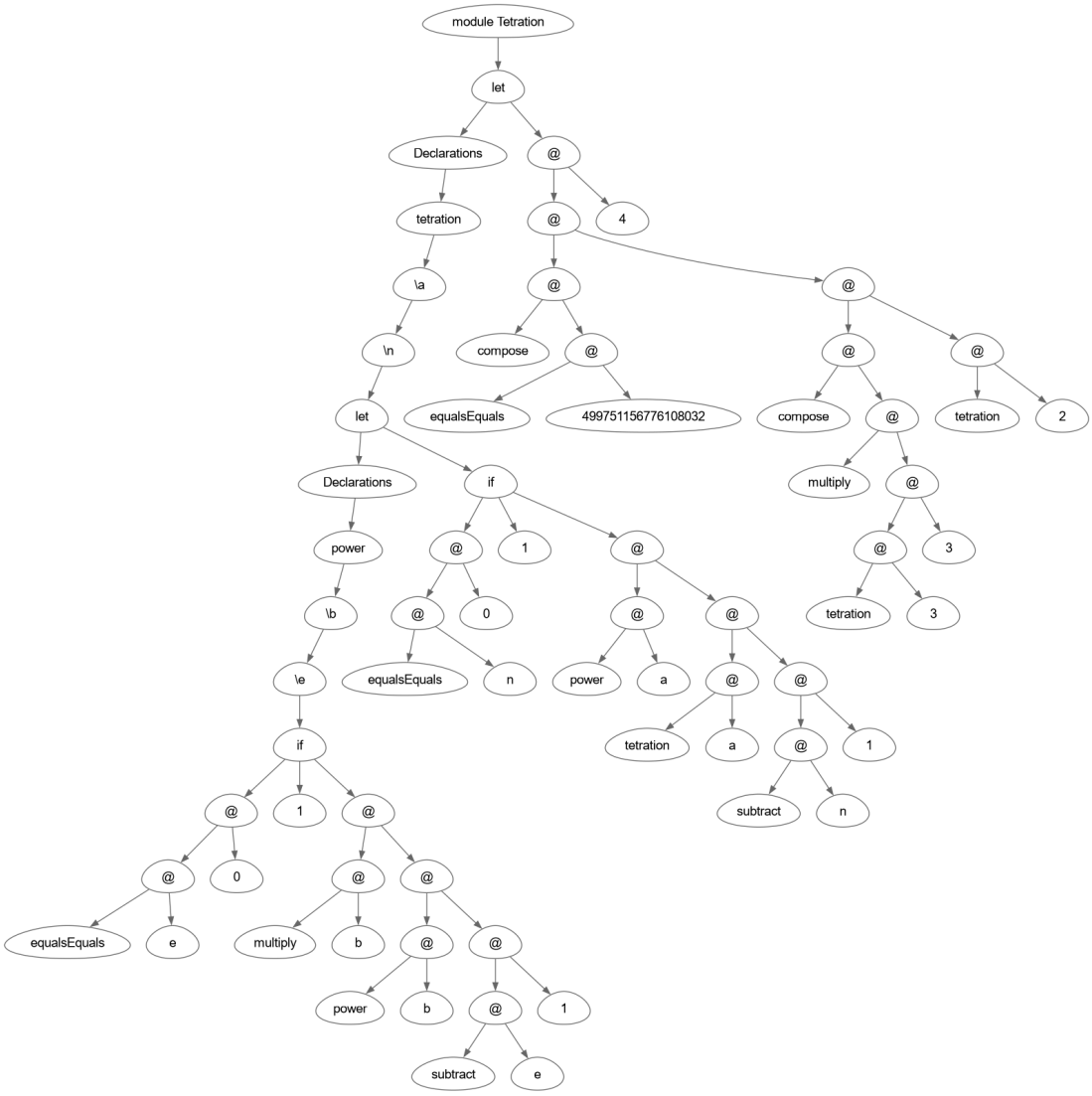

Figura 2.3: Esempio di AST

3. Inferenza di tipo

Lorem ipsum dolor sit amet, consectetur adipisci elit, sed do eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrum exercitationem ullamco laboriosam, nisi ut aliquid ex ea commodi consequatur. Duis aute irure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

In questo capitolo andremo a discutere ...

3.1 Sezione Esempio

Quello in figura 3.1 (esempio di riferimento a figura) ...

Figura 3.1: Esempio di figura

Esempio elenco puntato ...

- item 1
- item 2
- item 3

3.2 Section2

Esempio di citazione da bib [Cogno] Altro esempio [Tre] Altro esempio [PPJ11] Altro esempio [JJN07]

3.2.1 Subsection Esempio

```
1       GET /chat HTTP/1.1
2       Host: server.example.com
3       Upgrade: websocket
4       Connection: Upgrade
5       Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6       Origin: http://example.com
7       Sec-WebSocket-Protocol: chat, superchat
8       Sec-WebSocket-Version: 13
```

Codice 3.1: Esempio di listing

Versione	Chrome	Firefox	Internet Explorer	Opera	Safari
76	6	4.0	No	11.00(disabilitato)	5.0.1
7	No	6.0	No	No	No
10	14	7.0	HTML5 Labs	?	?
RFC 6455	16	11.0	10	12.10	6.0

Tabella 3.1: Esempio di Tabella

Versione	Android	Firefox Mob.	IE Mob.	Opera Mob.	Safari Mob.
76	?	?	?	?	?
7	?	?	?	?	?
10	?	7.0	?	?	?
RFC 6455	16(Chrome)	11.0	?	12.10	6.0

Tabella 3.2: Esempio di Tabella

Nelle Tabelle 3.1 e 3.2 è possibile vedere, rispettivamente per desktop e per mobile, il supporto dei vari browser per le diverse specifiche delle WebSocket.

Il codice completo dell'esempio è disponibile sul mio GitHub[1] (Esempio di link).

[1] https://github.com/...

3.2.2 Subsection Esempio

blablabla

4. Java

5. Compilatore

6. Esempi di traduzione

7. Conclusioni

Bibliografia

[bla] blabla. *HTML5*. URL: https://www.w3.org/TR/html5/.

[Cogno] Autorenome Cognome. «titolo». In: *ACM* (anno).

[JJN07] Fawcett Joe, McPeak Jeremy e C.Zakas Nicholas. *Ajax Guida per lo sviluppatore*. Milano: Hoepli, 2007.

[Mog91] Eugenio Moggi. «Notions of computation and monads». In: *Information And Computation* 93 (1 1991), pp. 55–92.

[PPJ11] Paperino Paolino, De Paperoni Paperone e Rockerducki John Davison. «Il deposito di Paperopoli». In: (2011).

[Tre] Trello. *Trello*. URL: https://trello.com/.

Indice analitico

Ringraziamenti

Ringrazio...