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

1.1 Motivazione

I linguaggi di programmazione più diffusi non sempre possono essere classificati solamente come procedurali, a oggetti, logici o funzionali: soprattutto con la crescente adozione di quest'ultimi nell'industria e con il progresso della tecnologia per l'esecuzione di algoritmi paralleli, l'introduzione di nuove *feature* ispirate alla programmazione funzionale è ben gradita dalla maggior parte degli sviluppatori.
Molti linguaggi moderni possono più opportunamente essere chiamati "ibridi" o "multi-paradigma", in virtù della combinazione di diverse metodologie.

`Java` nello specifico, a partire dalla versione 8, è stato arricchito con nuovi strumenti e costrutti che permettono di scrivere codice più conciso e dichiarativo: la domanda che ci si pone è quindi se sia possibile tradurre un linguaggio funzionale in codice imperativo, sfruttando le nuove funzionalità così come la programmazione a oggetti.

1.2 Obiettivi

Il progetto di cui questo documento è relazione ha come obiettivo principale la pianificazione e costruzione di un compilatore da codice sorgente funzionale a codice `Java`, con particolare attenzione alla semplicità di traduzione data dalle varie *feature* impiegate.

Il linguaggio dovrebbe essere sufficientemente espressivo da permettere la definizione di funzioni basilari e l'uso della ricorsione; le performance di compilazione ed esecuzione non sono aspetti prioritari, ma devono essere tali da consentire l'utilizzo pratico con fini educativi e dimostrativi.

1.3 Struttura della tesi

Al fine di offrire un quadro completo dello studio effettuato e del progetto, esplorando aspetti sia teorici che pratici, la tesi è suddivisa nei seguenti capitoli:

2. introduzione al paradigma funzionale e panoramica generale del linguaggio creato;
3. nozioni sui sistemi di tipo per linguaggi funzionali, sistema adottato e inferenza;
4. elenco delle caratteristiche di `Java` utili e primi esempi di traduzione;
5. descrizione del software realizzato, dettagli d'implementazione e ulteriori esempi;
6. conclusioni, risultati ottenuti, osservazioni finali e possibili sviluppi futuri.

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, [fan·èks] o [fan·ìks] in italiano.

2.1 Linguaggi funzionali

Nonostante i linguaggi di programmazione non si possano confinare all'interno di un solo paradigma, parlando di linguaggi di programmazione si fa comunque 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 codice che segue una precisa sequenza d'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`.

I linguaggi dichiarativi, invece, sono fondamentali per lo scopo del progetto: 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 d'interrogazione come `SQL`, linguaggi logici come `Prolog` e soprattutto i linguaggi funzionali: `Lisp`, `Clojure`, `Elixir`, `OCaml` e `Haskell` sono alcuni esempi.

Alla base dei linguaggi funzionali vi è il **lambda calcolo** [Chu32; Chu33]: un sistema formale definito dal matematico *Alonzo Church* (supervisore di *Alan Turing* durante il dottorato), equivalente alla macchina di Turing, ma fondato sulle funzioni pure.

La grammatica del lambda calcolo verrà presentata poco più avanti (sezione 2.2), ma le regole che ne governano il funzionamento e il modo in cui queste vengano utilizzate per ridurre le espressioni a una forma normale esulano dai fini di questo documento.
Rimane comunque rilevante elencare le principali qualità che un linguaggio funzionale usualmente 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.

2.1.1 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 d'input/output. Inoltre, molti linguaggi prevalentemente imperativi adottano ormai da tempo alcune caratteristiche tipiche dei linguaggi funzionali (e.g. `Rust`, il linguaggio più amato[1] dagli sviluppatori secondo i 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 invece completamente puri.
`Haskell` si avvale di un pattern di programmazione chiamato *monadi* [Mog91] per gestire le operazioni d'input/output e altre impurità, mantenendo le funzioni pure.

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*.

[1] Stack Overflow Developer Survey 2023 (`https://survey.stackoverflow.co/2023`), *Rust is the most admired language*

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

- **variabile**: simbolo rappresentante un parametro;
- **applicazione**: applicazione di funzione a un argomento (entrambi espressioni);
- **astrazione**: definizione di una funzione anonima, con un solo input x (variabile vincolata) e un solo output E (espressione); per definire funzioni con più parametri si debbono usare molteplici astrazioni annidate (tecnica detta *currying*, dal nome del matematico e logico *Haskell Brooks Curry*).

Modulo	M	::=	$nome \cdot L$	
Dichiarazione	D	::=	$?(schema\ di\ tipo) \cdot id = E$	funzione
Espressione	E	::=	c	costante
		|	x	variabile
		|	$E_l\ E_r$	applicazione
		|	$\lambda x\ .\ E$	astrazione
		|	L	let
		|	$\textbf{if}\ E_c\ \textbf{then}\ E_t\ \textbf{else}\ E_e$	if
Let	L	::=	$\textbf{let} \cdot D\ (\cdot D)^* \cdot \textbf{in}\ E$	

Figura 2.2: Grammatica di **Funx**

In Figura 2.2 è facile constatare la presenza delle ulteriori produzioni per la definizione del modulo corrente (informazione inclusa a prescindere dal fatto che il linguaggio per ora non supporti 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, di cui si parlerà più approfonditamente nella sezione 3.1.2.
Per quanto riguarda invece le espressioni, vengono introdotte tre nuove regole:

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

2.2.1 Zucchero sintattico

Con lo scopo di rendere il codice più leggibile, conciso e semplice, **Funx** introduce dello zucchero sintattico (del tutto simile a quello di `Haskell`). In Tabella 2.1 sono riportati l'indispensabile per evitare il parsing dell'indentazione, le semplificazioni comuni utili all'arricchimento del lambda calcolo, e infine tutti gli operatori simbolici supportati al momento (assieme alla notazione per indicarne associatività e precedenza).

Zucchero	**Sostituzione**
`\x -> e`	`λx . e`
`\x y -> e`	`λx . λy . e`
`f x y = e`	`f = λx . λy . e`

`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`

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

`if b then e1 else e2 fi`	`if b then e1 else e2`
`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: Zucchero sintattico

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 4.2 esporrà invece il motivo della traduzione degli operatori booleani binari in if.

Alcuni esempi di funzioni sono presentati nel Codice 2.1; seppur superflua, l'indentazione è inclusa per maggiore chiarezza.

```
1  main = factorial 20
2
3  factorial : Int -> Int
4  factorial n = if n == 0 then 1 else n * factorial (n - 1) fi
5
6  even : Int -> Bool
7  even = let
8         even1 : Int -> Bool
9         even1 n = if n == 0 then True else odd (n - 1) fi
10
11        odd : Int -> Bool
12        odd n = if n == 0 then False else even1 (n - 1) fi
13     in even1
14
15 gcd : Int -> Int -> Int
16 gcd a b = if b == 0 then a else gcd b (a % b) fi
17
18 xor : Bool -> Bool -> Bool
19 xor a b = (a || b) && !!(a && b)
```

Codice 2.1: Esempio di programma

3. Inferenza di tipo

Dopo aver discusso la sintassi di **Funx**, è importante far notare come i programmi non abbiano bisogno di annotazioni di tipo, nonostante siano stati adottati tipi statici.
In questo capitolo affronteremo l'argomento dei sistemi di tipo e dell'inferenza, meccanismo proprio di molti linguaggi, funzionali e non, che rende possibile la deduzione automatica del tipo di un termine basandosi sull'utilizzo delle variabili e delle funzioni.

3.1 Sistemi di tipo

Durante la genesi di ogni linguaggio di programmazione, una delle scelte più significative riguarda l'introduzione di un sistema per gestire i tipi di variabili ed espressioni.
Tali sistemi di tipo sono di fatto insiemi di regole logiche che permettono di assegnare una proprietà *"type"* a ciascuno dei termini del linguaggio che ne necessitano.
Sono principalmente suddivisi in due categorie:

- **tipizzazione statica**: i tipi sono definiti a tempo di compilazione e non possono cambiare mentre il programma è in esecuzione;
- **tipizzazione dinamica**: i tipi vengono stabiliti durante l'esecuzione e possono cambiare in qualsiasi momento.

Oltre a questa distinzione esistono varie sfumature e approcci differenti, informalmente classificati in base alla rigidità delle regole di tipizzazione. Si parla di *tipizzazione debole* quando ad esempio sono consentite conversioni implicite tra tipi diversi, *tipizzazione forte* se sono impedite, oppure qualora sia o meno disponibile l'aritmetica dei puntatori.

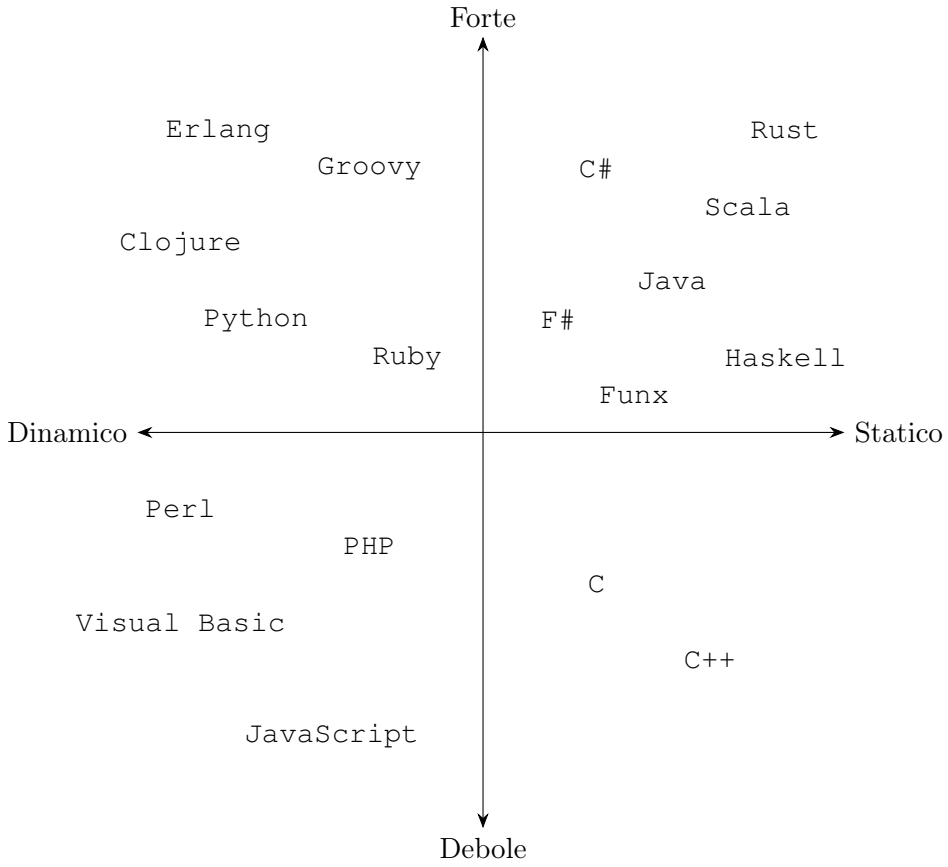

Figura 3.1: Alcuni linguaggi e loro sistemi di tipo

Grazie ai tipi dinamici, linguaggi quali `Python` e `JavaScript` permettono veloce prototipazione, flessibilità e codice più conciso, a discapito però di una più alta probabilità d'incontrare errori importanti a runtime, piuttosto che in fase di compilazione.

Al contrario, i tipi statici spesso migliorano naturalmente la mantenibilità di un progetto: viene limitata la possibilità di scorciatoie nello sviluppo, ma si hanno maggiori garanzie di correttezza, in quanto il compilatore può implementare ulteriori controlli e segnalare errori semantici più precisi già prima dell'esecuzione del programma.

D'altro canto, l'obbligo di specificare i tipi di ogni variabile, oggetto, funzione e parametro può risultare tedioso e talvolta ridondante; molti linguaggi moderni, tra cui `Haskell`, ovviano a quest'inconvenienza tramite l'uso dell'inferenza di tipo.

Gli algoritmi d'inferenza introducono numerosi benefici, in particolare:

- la scrittura del codice è meno onerosa per lo sviluppatore a prescindere dal sistema di tipi utilizzato, e diviene quindi estremamente vantaggioso utilizzare tipi statici;
- le annotazioni ora opzionali possono essere aggiunte dal programmatore quando vi sono casi difficili da disambiguare automaticamente, oppure per migliorare la leggibilità del codice;
- gli strumenti di sviluppo per il linguaggio possono sfruttare informazioni fornite dal motore inferenziale per suggerire il tipo delle espressioni e arricchire i messaggi di errore e di warning.

3.1.1 λ-cubo

Al fine di comprendere quale sistema il linguaggio **Funx** implementi, prima di discutere l'inferenza si vuol descrivere brevemente il *λ-cubo*, lambda cubo [Bar91], un modello introdotto per classificare i sistemi di tipo applicabili al lambda calcolo.
In Figura 3.2 è possibile osservare come la struttura del cubo abbia all'origine il *lambda calcolo semplicemente tipato* ($\lambda\!\rightarrow$) e come le tre dimensioni in cui si sviluppa rappresentino ciascuna un'estensione del sistema:

- **tipi dipendenti** ($\rightarrow$): la definizione dei tipi può dipendere dai valori delle variabili (implementati da linguaggi funzionali come `Agda`, `Coq` e `Idris`);
- **polimorfismo parametrico** ($\uparrow$): i tipi possono essere polimorfi, generalizzati tramite variabili di tipo (presenti nei sistemi adottati da `ML`, `OCaml` e `Haskell`);
- **costruttori di tipo** ($\nearrow$): capacità di costruire nuovi tipi a partire da tipi esistenti (`Haskell` ne fa grande uso poiché ogni nuovo tipo, dichiarato con la keyword `data`, è un nuovo costruttore di tipo).

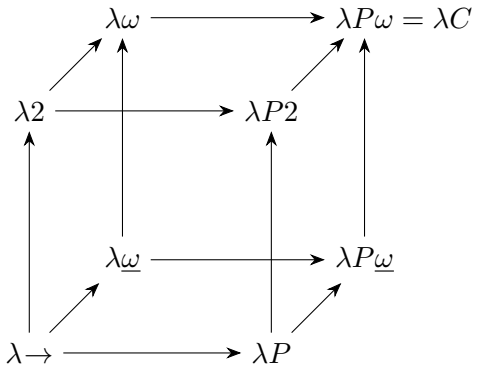

Figura 3.2: λ-cubo

Senza entrare troppo nei dettagli, in ordine crescente di potenza espressiva:

- $\lambda\!\rightarrow$ (*lambda calcolo semplicemente tipato*): tipi monomorfi;
- $\lambda\underline{\omega}$ (*lambda weak omega*): costruttori di tipo;
- $\lambda 2$ (*lambda due, lambda F, lambda calcolo polimorfico*): polimorfismo parametrico;
- λP (*lambda P*): tipi dipendenti;
- $\lambda P\underline{\omega}$ (*lambda P weak omega*): costruttori di tipo e tipi dipendenti;
- $\lambda\omega$ (*lambda omega*): costruttori di tipo e polimorfismo parametrico;
- $\lambda P2$ (*lambda P due*): polimorfismo parametrico e tipi dipendenti;
- $\lambda P\omega = \lambda C$ (*lambda P omega, lambda C, calcolo delle costruzioni*): combinazione di tutte le tre estensioni.

3.1.2 Sistema FC

Tra i vari sistemi di tipo per il lambda calcolo, uno dei più interessanti è *lambda F* (vertice $\lambda 2$ in Figura 3.2) poiché molto utile per la generalizzazione delle funzioni: un problema molto ricorrente nella programmazione con qualsiasi linguaggio è infatti la duplicazione di codice per funzioni che svolgono operazioni simili su tipi diversi.

Il *sistema F* risolve tale problema introducendo il **polimorfismo parametrico** e di conseguenza la distinzione tra tipi monomorfi (monotipi) e tipi polimorfi (politipi).

I tipi delle funzioni possono essere caratterizzati tramite quantificatori universali e variabili di tipo ove sia necessario un tipo generico (spesso vengono usate singole lettere dell'alfabeto greco o latino).

Tuttavia, $\lambda 2$ nella sua forma più pura, oltre a non essere un sistema Turing-completo (è possibile definire solamente la ricorsione primitiva), rende l'inferenza di tipo trattata nella sezione 3.2 un problema non decidibile [Wel99].

Pertanto, il linguaggio `Haskell` non implementa semplicemente il *sistema F*, ma piuttosto una versione ristretta di $\lambda\omega$ chiamata *sistema FC* [Eis15].

Quest'ultima include anche i costruttori di tipo (*funzioni di tipo* in **Funx**), frenando però il polimorfismo ai cosiddetti *tipi polimorfici di rango 1* (*polimorfismo predicativo*): tale limitazione si manifesta nella scrittura di tutti i quantificatori universali all'inizio di un tipo polimorfo (che prende il nome di *schema di tipo*).

Le versioni invece più espressive e più vicine a $\lambda\omega$ sono:

- *polimorfismo di rango superiore*: supporta quantificatori universali in qualsiasi punto nelle definizioni delle funzioni (e.g. `Bool -> (forall b . b -> b)`); `Haskell` lo realizza con l'estensione `RankNTypes` del compilatore `GHC`, mentre offre anche l'estensione `Rank2Types`, per la quale l'inferenza rimane decidibile;

- *polimorfismo impredicativo*: permette di quantificare le variabili di tipo in modo arbitrario, anche e soprattutto all'interno dei costruttori di tipo (e.g. `Maybe (forall a . a -> a) -> Bool`, possibile in `Haskell` abilitando l'estensione `ImpredicativeTypes`).

Il linguaggio **Funx** ovviamente non è correntemente in grado di supportare queste estensioni del sistema di tipo, così come non è possibile definire nuovi tipi o fare uso di *classi di tipo* simili a quelle proprie di `Haskell`. Affermare che **Funx** adotti il *sistema FC* potrebbe lasciare intendere un linguaggio più espressivo di quanto non sia in realtà: è dunque più opportuno realizzare il *sistema HM*, di cui il *sistema FC* è un ampliamento, e che comunque ben si presta allo scopo principe di traduzione in `Java`.

In Tabella 3.1 si possono osservare i tipi di alcune funzioni polimorfe di `Haskell`: la sintassi di **Funx** è molto simile (identica in ognuno dei casi presentati), con l'eccezione che la parola chiave `forall` è completamente assente dal linguaggio, in quanto ogni identificatore che inizia con una lettera minuscola è considerato una variabile di tipo da quantificare universalmente (sezione 5.1.1).

Funzione	**Schema**
`id`	`forall a . a -> a`
`const`	`forall a b . a -> b -> a`
`(.)`	`forall b c a . (b -> c) -> (a -> b) -> a -> c`
`flip`	`forall a b c . (a -> b -> c) -> b -> a -> c`
`($)`	`forall a b . (a -> b) -> a -> b`
`(&)`	`forall a b . a -> (a -> b) -> b`

Tabella 3.1: Esempi di funzioni polimorfe

In Figura 3.3 è mostrata la grammatica per la definizione dei tipi implementati nel linguaggio **Funx**. Si noti come i tipi monomorfi siano solo variabili di tipo o applicazioni di funzioni ad altri tipi; al momento il linguaggio mette a disposizione le funzioni di tipo più elementari, la cui arietà è indicata in pedice.

Schema di tipo	σ	$::=$	τ	monotipo
		$\mid$	$\forall\ \alpha\ .\ \sigma$	politipo
Tipo	τ	$::=$	α	variabile di tipo
		$\mid$	$F\ \tau \ldots \tau$	applicazione di funzione di tipo
Funzione di tipo	F	$::=$	$\rightarrow_2$	funzione
		$\mid$	$Bool_0$	booleano
		$\mid$	Int_0	intero

Figura 3.3: Grammatica del sistema di tipo di **Funx**

3.2 Inferenza secondo Hindley–Milner

Come già accennato, il sistema *Hindley–Milner (HM)* [Hin69; Mil78], o *Damas-Hindley-Milner* [DM82], è un sistema di tipo per il lambda calcolo con polimorfismo parametrico largamente utilizzato in molti moderni linguaggi di programmazione ad alto livello. Il maggiore punto di forza del sistema è il relativo metodo d'inferenza, in grado di dedurre automaticamente il tipo di un termine senza annotazioni esplicite fornite dagli sviluppatori.

Prima di presentare l'algoritmo d'inferenza è necessario complementare le nozioni di lambda calcolo esteso (Figure 2.1 e 2.2) e del sistema di **Funx** (Figura 3.3) con due concetti fondamentali:

- **contesto**: una insieme di associazioni tra variabili e schemi di tipo, che rappresenta lo stato corrente dell'ambiente in cui un termine viene analizzato; l'unione tra la grammatica del linguaggio e il sistema di tipi è data da un giudizio di tipo effettuato nel contesto su un termine (espressione);
- **variabili libere**: l'insieme delle variabili libere di un tipo è semplicemente il complemento delle variabili vincolate, quantificate universalmente in un politipo.

Contesto	Γ	$::=$	ϵ	contesto vuoto
		$\mid$	$\Gamma + x : \sigma$	aggiunta di associazione
Giudizio di tipo		$::=$	$\Gamma \vdash E : \sigma$	

$free(\forall\ \alpha\ .\ \sigma)$	$=$	$free(\sigma) - \{\alpha\}$	tipo polimorfo
$free(\alpha)$	$=$	$\{\alpha\}$	variabile di tipo
$free(F\ \tau_1 \ldots \tau_n)$	$=$	$\bigcup_{i=1}^{n} free(\tau_i)$	applicazione di funzione di tipo
$free(\Gamma)$	$=$	$\bigcup_{x:\ \sigma \in \Gamma} free(\sigma)$	contesto
$free(\Gamma \vdash E\ :\ \sigma)$	$=$	$free(\sigma) - free(\Gamma)$	giudizio di tipo

Figura 3.4: Definizioni di contesto e variabili libere

Le regole d'inferenza [Clé+86] del *sistema HM*, riportate in Figura 3.5, informano il comportamento dell'*algoritmo* $\mathcal{W}$, implementazione dell'inferenza di tipo.
In aggiunta a principi già noti, le peculiarità non immediatamente chiare sono:

- *constantType*: funzione che restituisce il tipo di una costante;
- $\sigma \sqsubseteq \tau$: indica intuitivamente che σ è più generale di τ (τ è un'*istanza* di σ);
- $Clos_\Gamma$: ottiene la *chiusura* di una variabile, ossia il suo tipo più generale, quantificando universalmente le variabili libere del tipo iniziale.

$$\frac{\tau = constantType(c)}{\Gamma \vdash c : \tau} \quad \text{[costante]}$$

$$\frac{x : \sigma \in \Gamma \qquad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau} \quad \text{[variabile]}$$

$$\frac{\Gamma \vdash E_l : \tau \to \beta \qquad \Gamma \vdash E_r : \tau}{\Gamma \vdash E_l\ E_r : \beta} \quad \text{[applicazione]}$$

$$\frac{\Gamma + x : \beta \vdash E : \tau}{\Gamma \vdash \lambda x\ .\ E : \beta \to \tau} \quad \text{[astrazione]}$$

$$\frac{\overline{\Gamma} = \Gamma + \vec{x} : \vec{\beta} \qquad \overline{\Gamma} \vdash \vec{E} : \vec{\kappa} \qquad \overline{\Gamma}\{Clos_{\overline{\Gamma}}(\vec{\kappa})/\vec{\beta}\} \vdash E_i : \tau}{\overline{\Gamma} \vdash \textbf{let}\ \vec{x} = \vec{E}\ \textbf{in}\ E_i : \tau} \quad \text{[let]}$$

$$Clos_\Gamma(\tau) = \forall\ \hat{\alpha}\ .\ \tau \qquad \hat{\alpha} = free(\tau) - free(\Gamma)$$

$$\frac{\Gamma \vdash E_c : Bool \qquad \Gamma \vdash E_t : \tau \qquad \Gamma \vdash E_e : \tau}{\Gamma \vdash \textbf{if}\ E_c\ \textbf{then}\ E_t\ \textbf{else}\ E_e : \tau} \quad \text{[if]}$$

Figura 3.5: Regole di inferenza del *sistema HM* in **Funx**

L'*algoritmo* $\mathcal{W}$ [LY98] in Figura 3.6 fornisce un'implementazione del procedimento finora descritto, servendosi delle regole d'inferenza e dettagliando i passi per la deduzione del tipo di ogni espressione disponibile in **Funx**.
L'algoritmo appare nel codice del compilatore in una forma molto simile, seppur talvolta più complessa, come nel caso dell'espressione `let` (sezione 5.3.2).

$$\mathcal{W} : Context \times Expression \to Substitution \times Type$$

$$\begin{array}{lcll}
\mathcal{W}(\Gamma, c) & = & (id, constantType(c)) \\
\mathcal{W}(\Gamma, x) & = & \text{let} & \forall\ \vec{\alpha}\ .\ \tau = \Gamma(x),\ new\ \vec{\beta} \\
 & & \text{in} & (id, \{\vec{\beta}/\vec{\alpha}\}\tau) \\
\mathcal{W}(\Gamma, E_l\ E_r) & = & \text{let} & (S_l, \tau_l) = \mathcal{W}(\Gamma, E_l) \\
 & & & (S_r, \tau_r) = \mathcal{W}(S_l\Gamma, E_r) \\
 & & & S_a = \mathcal{U}(S_r\tau_l, \tau_r \to \beta),\ new\ \beta \\
 & & \text{in} & (S_aS_rS_l, S_a\beta) \\
\mathcal{W}(\Gamma, \lambda x\ .\ E) & = & \text{let} & (S, \tau) = \mathcal{W}(\Gamma + x : \beta, E),\ new\ \beta \\
 & & \text{in} & (S, S\beta \to \tau) \\
\mathcal{W}(\Gamma, \textbf{let}\ \vec{x} = \vec{E}\ \textbf{in}\ E_i) & = & \text{let} & \overline{\Gamma} = \Gamma + \vec{x} : \vec{\beta},\ new\ \vec{\beta} \\
 & & & (\vec{S}, \vec{\kappa}) = \mathcal{W}(\overline{\Gamma}, \vec{E}) \\
 & & & (S_i, \tau) = \mathcal{W}(\vec{S}\,\overline{\Gamma}\{Clos_{\vec{S}\,\overline{\Gamma}}(\vec{\kappa})/\vec{\beta}\}, E_i) \\
 & & \text{in} & (S_i\vec{S}, \tau) \\
\mathcal{W}(\Gamma, \textbf{if}\ E_c\ \textbf{then}\ E_t\ \textbf{else}\ E_e) & = & \text{let} & (S_c, \tau_c) = \mathcal{W}(\Gamma, E_c) \\
 & & & (S_t, \tau_t) = \mathcal{W}(S_c\Gamma, E_t) \\
 & & & (S_e, \tau_e) = \mathcal{W}(S_tS_c\Gamma, E_e) \\
 & & & S_{cb} = \mathcal{U}(\tau_c, Bool) \\
 & & & S_{te} = \mathcal{U}(S_e\tau_t, \tau_e) \\
 & & \text{in} & (S_{te}S_{cb}S_eS_tS_c, S_{te}\tau_t)
\end{array}$$

Figura 3.6: Algoritmo $\mathcal{W}$

Dato un contesto in cui analizzare un'espressione, $\mathcal{W}$ restituisce un tipo e una sostituzione. Quest'ultima può essere vuota (id), generata manualmente ($\{\tau_2/\tau_1\}$) o dall'*unificazione* di altri tipi, e rappresenta una mappatura tra variabili di tipo e altri tipi; può essere applicata a tipi e contesti.

La funzione *new* esprime la creazione di nuove variabili di tipo, con il vincolo che non siano state ancora usate dall'algoritmo.

Infine, l'*unificazione* (funzione $\mathcal{U}$) è un'operazione altrettanto importante per l'inferenza di tipo, con la quale si cerca di trovare una sostituzione che renda due tipi uguali.

$$\mathcal{U} : Type \times Type \to Substitution$$

$$\begin{array}{lcl}
\mathcal{U}(\alpha, \alpha) & = & id \\
\mathcal{U}(\alpha, \tau) & = & \{\tau/\alpha\} \\
\mathcal{U}(\tau, \alpha) & = & \{\tau/\alpha\} \\
\mathcal{U}(F\ \vec{\tau}, F\ \vec{\kappa}) & = & \mathcal{U}(\vec{\tau}, \vec{\kappa})
\end{array}$$

Figura 3.7: Funzione $\mathcal{U}$

4. Java

Parallelamente alle prime fasi di sviluppo è stata svolta un'analisi di `Java`[1] per valutare quali fossero le caratteristiche del linguaggio utili alla traduzione di codice **Funx**.
Questo capitolo riporta pertanto una breve panoramica delle principali funzionalità impiegate, accompagnate da esempi di traduzione che illustrano alcuni dei risultati ottenibili con il compilatore (si ricorda che altri esempi sono esibiti durante il Capitolo 5).

Le funzioni non dichiarate all'interno degli esempi stessi provengono dalla piccola libreria standard (suddivisa in `FunxPrelude` e `JavaPrelude`, di cui la seconda include funzioni necessariamente definite in `Java`), parzialmente presentata nella Tabella 2.1 con gli operatori simbolici.

4.1 Interfacce funzionali

Nel tradurre un linguaggio funzionale viene naturale pensare immediatamente alle *interfacce funzionali* e *lambda espressioni* introdotte in `Java` 8[2] per rappresentare funzioni anonime: l'interfaccia generica `Function` è la più adatta a riprodurre il comportamento dell'astrazione, come mostrato nei Codici 4.1 e 4.2.

```
1 constant : a -> b -> a
2 constant x = \y -> x
```

Codice 4.1: Semplice funzione in **Funx**

```
1 public static <a, b> Function<a, Function<b, a>> constant() {
2     return (x -> (y -> x));
3 }
```

Codice 4.2: Corrispondente metodo in `Java`

Dato il naturale *currying* di oggetti `Function`, questo tipo di traduzione ha il vantaggio di permettere l'applicazione parziale di funzioni (tramite una sequenza di `apply()` in numero minore rispetto al totale degli input), ma il grande svantaggio della creazione di una nuova istanza della funzione per ogni chiamata.

Poiché `constant` ha tipo polimorfo (sezione 4.3), il metodo utilizza parametri di tipo e deve quindi necessariamente restituire un nuovo oggetto con ogni chiamata: nonostante la performance delle traduzioni non sia un obiettivo primario del progetto, la versione attuale del compilatore fa uso di alcune piccole ottimizzazioni nella trasposizione delle funzioni monomorfe, approfondite nella sezione 5.4.3.

[1] OpenJDK (`https://openjdk.org`)
[2] OpenJDK 8 (`https://openjdk.org/projects/jdk8`)

In aggiunta alla classe `Function`, nella parte nativa (codice `Java`) della libreria standard di **Funx** è definita un'ulteriore interfaccia funzionale per creare espressioni `let`: in questo caso vi è un grande utilizzo di classi anonime, potenzialmente annidate, rappresentanti le dichiarazioni locali e l'espressione principale (metodo `eval`).

```java
@FunctionalInterface
public interface Let<T> {
   T _eval();
}
```

Codice 4.3: Interfaccia funzionale per espressioni `let`

```
hundredsSum : Int -> Int -> Int
hundredsSum = let
      on : (a -> a -> b) -> (c -> a) -> c -> c -> b
      on op f x y = op (f x) (f y)
   in on add (multiply 100)
```

Codice 4.4: Espressione `let` in **Funx**

```java
public static Function<Long, Function<Long, Long>> hundredsSum;

static {
   hundredsSum = (new Let<>() {
      private <a, b, c>
         Function<
               Function<a, Function<a, b>>,
               Function<Function<c, a>, Function<c, Function<c, b>>>>
            on() {
         return (op -> (f -> (x -> (y -> op.apply(f.apply(x)).apply(f.apply(y)))))));
      }

      @Override
      public Function<Long, Function<Long, Long>> _eval() {
         return this.<Long, Long, Long>on().apply(add).apply(multiply.apply(100L));
      }
   })._eval();
}
```

Codice 4.5: Corrispondente classe anonima in `Java`

Nei Codici 4.4 e 4.5 si può vedere che la funzione `hundredsSum` è implementata attraverso la chiamata al metodo principale dell'interfaccia funzionale `Let`, realizzato internamente alla classe anonima con il supporto del metodo polimorfo `on`.

Inoltre, è immediatamente evidente come le traduzioni in `Java` siano progressivamente più complesse e meno leggibili con l'introduzione di nuove funzionalità: l'esempio più eclatante è dato proprio dal tipo di ritorno del metodo locale `on`, divenuto di difficile comprensione rispetto alla sintassi molto concisa del linguaggio funzionale.

4.2 Operatore ternario

Una delle peculiarità della traduzione da **Funx** a `Java` è l'uso dell'operatore ternario (`condition ? thenBranch : elseBranch`) ogni qualvolta siano presenti espressioni condizionali `if-then-else`.

I linguaggi funzionali sfruttano spesso una caratteristica (non menzionata nella sezione 2.1) che prende il nome di *lazy evaluation* (valutazione pigra): fino a quando il risultato di un'espressione non è richiesto per un successivo calcolo, questa non verrà completamente valutata.
Oltre a offrire molteplici possibilità di ottimizzazione dal punto di vista del tempo di esecuzione, tale comportamento è molto comodo nella scrittura di funzioni che per esempio potrebbero terminare prima del previsto o magari effettuare computazioni su strutture dati infinite. I linguaggi che sono *lazy evaluated* di default impiegano nella maggior parte dei casi un *garbage collector* per liberare la memoria occupata dalle espressioni non valutate e non più rilevanti.

Il linguaggio `Java` non adotta la *lazy evaluation* di default se non in casi particolari, tra cui gli operatori booleani binari, il costrutto `if-then-else` (e corrispondente operatore ternario) e altre funzionalità più avanzate tra cui gli *stream* e le *lambda espressioni* già viste. Utilizzando quest'ultime si potrebbero ottenere risultati simili, in termini di valutazione pigra, a quelli di un linguaggio funzionale; tuttavia, rendere **Funx** un linguaggio completamente pigro avrebbe comportato una traduzione indubbiamente ancora più complessa, molteplici rischi di peggiorare le prestazioni dei programmi e un'implementazione del compilatore che va oltre lo scopo di questo progetto.

Nonostante ciò, la scelta di ridurre gli operatori booleani binari (*and* e *or*) a espressioni con operatore ternario è stata considerata quasi obbligatoria per conservarne la natura pigra: gli operatori ternari utilizzati a questo scopo derivano direttamente dalla costruzione dell'**AST** (sezione 5.2.2), motivo per cui non vengono riconvertiti in operatori nativi di `Java` in fase di traduzione.

```
power : Int -> Int -> Int
power b e = if e == 0 then 1 else b * power b (e - 1) fi

xor : Bool -> Bool -> Bool
xor a b = (a || b) && !!(a && b)
```

Codice 4.6: If e operatori booleani in **Funx**

```
public static Function<Long, Function<Long, Long>> power;

public static Function<Boolean, Function<Boolean, Boolean>> xor;

static {
   power = (b -> (e -> ((JavaPrelude.<Long>equalsEquals().apply(e).apply(0L))
      ? (1L)
      : (multiply.apply(b)
         .apply(power.apply(b).apply(subtract.apply(e).apply(1L)))))));

   xor = (a -> (b ->
      ((((a) ? (true) : (b))) ? (not.apply(((a) ? (b) : (false)))) : (false))));
}
```

Codice 4.7: Corrispondenti operatori ternari in `Java`

4.3 Tipi generici

Il sistema di tipo di **Funx** necessita la traduzione di funzioni polimorfe, e la soluzione più semplice e idiomatica in `Java` è l'utilizzo dei *generics*: tramite i parametri di tipo generici è possibile definire classi e metodi che agiscono su molteplici tipi di dati, implementando comportamenti che possono essere condivisi dai diversi elementi del dominio di tipi delle funzioni rappresentate.

Nel contesto del *sistema HM* di **Funx**, le variabili quantificate universalmente nei politipi hanno una diretta corrispondenza con i parametri di tipo che possono essere dichiarati tra i modificatori di visibilità e il tipo di ritorno di un metodo, il quale a sua volta combacia con la parte interna dello schema di tipo.

Nei Codici 4.8 e 4.9 si può notare come `Java` non sempre sia in grado d'inferire i tipi desiderati per le funzioni polimorfe: queste devono infatti essere istanziate esplicitamente usando la classe di appartenenza e le parentesi angolari (questa limitazione richiederà alcuni espedienti in casi limite illustrati nelle sezioni 5.4.4 e 5.4.5).

```
1  sumToN : Int -> Int
2  sumToN = let
3        ap : (a -> b -> c) -> (a -> b) -> a -> c
4        ap op f x = op x (f x)
5     in (flip divide 2) . ap multiply (add 1)
```

Codice 4.8: Scrittura e utilizzo di funzioni polimorfe in **Funx**

```
1  public static Function<Long, Long> sumToN;
2
3  static {
4     sumToN = (new Let<>() {
5        private <a, b, c>
6           Function<
7              Function<a, Function<b, c>>,
8              Function<Function<a, b>, Function<a, c>>>
9                 ap() {
10          return (op -> (f -> (x -> op.apply(x).apply(f.apply(x)))));
11       }
12
13       @Override
14       public Function<Long, Long> _eval() {
15          return FunxPrelude.<Long, Long, Long>compose()
16             .apply(FunxPrelude.<Long, Long, Long>flip().apply(divide).apply(2L))
17             .apply(this.<Long, Long, Long>ap().apply(multiply).apply(add.apply(1L)));
18       }
19    })._eval();
20 }
```

Codice 4.9: Corrispondenti proprietà e metodi `Java`

5. Compilatore

Il software per il progetto è stato sviluppato mantenendo il codice sul repository `GitHub` **Funx-jt**[1], nel cui nome *"jt"* è l'acronimo per *"Java Transpiler"*.
La prima release stabile è disponibile sul repository (`Funx-jt-0.1.1`) con una *Command Line Interface* per traduzione ed esecuzione di programmi **Funx**; è stato utilizzato `Gradle`[2] per la compilazione e la gestione delle dipendenze, assieme alla più recente versione *Long Term Support (LTS)* di `Java`, OpenJDK 21[3].

Nel corso di questo capitolo si discuteranno le fasi di compilazione e la struttura del software, analizzando nel dettaglio le parti più importanti.

```
Usage: Funx-jt [-hV] [COMMAND]
A Funx to Java source transpiler.
  -h, --help      Show this help message and exit.
  -V, --version   Print version information and exit.
Commands:
  t  Transpile a funx program
  r  Run a funx program
```

```
Usage: Funx-jt t [-adFhIJp] [-o=<outputDir>] <input>
Transpile a funx program
      <input>                Input funx source file
  -a, --ast                  Output abstract syntax tree visualization
  -d, --dot                  Keep .dot file after AST visualization
  -F, --no-fancy-types       Don't use fancy types for inference annotations
  -h, --help                 Show this help message and exit.
  -I, --no-inference         Omit AST type inference annotations
  -J, --no-java              Skip Java transpilation
  -o, --output=<outputDir>   Output directory
  -p, --parse-tree           Output raw parse tree visualization
```

```
Usage: Funx-jt r [-h] <input>
Run a funx program
      <input>   Input java source file
  -h, --help    Show this help message and exit.
```

Figura 5.1: Possibili comandi e opzioni della *CLI*

[1] massimopavoni/Funx-jt (`https://github.com/massimopavoni/Funx-jt`)
[2] Gradle Build Tool (`https://gradle.org`)
[3] OpenJDK 21 (`https://openjdk.org/projects/jdk/21`)

5.1 ANTLR

Al fine di semplificare lo sviluppo di *lexer* e *parser* per il linguaggio funzionale ideato è stato scelto il generatore di *parser* chiamato `ANTLR`[4] [PQ95; Par13].
Grazie a tale strumento il processo iterativo di creazione della grammatica di **Funx** è stato notevolmente semplificato e accelerato, in quanto `ANTLR` mette a disposizione del programmatore un linguaggio per definire uno o più file di specifica per lessico e sintassi (directory `Funx-jt/src/main/antlr` nel repository): questi vengono poi processati per generare il codice sorgente del *lexer* e del *parser*.

5.1.1 Analisi lessicale

Data la probabile complessità delle regole della grammatica di **Funx**, fin dall'inizio la definizione dei *token* (lessemi) del linguaggio è stata separata dalla specifica del *parser*.
Il file `FunxLexer.g4` descrive i lessemi dividendoli nelle seguenti categorie:

1. *whitespace*: caratteri di spaziatura e tabulazione;
2. *comments*: commenti di linea e blocco;
3. *keywords*: parole chiave del linguaggio;
4. *Java keywords*: parole chiave del linguaggio Java, da evitare;
5. *types*: tipi di dato (funzioni di tipo con arità 0);
6. *literals*: costanti booleane e numeriche;
7. *variables*: identificatori per variabili di tipo o nomi di funzioni;
8. *module*: identificatori per il modulo;
9. vari operatori simbolici per:
    - *bool*: valori booleani;
    - *comparison*: confronti tra numeri;
    - *arithmetic*: operazioni aritmetiche;
    - *other symbols*: simboli della sintassi (come `->`) e varie funzioni di libreria;
10. *delimiters*: parentesi tonde, quadre e graffe.

Le categorie 1 e 2 contengono *token* da scartare, tranne `NEWLINE`, mentre la categoria 4 è utile qualora eventualmente si permetta allo sviluppatore di utilizzare tali parole chiave riservate, effettuando una rinomina automatica; le categorie 7 e 8 devono necessariamente apparire dopo le categorie 3 e 4, poiché tra *keyword* e identificatori di ogni genere le prime devono avere la precedenza (la posizione della categoria 5 tiene conto di una possibile futura estensione per consentire la creazione di nuovi tipi).

Oltre alle categorie illustrate, in testa al file sono presenti dei cosiddetti *fragment* (frammenti) che semplificano le espressioni regolari dei *token* e complessivamente aumentano la leggibilità della specifica.

[4] ANother Tool for Language Recognition (`https://www.antlr.org`)

```
lexer grammar FunxLexer;

// Fragments
fragment LALPHA: [a-z];
fragment UALPHA: [A-Z];
fragment ALPHA: LALPHA | UALPHA;
fragment ALPHA_: ALPHA | UnderScore;

fragment DIGIT: [0-9];
fragment DECIMAL: DIGIT+;

// Whitespace
NEWLINE: '\r'? '\n' | '\r';

TAB: [\t]+ -> skip;
WS: [\u0020\u00a0\u1680\u2000\u200a\u202f\u205f\u3000]+ -> skip;

// Comments
CloseMultiComment: './';
OpenMultiComment: '/.';
SingleComment: '//';

COMMENT: SingleComment ~[\r\n]* -> skip;
MULTICOMMENT: OpenMultiComment .*? CloseMultiComment -> skip;

// Keywords
ELSE: 'else';
FI: 'fi';
IF: 'if';
IN: 'in';
LET: 'let';

// Java keywords
RESERVED_JAVA_KEYWORD: 'abstract' | 'assert' | 'boolean' | 'break' | 'byte' | [...];

// Types
TYPE: BOOLTYPE | INTTYPE;
BOOLTYPE: 'Bool';

// Literals
INT: DECIMAL | OpenParen '-' DECIMAL CloseParen;

// Variables
VARID: LALPHA (ALPHA_ | DIGIT)*;

// Module
MODULEID: UALPHA (ALPHA_ | DIGIT)*;

// Bool
And: '&&';
Not: '!!';

// Comparison
EqualsEquals: '==';
NotEquals: '!=';

// Arithmetic
Add: '+';

// Other symbols
UnderScore: '_';
Arrow: '->';

// Delimiters
OpenParen: '(';
CloseParen: ')';
```

Codice 5.1: Alcuni *token* del *lexer*

5.1.2 Analisi sintattica

Il file `FunxParser.g4` contiene le regole concrete della grammatica di **Funx**: nonostante la somiglianza con le grammatiche delle Figure 2.2 e 3.3, è evidente che queste non collimino esattamente a causa di zucchero sintattico e requisiti di `ANTLR`.

Lo strumento utilizzato, infatti, è un generatore di *parser* di tipo *top-down* per grammatiche *LL*, le quali in generale non supportano regole ricorsive a sinistra.

Essendo tali regole spesso comuni nella definizione di qualsiasi linguaggio di programmazione, **Funx** incluso, `ANTLRv4` offre un diverso tipo di parsing, detto *Adaptive LL(*)* [PF11; PHF14]: quest'ultimo è in grado di riscrivere automaticamente le grammatiche, eliminando la ricorsione a sinistra diretta (e.g. linee 36 e 38-43), così da non incorrere in regole ambigue che potrebbero causare *backtracking* e conseguente *overhead*.

Il Codice 5.2 riporta integralmente le regole concrete della sintassi di **Funx**, tra cui:

- *module*: nome del modulo, funzione `main` opzionale e dichiarazioni globali;
- *main*: funzione `main`, diversa dalle dichiarazioni classiche per l'assenza di schema di tipo e parametri lambda;
- *declaration*: funzione con nome, tipo e parametri (e opzionalmente `with` per funzioni locali);
- *typeElems*: tipo di una funzione, definito ricorsivamente secondo la grammatica del sistema di tipo di **Funx**;
- *statement*: per evitare ricorsione a sinistra indiretta, la separazione tra *statement* ed *expression* forza l'uso di parentesi nei casi in cui lambda astrazioni, let e if siano usati all'interno di un'espressione;
- *expression*: racchiude l'applicazione funzionale, tutte le regole relative agli operatori simbolici, specificandone la priorità implicita (Tabella 2.1), e le espressioni primarie (costanti, variabili e parentesi per controllare la precedenza);
- *lambda, let, if*: corrispondenti alle produzioni per astrazione, let e if della grammatica formale.

```
parser grammar FunxParser;
options { tokenVocab = FunxLexer; }

// Module
module: (MODULE MODULEID (Dot MODULEID)* NEWLINE+)?
    (main NEWLINE+)? declarations EOF;

declarations: declaration (NEWLINE declaration?)*;

main: id = MAIN Equals statement with?;

// Declaration
declaration: (declarationScheme NEWLINE)?
    id = VARID lambdaParams? Equals statement with?;

declarationScheme: id = VARID Colon typeElems;

with: NEWLINE WITH localDeclarations OUT;

localDeclarations: NEWLINE declarations NEWLINE;

// Type
typeElems: OpenParen typeElems CloseParen # parenType
    | VARID # typeVar
    | TYPE # namedType
    | <assoc = right> typeElems Arrow typeElems # arrowType;

// Statement
statement: expression # expressionStatement
    | lambda # lambdaStatement
    | let # letStatement
    | ifS # ifStatement;

// Expression
expression: primary # primExpression
    | expression expression # appExpression
    | <assoc = right> expression bop = Dot expression # composeExpression
    | expression bop = (Divide | Modulo | Multiply) expression # divModMultExpression
    | expression bop = (Add | Subtract) expression # addSubExpression
    | expression
       bop = (GreaterThan | GreaterThanEquals | LessThan | LessThanEquals)
       expression # compExpression
    | expression bop = (EqualsEquals | NotEquals) expression # eqExpression
    | uop = Not expression # notExpression
    | <assoc = right> expression bop = And expression # andExpression
    | <assoc = right> expression bop = Or expression # orExpression
    | <assoc = right> expression bop = Dollar expression # rightAppExpression;

primary: OpenParen statement CloseParen # parenPrimary
    | constant # constPrimary | VARID # varPrimary;

// Lambda
lambda: Backslash lambdaParams? Arrow statement;

lambdaParams: VARID+;

// Let
let: LET localDeclarations IN statement;

// If
ifS: IF statement THEN statement ELSE statement FI;

// Constant
constant: BOOL | numConstant;

numConstant: INT;
```

Codice 5.2: Grammatica per il *parser*

5.2 Albero sintattico astratto

Il *parser* generato da `ANTLR` utilizza i *token* identificati dal *lexer* per costruire un albero di parsing concreto (*CST, concrete syntax tree*), molto complesso e poco comprensibile. È dunque opportuno astrarre i dettagli del parsing in una struttura più semplice e facile da interpretare, che rispecchia precisamente la grammatica del linguaggio (Figura 2.2). Tale operazione è effettuata dalla grande maggioranza dei compilatori moderni e il risultato è chiamato **AST**, *abstract syntax tree* (albero sintattico astratto).

5.2.1 Gerarchia delle classi

Il primo passo per la costruzione di un **AST** per la sintassi di **Funx** è la definizione di una gerarchia di classi `Java` che rappresentano i nodi dell'albero (package `com.github.massimopavoni.funx.jt.ast.node`).

La classe astratta `ASTNode` è la radice dell'ordinamento poiché sarà usata per gli oggetti creati a partire dal *CST*: contiene la proprietà `inputPosition` per facilitare la segnalazione di errori e vincola le classi figlie all'implementazione del metodo astratto `accept` per visitare i nodi.
Un'altra classe astratta, derivata dalla precedente, è `Expression`, la quale identifica un'espressione ed è dotata di alcuni campi e metodi utili all'inferenza di tipo.

Ogni altra sottoclasse (a eccezione di `Declarations`, utilizzata per dichiarazioni globali e locali) è una trascrizione in `Java` delle produzioni della grammatica formale:

- `Module`: modulo del programma;
- `Declaration`: dichiarazione di funzione;
- `Constant`: termini costanti;
- `Variable`: simboli per variabili;
- `Application`: applicazione di funzione;
- `Lambda`: astrazione per le funzioni anonime;
- `Let`: contenitore di dichiarazioni locali;
- `If`: costrutto condizionale.

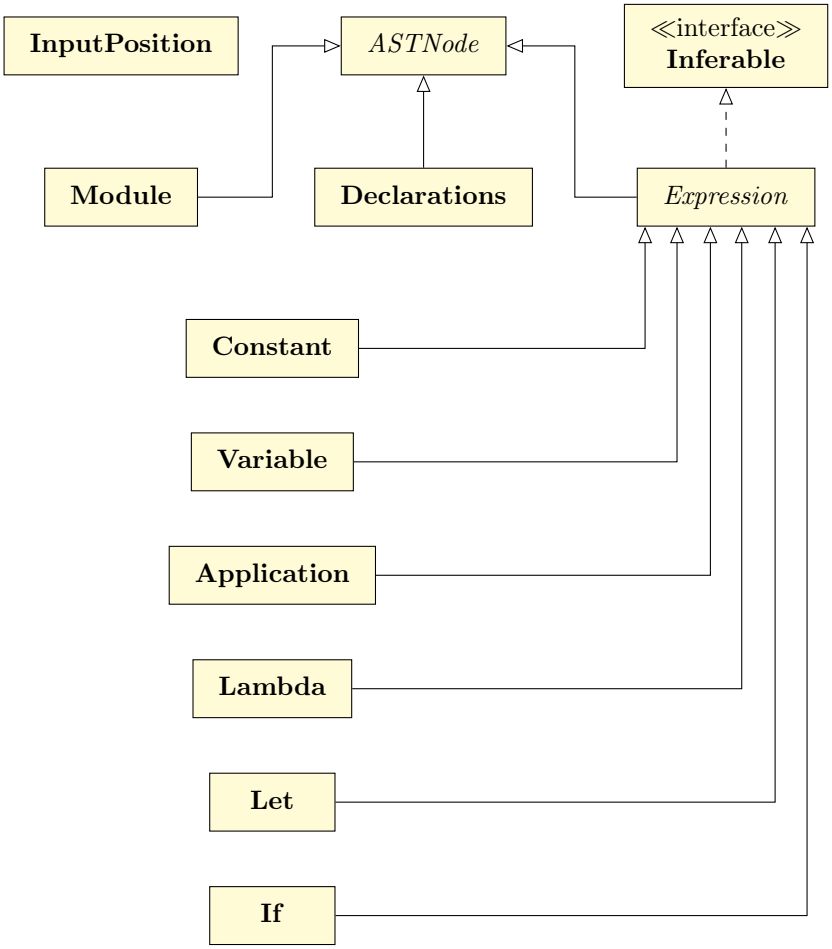

Figura 5.2: Diagramma semplificato delle classi dell'**AST**

```java
public static final class Lambda extends Expression {
    public final String paramId;
    public final Expression expression;

    public Lambda(InputPosition inputPosition, String paramId, ASTNode expression) {
        super(inputPosition); // ASTNode constructor
        this.paramId = paramId;
        this.expression = (Expression) expression;
    }

    @Override // from Inferable interface
    public Utils.Tuple<Substitution, Type> infer(Context ctx) { ... }

    @Override // from Expression abstract class
    protected void propagateSubstitution(Substitution substitution) { ... }

    @Override // from ASTNode abstract class
    public <T> T accept(ASTVisitor<? extends T> visitor) {
        return visitor.visitLambda(this);
    }
}
```

Codice 5.3: Esempio di classe della gerarchia dell'**AST**

5.2.2 AST builder

`ANTLR` offre due diversi approcci per analizzare l'albero di parsing:

- *listener*: i metodi implementati per l'analisi dei nodi sono chiamati seguendo la ricerca in profondità (*DFS, depth-first search*);
- *visitor*: i nodi possono essere visitati arbitrariamente tramite il metodo `visit`, senza seguire un ordine specifico; non tutti i metodi devono essere implementati ed è possibile ignorare dei nodi o ri-visitarli.

La seconda opzione è indubbiamente più versatile, e può essere per esempio anche utilizzata qualora si desideri evitare completamente la costruzione di un **AST** e chiamare i metodi di visita per eseguire il codice (e.g. linguaggi interpretati).
Nel caso di **Funx-jt** la scelta ricade appunto sul *visitor* in virtù della probabilità di dover "saltare" alcuni nodi e ottenere direttamente le informazioni interne.
Durante la compilazione del progetto `Java`, `Gradle` è configurato per generare le seguenti classi nel package `com.github.massimopavoni.funx.jt.parser`:

- `FunxLexer`: *lexer*;
- `FunxParser`: *parser*;
- `FunxParserVisitor`: interfaccia generica per differenti implementazioni del *visitor* (estende l'interfaccia `ParseTreeVisitor` di `ANTLR`);
- `FunxParserBaseVisitor`: classe predefinita che percorre l'intero albero di parsing (estende la classe astratta `AbstractParseTreeVisitor` per ereditare alcuni metodi standard come `visitChildren`).

La classe `ASTBuilder` estende a sua volta `FunxParserBaseVisitor` e sovrascrive quasi tutti i metodi ereditati; in particolare, `ASTBuilder` effettua la composizione degli schemi di tipo definiti dall'utente (sezione 5.3.1) e l'eliminazione dello zucchero sintattico mediante l'uso di alcune proprietà e metodi ausiliari.

Inoltre, all'interno del package `com.github.massimopavoni.funx.jt.ast` viene definita una classe enumeratore, `PreludeFunction`, contenente le funzioni di libreria standard, con i relativi simboli e gli schemi di tipo corrispondenti.

```
1  public enum PreludeFunction {
2     COMPOSE(".", "compose",
3        new Scheme(Set.of(0L, 1L, 2L),
4           arrowOf(arrowOf(ZERO, ONE), arrowOf(TWO, ZERO), TWO, ONE)), false);
5
6     public final String symbol;
7     public final String id;
8     public final Scheme scheme;
9     public final boolean nativeJava;
10
11    PreludeFunction(String symbol, String id,
12       Scheme scheme, boolean nativeJava) { ... }
13
14    public static PreludeFunction fromSymbol(String symbol) {
15       return Utils.enumFromField(PreludeFunction.class,
16          f -> f.symbol.equals(symbol));
17    }
18 }
```

Codice 5.4: Parte del codice di `PreludeFunction`

```
private ASTNode createLambdaChain(
      InputPosition position, Deque<String> params, ASTNode expression) {
   if (params.size() == 1)
      return new Expression.Lambda(
         position,
         params.getFirst(),
         expression);
   return new Expression.Lambda(
      position,
      params.pop(),
      createLambdaChain(position, params, expression));
}

private ASTNode binarySymbolApplication(
      InputPosition position, String symbol, ASTNode left, ASTNode right) {
   return new Expression.Application(
      position,
      new Expression.Application(
            position,
            new Expression.Variable(
                  position,
                  PreludeFunction.fromSymbol(symbol).id),
            left),
      right);
}
```

Codice 5.5: Metodi per astrazioni annidate e operatori simbolici binari

```
public class ASTBuilder extends FunxParserBaseVisitor<ASTNode> {
   @Override
   public ASTNode visitAppExpression(FunxParser.AppExpressionContext ctx) {
      return new Expression.Application(getInputPosition(ctx),
         visit(ctx.expression(0)),
         visit(ctx.expression(1)));
   }

   @Override
   public ASTNode visitComposeExpression(FunxParser.ComposeExpressionContext ctx) {
      return binarySymbolApplication(getInputPosition(ctx),
         Utils.fromLexerToken(ctx.bop.getType()),
         visit(ctx.expression(0)),
         visit(ctx.expression(1)));
   }

   @Override
   public ASTNode visitOrExpression(FunxParser.OrExpressionContext ctx) {
      // transform logical disjunction into if statement for lazy behavior
      return new Expression.If(getInputPosition(ctx),
         visit(ctx.expression(0)),
         new Expression.Constant(InputPosition.UNKNOWN, true),
         visit(ctx.expression(1)));
   }

   @Override
   public ASTNode visitLambda(FunxParser.LambdaContext ctx) {
      // lambda params are syntactic sugar for a lambda chain
      return createLambdaChain(getInputPosition(ctx),
         ctx.lambdaParams().VARID().stream().map(ParseTree::getText)
               .collect(Collectors.toCollection(ArrayDeque::new)),
            visit(ctx.statement()));
   }
}
```

Codice 5.6: Alcuni metodi `visit` di `ASTBuilder`

Il codice del compilatore include una classe astratta `ASTVisitor` atta a permettere l'analisi dell'**AST** in modo simile ai *visitor* di `ANTLR`. Il Codice 5.7 e la Figura 5.3 mostrano un **AST** come verrebbe creato da `ASTBuilder`; la *CLI* sviluppata fornisce delle opzioni per la visualizzazione dell'albero, generata utilizzando il software `Graphviz`[5] con un *visitor* (`GraphvizBuilder`) che scrive una stringa nel linguaggio `DOT`.

```
1 tautology : Bool -> Bool
2 tautology x = x || !!x
```

Codice 5.7: Programma in **Funx**

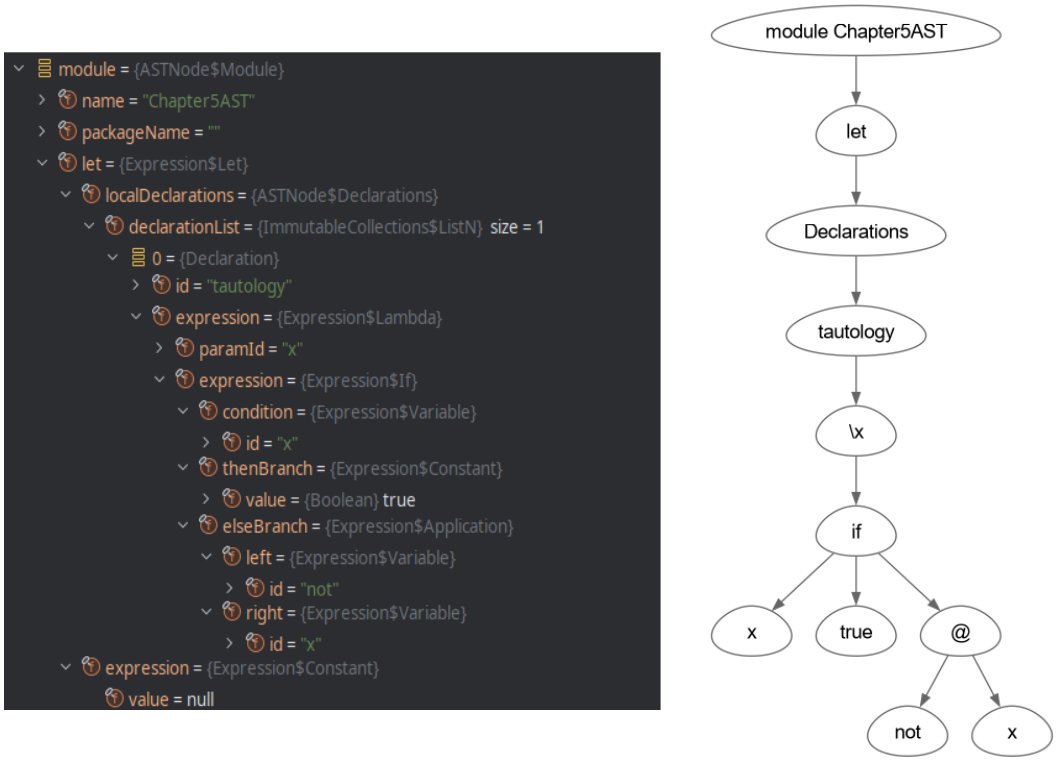

Figura 5.3: Oggetto **AST** in *debug* e visualizzazione dell'albero

[5]Graph Visualization Software (`https://graphviz.org`)

5.3 Motore inferenziale

I sistemi di tipo e l'inferenza descritti nel Capitolo 3 costituiscono una parte imprescindibile del software, data l'incapacità di simulare il polimorfismo parametrico del *sistema HM* lasciando al compilatore `Java` il compito d'inferire e controllare i tipi generici dei metodi (o classi). In tal caso, infatti, la compilazione fallirebbe a causa di tipi "troppo generici" e *type casting* non permesso in maniera implicita (e.g. non si è in grado di conciliare un tipo generico con una lambda espressione).

```
Impossible.java:9: error: method apply in interface Function<T,R> cannot be applied to given types;
    id().apply(x -> y -> x || y).apply(true).apply(false);
        ^
  required: Object
  found:    (x)->(y)->x || y
  reason: argument mismatch; Object is not a functional interface
  where T,R are type-variables:
    T extends Object declared in interface Function
    R extends Object declared in interface Function
1 error
error: compilation failed
```

Figura 5.4: Esempio di errore in `Java` dovuto a tipi generici

La linea di codice citata in Figura 5.4 non è intrinsecamente errata, ma non può essere compilata in mancanza d'informazioni riguardanti il parametro di tipo generico della funzione `id` (con tipo di ritorno `Function<T, T>`): il problema da risolvere è quindi conoscere il tipo dell'espressione in input alla prima chiamata del metodo `apply`.

Il motore inferenziale è la parte del compilatore che si occupa di stabilire i tipi delle espressioni seguendo le regole d'inferenza e i passi dell'*algoritmo* $\mathcal{W}$ [Gra06] descritti nella sezione 3.2; la fase d'inferenza avviene subito dopo parsing e costruzione dell'**AST**, anticipando la generazione del codice `Java`.

5.3.1 Sistema HM

Seguendo la grammatica del sistema di tipo di **Funx** e similmente alla gerarchia per l'albero sintattico astratto, l'implementazione del *sistema HM* presenta le seguenti classi e sottoclassi:

- `Scheme`: schemi di tipo;
- `Type`: classe astratta per i monotipi;
- `Error`: tipo errore per la continuazione dell'inferenza;
- `Boring`: tipo vuoto per costanti con valore `null`;
- `Variable`: variabili di tipo;
- `FunctionApplication`: applicazione di funzione di tipo.

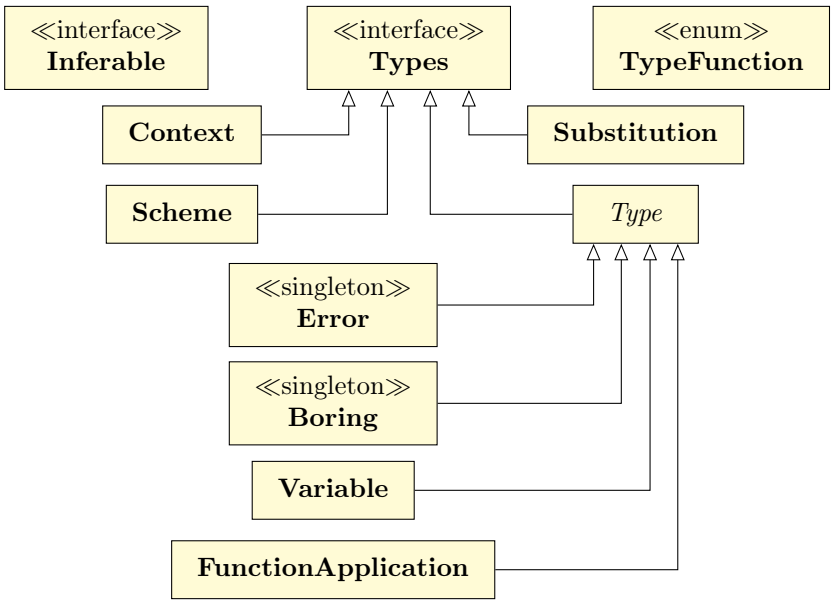

Figura 5.5: Diagramma semplificato delle classi del *sistema HM*

```java
public static final class Variable extends Type {
   public final long id;

   public Variable(long id) { this.id = id; }

   public static String toString(long id) {
      return id < 26 ? Character.toString((char) ('a' + id)) : "t" + id;
   }

   public static String toFancyString(long id) {
      return id < 24 ? Character.toString((char) (945 + id)) : "t" + id;
   }

   @Override
   public Set<Long> freeVariables() { ... }

   @Override
   public Type applySubstitution(Substitution substitution) { ... }
}
```

Codice 5.8: Esempio di sottoclasse di `Type`

Altre classi e interfacce di supporto sono fondamentali per la definizione dei comportamenti precedentemente descritti:

- `Types`: interfaccia per operazioni comuni negli oggetti che agiscono a stretto contatto con i tipi; specifica i metodi per il calcolo delle variabili libere e l'applicazione di una sostituzione;
- `Inferable`: interfaccia già visibile in Figura 5.2 e che identifica i nodi dell'**AST** che implementano un metodo d'inferenza (i.e. le sottoclassi di `Expression`);
- `TypeFunction`: enumerazione delle funzioni di tipo disponibili;
- `Substitution`: lista di sostituzioni da variabili di tipo a monotipi;
- `Context`: contesto di inferenza con vincoli tra variabili e schemi di tipo.

```java
package com.github.massimopavoni.funx.jt.ast.node;

public sealed interface Inferable permits Expression {
   Utils.Tuple<Substitution, Type> infer(Context ctx);
}

// -----------------

package com.github.massimopavoni.funx.jt.ast.typesystem;

sealed interface Types<T extends Types<T>>
      permits Type, Scheme, Substitution, Context {
   Set<Long> freeVariables();

   T applySubstitution(Substitution substitution);
}
```

Codice 5.9: Interfacce utili nel *sistema HM*

```java
public final class Context implements Types<Context> {
   private final Map<String, Scheme> environment;

   public Scheme bindingOf(String variable) {
      return environment.get(variable);
   }

   @Override
   public Set<Long> freeVariables() {
      return environment.values().stream().flatMap(s -> s.freeVariables().stream())
         .collect(ImmutableSet.toImmutableSet());
   }

   @Override
   public Context applySubstitution(Substitution substitution) {
      Context newCtx = new Context(this);
      newCtx.environment.replaceAll((v, s) -> s.applySubstitution(substitution));
      return newCtx;
   }
}

public final class Substitution implements Types<Substitution> {
   public static final Substitution EMPTY = new Substitution();
   private final Map<Long, Type> variableTypes;

   public Type substituteOf(Long variable) {
      return variableTypes.get(variable);
   }

   @Override
   public Set<Long> freeVariables() {
      // similar to Context
   }

   @Override
   public Substitution applySubstitution(Substitution substitution) {
      return new Substitution(variableTypes.entrySet().stream()
         .collect(Collectors.toMap(Map.Entry::getKey,
            e -> e.getValue().applySubstitution(substitution))));
   }
}
```

Codice 5.10: Altre classi del *sistema HM*

5.3.2 Inferenza su espressioni

La classe `InferenceEngine` contiene solamente proprietà e metodi statici, e non adotta lo stesso procedimento di esplorazione dell'albero sintattico delle classi figlie di `ASTVisitor` (visualizzazione e traduzione in `Java`): il cammino attraverso l'**AST** inizia con un metodo di avvio per chiamate ricorsive sulle funzioni d'inferenza di ciascun nodo, a partire delle dichiarazioni globali di un modulo.

Come già accennato nella sezione 3.2, i metodi sono molto simili all'*algoritmo* $\mathcal{W}$ mostrato, e le classi ausiliarie appena descritte facilitano lo sviluppo delle funzioni d'inferenza seguendo tale modello da vicino.
Ciò nonostante, si vuol descrivere brevemente l'implementazione del metodo `infer` della classe `Let`, in quanto di particolare interesse per le modifiche apportate al fine di gestire opportunamente la ricorsione e la generalizzazione dei tipi delle funzioni.
L'inferenza per `let`, presentata nel Codice 5.11, si divide in 4 fasi:

- linee 3-21, inizializzazione del contesto:
  - creazione di una mappa per le posizioni delle dichiarazioni;
  - copia del contesto originale;
  - controllo di dichiarazioni duplicate;
  - aggiornamento del contesto con schemi di tipo temporanei;
- linee 23-43, inferenza sulle dichiarazioni locali:
  - inizializzazione della sostituzione;
  - inferenza per ogni dichiarazione;
  - unificazione tra tipi noti (compresi quelli temporanei) e tipi inferiti;
  - progressiva composizione e applicazione delle sostituzioni al contesto;
- linee 45-53, generalizzazione dei tipi:
  - propagazione delle sostituzioni e chiamata alla funzione `generalize`;
  - controllo del tipo definito dallo sviluppatore;
  - aggiornamento del contesto;
- linee 55-60, inferenza sull'espressione principale:
  - chiamata ricorsiva;
  - composizione finale delle sostituzioni;
  - propagazione delle sostituzioni e assegnamento del tipo per il nodo `let`.

```java
@Override
public Utils.Tuple<Substitution, Type> infer(Context ctx) {
   // check for duplicate declarations within the same let
   Map<String, InputPosition> declarationPositions = new HashMap<>();

   Context newCtx = new Context(ctx);

   for (Declaration decl : localDeclarations.declarationList) {
      if (declarationPositions.containsKey(decl.id))
         InferenceEngine.reportError(decl.inputPosition,
            String.format("variable '%s' already declared at %s",
               decl.id, declarationPositions.get(decl.id)));
      else {
         declarationPositions.put(decl.id, decl.inputPosition);

         // while also updating the context with placeholder schemes,
         // so that self and mutual recursion can be handled
         newCtx.bind(decl.id,
            new Scheme(Collections.emptySet(), InferenceEngine.newTypeVariable()));
      }
   }

   // proceed to infer all local declarations
   Substitution subst = Substitution.EMPTY;

   for (Declaration decl : localDeclarations.declarationList) {
      Utils.Tuple<Substitution, Type> declInference = decl.expression.infer(newCtx);
      try {
         // unifying types of known bindings,
         // gradually composing substitutions and updating context
         subst = subst.compose(declInference.fst())
            .compose(newCtx.bindingOf(decl.id).type
               .applySubstitution(subst)
               .unify(declInference.snd())); // could throw TypeException

         decl.expression.type = declInference.snd().applySubstitution(subst);

         newCtx = newCtx.applySubstitution(subst);
      } catch (TypeException e) {
         InferenceEngine.reportError(decl.inputPosition, e.getMessage());
         decl.expression.type = Type.Error.INSTANCE;
      }
   }

   // finally generalize all types and check against actual user-defined schemes
   for (Declaration decl : localDeclarations.declarationList) {
      decl.expression.propagateSubstitution(subst);

      Scheme expectedScheme = decl.expression.type.generalize(newCtx);
      decl.checkScheme(expectedScheme);

      newCtx.bind(decl.id, decl.scheme());
   }

   // now it's possible to infer the expression type
   Utils.Tuple<Substitution, Type> exprInference = expression.infer(newCtx);

   subst = subst.compose(exprInference.fst());
   type = exprInference.snd().applySubstitution(subst);
   expression.propagateSubstitution(subst);

   return new Utils.Tuple<>(subst, type);
}
```

Codice 5.11: Metodo di inferenza per espressioni `let`

```
1  main = countdown 1825
2
3  countdown : Int -> Int
4  countdown = until (equalsEquals 0) (flip subtract 1)
5
6  until p f = until1
7     with
8        until1 x = if p x then x else until1 (f x) fi
9     out
```

Codice 5.12: Esempio di inferenza

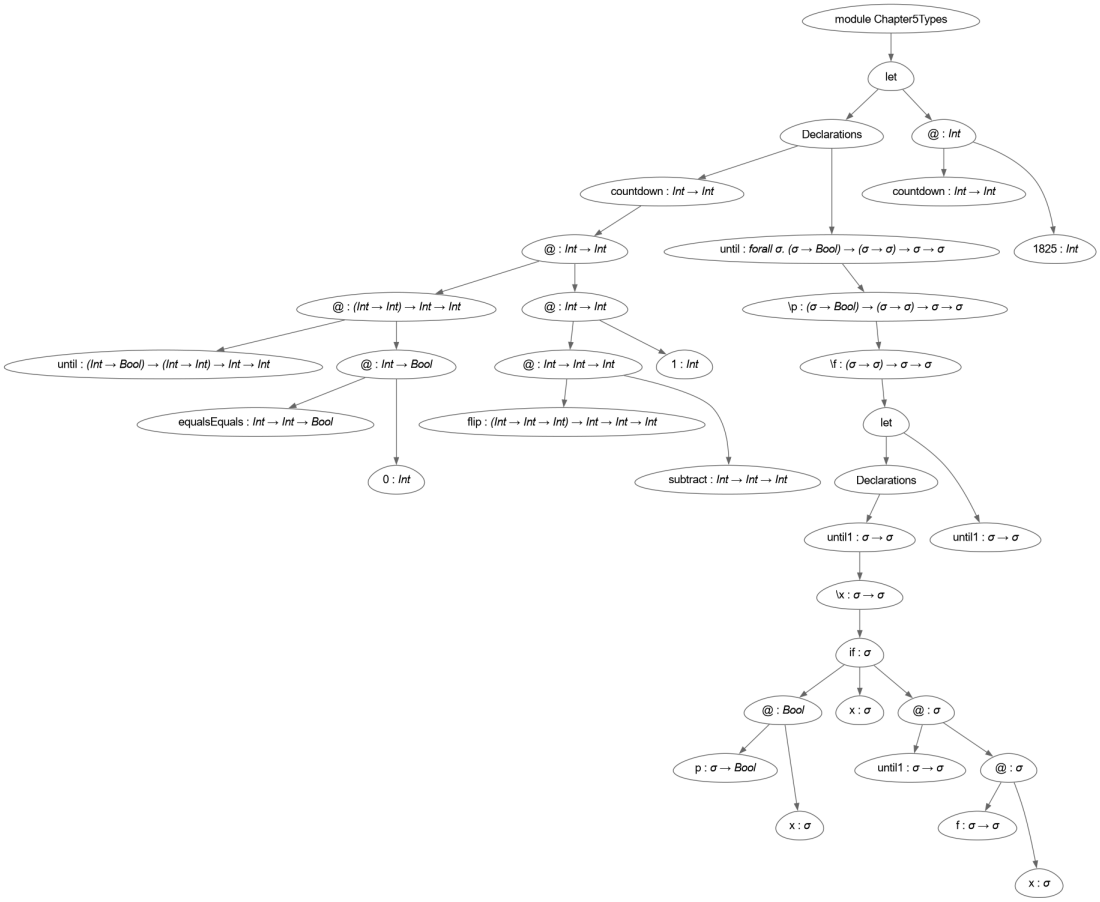

Figura 5.6: **AST** con annotazioni di tipo

5.4 Traduzione in Java

La sottoclasse più importante di `ASTVisitor` è `JavaTranspiler`, il *visitor* che si occupa dell'ultimo stadio di compilazione, la traduzione dell'**AST** in codice `Java`: come per `GraphvizBuilder`, la classe compone una stringa che rappresenta il programma `Java` corrispondente al codice **Funx** sorgente.

Avendo già illustrato alcuni esempi di traduzione nel Capitolo 4, in questa sezione si discuteranno le scelte e i compromessi nel processo di traduzione, e il modo in cui le limitazioni di `Java` possano essere talvolta aggirate "piegando" le regole.

5.4.1 Membri statici

Il paradigma dichiarativo dei linguaggi funzionali è ben diverso dalla programmazione a oggetti di molti altri linguaggi rinomati, motivo per cui la scelta di tradurre ogni programma **Funx** in un'unica classe statica è vista come semplice soluzione per evitare complicanze e *overhead* per la creazione di oggetti in aggiunta alle `Function`.

Ogni funzione definita diviene perciò una proprietà statica della classe in caso di monotipi (sezione 5.4.3) o un metodo statico con parametri di tipo in caso di politipi (sezione 5.4.4). Fanno eccezione le funzioni appartenenti a espressioni `let` annidate: essendo classi anonime, queste creano un unico oggetto con proprietà e metodi privati (accessibili solamente al metodo pubblico `eval` della classe `Let`).

La traduzione inizia con *import* statici per la libreria standard e un costruttore privato.

```
// append package, imports, class declaration and constructor
builder.append(module.packageName.isEmpty()
      ? ""
      : String.format("package %s;%n", module.packageName.toLowerCase()))
   .append("\n\nimport ").append(Function.class.getName())
   .append(";\n\nimport ").append(JavaPrelude.class.getName())
   .append(";\n\nimport ").append(FunxPrelude.class.getName())
   .append(";\n\nimport static ").append(JavaPrelude.class.getName())
   .append(".*;\nimport static ").append(FunxPrelude.class.getName())
   .append(".*;\n\npublic class ").append(module.name).append(" {\n")
   .append("private ").append(module.name)
   .append("() {\n// private constructor to prevent instantiation\n}\n\n");
```

Codice 5.13: Prime aggiunte alla stringa `Java`

```
import java.util.function.Function;

import com.github.massimopavoni.funx.lib.JavaPrelude;

import com.github.massimopavoni.funx.lib.FunxPrelude;

import static com.github.massimopavoni.funx.lib.JavaPrelude.*;
import static com.github.massimopavoni.funx.lib.FunxPrelude.*;

public class Chapter5Header {
   private Chapter5Header() {
      // private constructor to prevent instantiation
   }

   // ...
}
```

Codice 5.14: Corrispondente codice `Java` generato

5.4.2 Stack dei contesti

Durante l'esplorazione dell'**AST** è necessario tenere traccia dello *scope*, il contesto, del quale le dichiarazioni o i parametri lambda fanno parte: la soluzione adottata prevede l'uso di uno *stack* contenente i contesti attivi, rappresentati a loro volta da mappe (le variabili dichiarate sono le chiavi, le relative informazioni sono i valori).

Lo *stack* dei contesti è in realtà implementato attraverso una lista, per via della necessità d'iterare su tutti i contesti e trovare una variabile richiesta (*lookup*).

```
1  private final List<Map<String, Utils.Tuple<Scheme, String>>>
2      scopes = new ArrayList<>();
3
4  private void addToScope(List<Declaration> declarations, String scope) {
5      scopes.addFirst(declarations.stream()
6          .collect(ImmutableMap.toImmutableMap(
7              decl -> decl.id,
8              decl -> new Utils.Tuple<>(decl.scheme(), scope))));
9  }
10
11 private void addToScope(String id, Scheme scheme, String scope) {
12     scopes.addFirst(Collections.singletonMap(id, new Utils.Tuple<>(scheme, scope)));
13 }
```

Codice 5.15: Struttura e metodi per lo *scope*

```
1  @Override
2  public Void visitModule(ASTNode.Module module) {
3      // add Prelude functions to the scope
4      scopes.addFirst(Arrays.stream(PreludeFunction.values())
5          .collect(ImmutableMap.toImmutableMap(
6              pf -> pf.id,
7              pf -> new Utils.Tuple<>(pf.scheme, pf.nativeJava
8                  ? JavaPrelude.class.getSimpleName()
9                  : FunxPrelude.class.getSimpleName()))));
10     // add module functions to the scope
11     addToScope(module.let.localDeclarations.declarationList, module.name);
12     // ...
13     scopes.removeFirst();
14     return null;
15 }
16
17 @Override
18 public Void visitLambda(Expression.Lambda lambda) {
19     // add lambda parameter to the scope
20     addToScope(lambda.paramId, new Scheme(Collections.emptySet(),
21         ((Type.FunctionApplication) lambda.type()).arguments.getFirst()), null);
22     // ...
23     scopes.removeFirst();
24     return null;
25 }
26
27 @Override
28 public Void visitLet(Expression.Let let) {
29     currentLevel++;
30     // add let local declarations to the scope
31     addToScope(let.localDeclarations.declarationList, "this");
32     // ...
33     // restore previous scope state and level
34     scopes.removeFirst();
35     currentLevel--;
36     return null;
37 }
```

Codice 5.16: Aggiunta di contesto per modulo, lambda astrazioni e `let`

Nei Codici 5.15 e 5.16 sono mostrate:

- la lista dei contesti e le funzioni per la loro gestione: le tuple assegnate ai nomi delle dichiarazioni di uno *scope* conservano lo schema di tipo e il nome della classe in cui la variabile è definita (`JavaPrelude`, `FunxPrelude`, nome del modulo creato, `null` per parametri lambda e `this` per i `let`);
- l'aggiunta dei contesti per `Module`, `Lambda` e `Let`, rispettivamente: funzioni di libreria e dichiarazioni del modulo stesso, parametro lambda, dichiarazioni locali con l'incremento del livello di annidamento.

5.4.3 Dichiarazioni monomorfe

La possibilità in **Funx** di utilizzare funzioni ricorsive (e/o mutuamente ricorsive) e la volontà di evitare la traduzione in metodi quando possibile sono in conflitto a causa d'*Illegal Self Reference* e *Illegal Forward Reference*: tali errori si presentano durante la compilazione del codice `Java` qualora i campi statici che identificano funzioni monomorfe vengano dichiarati e inizializzati nello stesso *statement* (stessa linea).

La dichiarazione delle proprietà deve avvenire prima dell'inizializzazione di altre variabili che ne fanno uso; si potrebbe effettuare un'analisi iniziale dell'**AST** per identificare le dipendenze tra le funzioni (approccio di ordinamento topologico estremamente utile anche per l'inferenza), ma la soluzione adottata è di più semplice implementazione.

Come si può notare nel Codice 5.17 e in alcuni esempi già presentati in precedenza, si effettua la dichiarazione di ogni campo, pubblico e statico per le dichiarazioni globali, privato per quelle locali, e solo successivamente si inizializzano rispettivamente con blocco statico e metodo `eval`.

```
1  public class Chapter5Monomorphic {
2     private Chapter5Monomorphic() {
3        // private constructor to prevent instantiation
4     }
5
6     public static void main(String[] args) {
7        System.out.println(add.apply(add.apply(fun1).apply(fun2)).apply(letFun));
8     }
9
10    public static Long fun1;
11    public static Long fun2;
12    public static Long letFun;
13
14    static {
15       fun1 = 1L;
16       fun2 = 2L;
17       letFun =
18          (new Let<>() {
19             private Long a;
20             private Long b;
21
22             @Override
23             public Long _eval() {
24                a = 3L;
25                b = 4L;
26                return add.apply(a).apply(b);
27             }
28          })._eval();
29    }
30 }
```

Codice 5.17: Esempio di traduzione per funzioni monomorfe

Poiché potrebbero essere presenti diversi `let` annidati, è necessario tenere traccia delle espressioni corpo delle dichiarazioni monomorfe in modo da poterle inizializzare al momento corretto, dopo aver tradotto ulteriori classi interne.
La procedura di traduzione di dichiarazioni monomorfe si compone delle seguenti fasi:

- definizione di uno *stack* contenente mappe tra nomi delle dichiarazioni e nodi espressione corrispondenti;
- inserimento di una nuova mappa per il livello corrente di annidamento (modulo o espressione `let`);
- dichiarazione delle variabili e aggiunta delle espressioni monomorfe alla mappa corrente (potrebbero essere aggiunti nuovi livelli prima di poterne "riempire" uno);
- creazione del blocco statico (o metodo `eval` per le espressioni `let`) con l'inizializzazione delle funzioni monomorfe: in questa fase finale torna utile la versatilità del *visitor pattern* per posticipare la traduzione delle espressioni.

```
1  private final Deque<Map<String, Expression>>
2     monomorphicDeclarationsQueue = new ArrayDeque<>();
3
4  @Override
5  public Void visitLet(Expression.Let let) {
6     currentLevel++;
7     // ...
8     // use a new anonymous class for the let expression
9     // and push a new monomorphic let declarations map
10    builder.append("(new ")
11          .append(JavaPrelude.Let.class.getSimpleName()).append("<>() {\n");
12    monomorphicDeclarationsQueue.push(new LinkedHashMap<>());
13    visit(let.localDeclarations);
14    builder.append("""
15                @Override
16                public\s""")
17          .append(typeStringOf(let.expression.type()))
18          .append("\n_eval() {\n");
19    // if there are any monomorphic declarations, initialize them in the _eval method,
20    // then pop the map either way
21    if (!monomorphicDeclarationsQueue.getFirst().isEmpty())
22       monomorphicDeclarationsQueue.getFirst().forEach((id, expression) -> {
23          builder.append(id).append(" = ");
24          visit(expression); // deferred expression visit
25          appendSemiColon();
26          appendNewline();
27       });
28    monomorphicDeclarationsQueue.pop();
29    // ...
30    currentLevel--;
31    return null;
32 }
```

Codice 5.18: Traduzione di funzioni monomorfe in `let`

```
@Override
public Void visitDeclaration(Declaration declaration) {
   // top level declarations should be static and public,
   // while let local declarations should be private to the anonymous class
   builder.append(currentLevel == 0 ? "public static " : "private ");
   String scheme = schemeStringOf(declaration.scheme());
   if (declaration.scheme().variables.isEmpty()) {
      // defer initialization of monomorphic declarations
      builder.append(scheme).append(" ").append(declaration.id);
      appendSemiColon();
      monomorphicDeclarationsQueue
         .getFirst().put(declaration.id, declaration.expression);
   } else {
      // initialize polymorphic declarations immediately (as methods with generics)
      builder.append(scheme)
            .append(" ").append(declaration.id).append("() {\nreturn ");
      visit(declaration.expression);
      appendSemiColon();
      appendCloseBrace();
   }
   appendNewline();
   return null;
}
```

Codice 5.19: Metodo `visit` per le dichiarazioni

5.4.4 Istanziazione di funzioni polimorfe

Dal momento che il contesto può cambiare con l'introduzione di parametri lambda oltre che con espressioni `let`, il livello di annidamento di quest'ultime è segnalato da una variabile secondaria. Quest'ultima è utilizzata durante il *lookup* delle variabili per verificare se è possibile istanziare una funzione polimorfa in modo idiomatico per `Java`.

Tale variabile (`currentLevel`) è stata introdotta nei precedenti estratti di codice del `JavaTranspiler`, ma se ne fa maggior uso nel caso di funzioni polimorfe per specificare i parametri di tipo quando possibile.

Visitando un nodo `Variable` dell'albero sintattico astratto si incontrano tre casi:

1. funzione monomorfa o parametro lambda (tutte le variabili introdotte da una lambda astrazione possiedono un monotipo per via del polimorfismo di rango 1);

2. funzione polimorfa proveniente da uno *scope* compreso tra quello globale e il livello di annidamento corrente; opzione rappresentante uno dei casi limite già menzionati, causato dall'impossibilità di fare riferimento esplicito a un membro di una classe anonima esterna; la soluzione è utilizzare il metodo definito in `JavaPrelude` per istanziare la funzione parametrica tramite un *cast*;

3. funzione polimorfa proveniente dal livello di annidamento attivo, livello globale (*scope* con stesso nome del modulo) o libreria standard; in questa alternativa è possibile unificare un'istanza dello schema di tipo con il tipo della variabile nell'espressione considerata e quindi parametrizzare la chiamata.

```java
@Override
public Void visitVariable(Expression.Variable variable) {
   // firstly, find the variable in the scopes
   int i;
   Utils.Tuple@<Scheme, String@> variableScheme = null;
   for (i = 0; i <scopes.size(); i++)
      if ((variableScheme = scopes.get(i).get(variable.id)) != null)
         break;

   // cannot have a null tuple, since type inference would have failed before this
   if (Objects.requireNonNull(variableScheme).fst().variables.isEmpty())
      // 1 -> lambda param or monomorphic declaration
      builder.append(variable.id);

   else if (variableScheme.snd().equals("this") && i >0 && i <currentLevel)
      // 2 -> polymorphic declaration from an intermediate let scope
      // needs the worst: an unchecked cast
      builder.append(JavaPrelude.class.getSimpleName()).append(".<")
         .append(typeStringOf(variable.type())).append(">_instantiationCast(")
         .append(variable.id)
         .append("())");

   else {
      // 3 -> polymorphic declaration from Prelude,
      // top level or same let scope can properly use generics
      builder.append(variableScheme.snd()).append(".<");
      try {
         // to do so we need to instantiate the scheme and find the substitution
         Utils.Tuple@<Substitution, Type@> instantiation =
            variableScheme.fst().instantiate();

         Substitution subst = instantiation.fst()
            .applySubstitution(instantiation.snd().unify(variable.type()));

         // to then apply to the sorted variables
         builder.append(variableScheme.fst().sortedVariables.stream()
               .map(ov @@-> subst.variables().contains(ov)
                  ? typeStringOf(subst.substituteOf(ov))
                  : Type.Variable.toString(ov))
               .collect(Collectors.joining(", ")))
            .append(">").append(variable.id).append("()");
      } catch (TypeException e) {
         // should never happen
         throw new InferenceException(e.getMessage());
      }
   }
   return null;
}
```

Codice 5.20: Metodo `visit` per le variabili

```java
// Cast method for polymorphic functions instantiation
@SuppressWarnings("rawtypes, unchecked")
public static @<T extends Function@> T _instantiationCast(Function f) {
   return (T) f;
}
```

Codice 5.21: Metodo di istanziazione con *cast*

Nei Codici 5.22 e 5.23 si può osservare come il procedimento descritto produca variabili usate semplicemente, istanziate tramite *cast* o con parametri di tipo.

```
multipleIds x = let
      id1 = id
   in let
         id2 = id1
      in id2 x
```

Codice 5.22: `Let` annidati e differente uso di variabili polimorfe

```
public class Chapter5Nested {
   private Chapter5Nested() {
      // private constructor to prevent instantiation
   }

   public static <h> Function<h, h> multipleIds() {
      return (x ->
         (new Let<>() {
            private <d> Function<d, d> id1() {
               return FunxPrelude.<d>id(); // 3
            }

            @Override
            public h _eval() {
               return (new Let<>() {
                  private <f> Function<f, f> id2() {
                     return JavaPrelude
                        .<Function<f, f>>_instantiationCast(id1()); // 2
                  }

                  @Override
                  public h _eval() {
                     return this.<h>id2().apply(x); // 3 and 1
                  }
               })._eval();
            }
         })._eval());
   }
}
```

Codice 5.23: Corrispondente traduzione in `Java`

5.4.5 Type casting "selvaggio"

L'ultima particolarità della traduzione riguarda una stranezza occasionalmente necessaria quando le espressioni parte di un'applicazione richiedono un ulteriore *cast*: questi *cast* non sempre sono davvero necessari ma prevederne l'esigenza a priori è difficile, ed è il motivo per cui sono stati battezzati "selvaggi" (*wild cast*).

L'implementazione dei *wild cast* è data da una *flag* booleana abilitata nel nodo `Application` qualora le espressioni coinvolte siano oggetti `Lambda` o `Let`, mentre il *cast* è applicato nei metodi `visit` interessati.

```
@Override
public Void visitApplication(Expression.Application application) {
   // left and right expressions necessitate a wild cast
   // if they are lambda or let expressions
   if (application.left instanceof Expression.Lambda
         || application.left instanceof Expression.Let)
      wildCast = true;
   visit(application.left);

   builder.append(".apply(");
   if (application.right instanceof Expression.Lambda
         || application.right instanceof Expression.Let)
      wildCast = true;
   visit(application.right);
   builder.append(")");
   return null;
}
```

Codice 5.24: Metodo `visit` per le applicazioni di funzioni

```
@Override
public Void visitLambda(Expression.Lambda lambda) {
   // ...
   if (wildCast) {
      // wild cast is needed for lambdas in applications
      builder.append("(").append(typeStringOf(lambda.type())).append(") ");
      wildCast = false;
   }
   // ...
   return null;
}

@Override
public Void visitLet(Expression.Let let) {
   // ...
   if (wildCast) {
      // wild cast is needed for lets in applications
      builder.append("(").append(typeStringOf(let.type())).append(") ");
      wildCast = false;
   }
   // ...
   return null;
}
```

Codice 5.25: *Wild cast* in espressioni `lambda` e `let`

Nei Codici 5.26 e 5.27 si nota come questo approccio renda la traduzione ancora meno leggibile, ma si può facilmente verificare che la compilazione fallisce se alcuni *cast* sono rimossi (*cast* di destra in `reverseApply` e *cast* di sinistra in `anonymousIds`).

```
reverseApply = flip (let
      apply1 f x = f x
   in apply1)

anonymousIds = (\x -> x) (\x -> x)
```

Codice 5.26: Applicazione tra funzioni, espressioni `let` e `lambda`

```
public class Chapter5Wild {
   private Chapter5Wild() {
      // private constructor to prevent instantiation
   }

   public static <j, k> Function<j, Function<Function<j, k>, k>> reverseApply() {
      return FunxPrelude.<Function<j, k>, j, k>flip()
         .apply(
            (Function<Function<j, k>, Function<j, k>>) // right wild cast
               (new Let<>() {
                  private <h, i> Function<Function<h, i>, Function<h, i>> apply1() {
                     return (f -> (x -> f.apply(x)));
                  }

                  @Override
                  public Function<Function<j, k>, Function<j, k>> _eval() {
                     return this.<j, k>apply1();
                  }
               })._eval());
   }

   public static <n> Function<n, n> anonymousIds() {
      return ((Function<Function<n, n>, Function<n, n>>) x -> x) // left wild cast
         .apply(((Function<n, n>) x -> x)); // right wild cast
   }
}
```

Codice 5.27: Traduzione in `Java` con *cast* "selvaggi"

6. Conclusioni

All'epilogo del lavoro di tesi, è possibile affermare che il progetto è stato portato a termine con un più che discreto successo: il linguaggio funzionale **Funx** è molto semplice ma può essere esteso per aumentarne l'espressività, mentre il compilatore offre spazio per miglioramenti e ottimizzazioni.

Contemporaneamente, come ogni software, il risultato è lontano dall'essere perfetto, e durante le fasi finali di sviluppo e di scrittura di questo documento sono emerse criticità e *bug* nel traduttore che ne inficiano affidabilità ed efficienza in alcuni rari casi, non scoperti in precedenza.

In questo breve capitolo verranno confrontati i risultati con gli obiettivi prefissati, suggerite possibili estensioni per il linguaggio e accennato un uso didattico.

6.1 Obiettivi

Relativamente all'obiettivo di sviluppo del compilatore, l'approccio costruttivo della stringa `Java` costituisce senz'altro una traduzione molto semplice e non eccessivamente intricata: si è stati in grado di utilizzare le *feature* menzionate nel Capitolo 4 in modo appropriato e in accordo con la specifica del piccolo linguaggio funzionale.

Le soluzioni ai problemi di *parsing*, inferenza e generazione del codice si sono rivelate abbastanza agili e corrette da favorire tempi di compilazione ed esecuzione accettabili, ma ovviamente non sufficienti per giustificare la preferenza di **Funx** rispetto ad altri linguaggi maturi. D'altro canto, il progetto non ha mai avuto simili pretese, e da subito è stato chiaro che la scelta della `JVM` come piattaforma di destinazione avrebbe limitato le prestazioni dei programmi.

Per quanto il linguaggio permetta solamente di definire semplici funzioni usando numeri interi e costanti booleane, le minime funzionalità di cui è dotato soddisfano i requisiti prestabiliti, inclusa la possibilità di ricorsione.

6.2 Estensioni del linguaggio

Nel caso in cui si volesse continuare a sviluppare **Funx** e il compilatore annesso, quelle che seguono sono alcune estensioni utili a incrementarne notevolmente l'espressività:

- **tuple**: nuovo tipo analogo alle tuple di `Haskell`, da ideare anche all'interno di `Java` stesso data la mancanza di supporto;
- **liste**: implementazione della corrispondenza tra le liste di `Java` e un nuovo tipo in **Funx**, con il vantaggio di poter utilizzare alcune funzioni native;
- **stringhe**: la gestione di dati testuali si rivela essere potenzialmente tra le più complicate a seconda della traduzione dei singoli caratteri (utilizzare le liste in `Java` si rivelerebbe inefficiente);
- **switch-case**: aggiunta di una struttura di controllo più versatile rispetto all'attuale `if-else`, affine alle *switch expression* di `Java` e i *case-of* di `Haskell`;
- **tipi custom**: l'introduzione di tipi definiti dall'utente è un'altra espansione difficile da trattare, poiché andrebbero probabilmente usate le classi di `Java`, da generare sempre automaticamente;
- **suite di test**: con l'ampliamento del linguaggio, è indispensabile preparare una collezione di test per la verifica della correttezza del compilatore.

6.3 Scopo educativo

L'adozione di un linguaggio simile a **Funx** a scopo educativo offrirebbe opportunità significative per arricchire l'insegnamento della programmazione, soprattutto dove `Java` e altri linguaggi orientati agli oggetti dominano il curriculum. In ambito scolastico e accademico si tende spesso a concentrarsi molto sulla programmazione imperativa anche nelle fasi più avanzate dei corsi, lasciando agli studenti l'approfondimento di altri paradigmi introdotti con minore importanza.

Alcuni concetti fondamentali di ogni linguaggio sono tuttavia più facilmente compresi attraverso la programmazione funzionale: l'uso di **Funx**, assieme alla traduzione diretta in un linguaggio più familiare, sarebbe forse un modo per addolcire la transizione a un paradigma al quale la maggior parte degli studenti non è abituata.

Bibliografia

[Bar91] Henk Barendregt. «Introduction to generalized type systems». In: *Journal of Functional Programming* 1.2 (apr. 1991), pp. 125–154. DOI: 10.1017/s0956796800020025.

[Chu32] Alonzo Church. «A Set of Postulates for the Fundation of Logic». In: *Annals of Mathematics* 33.2 (apr. 1932), pp. 346–366. DOI: 10.2307/1968337.

[Chu33] Alonzo Church. «A Set of Postulates for the Foundation of Logic». In: *Annals of Mathematics* 34.4 (ott. 1933). Second paper, pp. 839–864. DOI: 10.2307/1968702.

[Clé+86] Dominique Clément et al. «A Simple Applicative Language: Mini-ML». In: *Proceedings of the 1986 ACM conference on LISP and functional programming - LFP '86* (8 ago. 1986), pp. 13–27. DOI: 10.1145/319838.319847.

[DM82] Luis Damas e Robin Milner. «Principal type-schemes for functional programs». In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82* (25 gen. 1982), pp. 207–212. DOI: 10.1145/582153.582176.

[Eis15] Richard A. Eisenberg. «System FC, as implemented in GHC». In: (2015). Last updated in 2020. URL: https://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf?raw=true.

[Gra06] Martin Grabmüller. «Algorithm W Step by Step». In: (26 set. 2006). URL: https://github.com/mgrabmueller/AlgorithmW/blob/master/pdf/AlgorithmW.pdf?raw=true.

[Hin69] J. Roger Hindley. «The Principal Type-Scheme of an Object in Combinatory Logic». In: *Transactions of the American Mathematical Society* 146 (dic. 1969), pp. 29–60. DOI: 10.2307/1995158.

[LY98] Oukseh Lee e Kwangkeun Yi. «Proofs about a Folklore Let-Polymorphic Type Inference Algorithm». In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20.4 (1 lug. 1998), pp. 707–723. DOI: 10.1145/291891.291892.

[Mil78] Robin Milner. «A Theory of Type Polymorphism in Programming». In: *Journal of Computer and System Sciences* 17.3 (dic. 1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[Mog91] Eugenio Moggi. «Notions of Computation and Monads». In: *Information and Computation* 93.1 (lug. 1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.

[Par13] Terence J. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 22 gen. 2013. ISBN: 978-1-934356-99-9.

[PF11] Terence J. Parr e Kathleen Fisher. «LL(*): the foundation of the ANTLR parser generator». In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '11* (4 giu. 2011), pp. 425–436. DOI: 10.1145/1993498.1993548.

[PHF14] Terence J. Parr, Sam Harwell e Kathleen Fisher. «Adaptive LL(*) parsing: the power of dynamic analysis». In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications - OOPSLA '14* (15 ott. 2014), pp. 579–598. DOI: 10.1145/2660193.2660202.

[PQ95] Terence J. Parr e Russell W. Quong. «ANTLR: A predicated-LL(k) parser generator». In: *Software: Practice and Experience* 25.7 (lug. 1995), pp. 789–810. DOI: 10.1002/spe.4380250705.

[Wel99] Joe B. Wells. «Typability and type checking in System F are equivalent and undecidable». In: *Annals of Pure and Applied Logic* 98.1-3 (30 giu. 1999), pp. 111–156. DOI: 10.1016/s0168-0072(98)00047-5.

Ringraziamenti

Grazie al Prof. Luca Padovani per avermi fatto scoprire i linguaggi funzionali.
Grazie a tutti i professori che insegnando con passione hanno stimolato la mia curiosità.

Grazie a tutti i colleghi studenti, passati e presenti.
Grazie ad Andrea, Nicola, Matteo e Thomas.
Grazie a Eduardo, Lorenzo e Marco.
Grazie davvero a Riccardo.

Grazie alla mia famiglia, papà, mamma e Davide.
Grazie a nonno, nonna, nonno e nonna, grazie ai cugini, grazie agli zii e alle zie.

Grazie ai miei amici, vicini e lontani.
Grazie a Gaetano, Matteo, Alberto, Endri e Stefano.
Grazie ad Alessandro, Costanza, Matteo, Tiffany, Giuseppe e Giacomo.

Grazie di cuore a Luca, Tommaso, Antonio, Luca, Nicola e Began.

Grazie di tutto ad Anaya.

Mille grazie vanno a ciascuno di voi ché avete formato la mia persona e mi avete fatto compagnia negli ultimi anni, e a un'infinità di altre persone che sono state importanti in un modo o nell'altro.