“Sapienza” Università di Roma
Ingegneria dell’Informazione,
Informatica e Statistica
Dipartimento di Informatica

Linguaggi di Programmazione

Author
Simone Bianco

17 ottobre 2023

Indice

Informazioni e Contatti

Appunti e riassunti personali raccolti in ambito del corso di *Linguaggi di Programmazione* offerto dal corso di laurea in Informatica dell'Università degli Studi di Roma "La Sapienza".

Ulteriori informazioni ed appunti possono essere trovati al seguente link:
https://github.com/Exyss/university-notes. Chiunque si senta libero di segnalare incorrettezze, migliorie o richieste tramite il sistema di Issues fornito da GitHub stesso o contattando in privato l'autore :

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

Gli appunti sono in continuo aggiornamento, pertanto, previa segnalazione, si prega di controllare se le modifiche siano già state apportate nella versione più recente.

Prerequisiti consigliati per lo studio:

Apprendimento del materiale relativo al corso *Algebra*.

1

Struttura e Rappresentazione

1.1 Algebre induttive

Definizione 1: Assiomi di Peano

L'insieme dei numeri naturali $\mathbb{N}$ è definito secondo i seguenti **assiomi di Peano**:

1. $0 \in \mathbb{N}$
2. $n \in \mathbb{N} \implies \text{succ}(n) \in \mathbb{N}$, dove $\text{succ} : \mathbb{N} \to \mathbb{N}$ è la funzione successore
3. $\forall n, m \in \mathbb{N}, \text{succ}(n) = \text{succ}(m) \implies n = m$, ossia succ è iniettiva
4. $\nexists n \in \mathbb{N} \mid \text{succ}(n) = 0$
5. $\forall S \subseteq \mathbb{N} \mid (0 \in S \wedge (n \in S \implies \text{succ}(n) \in S)) \implies S = \mathbb{N}$

Proposizione 1: Numeri naturali di Von Neumann

I numeri naturali di Von Neumann, indicati con $\mathcal{N}$, definiti come:

$$0_{\mathcal{N}} := \{\}$$

$$1_{\mathcal{N}} := \{\,\{\}\,\}$$
$$2_{\mathcal{N}} := \{\,\{\}, \{\{\}\}\,\}$$
$$3_{\mathcal{N}} := \{\,\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\,\}$$
$$\ldots$$

dove $\text{succ}_{\mathcal{N}} : \mathcal{N} \to \mathcal{N} : n \mapsto n \cup \{n\}$, soddisfano gli assiomi di Peano

Dimostrazione.

1. $0_{\mathcal{N}} \in \mathcal{N}$ per definizione stessa di $\mathcal{N}$
2. $n \in \mathcal{N} \implies \mathrm{succ}_{\mathcal{N}}(n) \in \mathcal{N}$ per definizione stessa di $\mathrm{succ}_{\mathcal{N}}$
3. Siano $n, m \in \mathcal{N}$ tali che $n \neq m$. In tal caso, ne segue automaticamente che:

$$n \neq m \implies n \cup \{n\} \neq m \cup \{m\} \iff \mathrm{succ}_{\mathcal{N}}(n) \neq \mathrm{succ}_{\mathcal{N}}(m)$$

   Per contro-nominale, dunque, otteniamo che:

$$\mathrm{succ}_{\mathcal{N}}(n) = \mathrm{succ}_{\mathcal{N}}(m) \implies n = m$$

4. Supponiamo per assurdo che $\exists n \in \mathbb{N} \mid \mathrm{succ}_{\mathcal{N}}(n) = 0_{\mathcal{N}}$. In tal caso, avremmo che:

$$succ(n) = 0_{\mathcal{N}} \iff n \cup \{n\} = 0_{\mathcal{N}} \iff n \cup \{n\} = \{\}$$

   ma ciò risulta assurdo poiché implicherebbe che l'insieme $\{\}$ contenga degli elementi. Di conseguenza, l'unica possibilità è che $\nexists n \in \mathbb{N} \mid \mathrm{succ}_{\mathcal{N}}(n) = 0_{\mathcal{N}}$

5. Supponiamo per assurdo che $\exists S \subseteq \mathcal{N} \mid (0_{\mathcal{N}} \in S \wedge (n \in S \implies \mathrm{succ}_{\mathcal{N}}(n) \in S)) \wedge S \neq \mathcal{N}$. Consideriamo quindi $\mathcal{N} - S = \{n_1, \dots, n_k\}$. Per via del secondo assioma, ogni elemento di $\mathcal{N} - S$ deve avere un proprio successore e un proprio predecessore in $\mathcal{N}$.

   Poiché per ipotesi si ha che $n \in S \implies \mathrm{succ}_{\mathcal{N}}(n) \in S$, ne segue che tutti i predecessori degli elementi in $\mathcal{N} - S$ non possano essere in S, poiché altrimenti tali elementi sarebbero in S. Inoltre, poiché $\mathrm{succ}_{\mathcal{N}}$ è iniettiva, ne segue che i successori degli elementi in $\mathcal{N} - S$ non possano essere in S, poiché esiste già un predecessore in S per ogni elemento in S.

   Di conseguenza, ogni predecessore ed ogni successore degli elementi di $\mathcal{N} - S$ deve essere in $\mathcal{N} - S$ stesso. Consideriamo quindi (per comodità) la seguente catena di successori in $\mathcal{N} - S$:

$$n_1 \to n_2 \to \dots \to n_k \to n_1$$

   Notiamo a questo punto che:

$$\mathrm{succ}_{\mathcal{N}}^k(n_1) = n_1 \implies n_1 \in n_1$$

   contraddicendo gli assiomi insiemistici per cui un insieme non possa essere contenuto in se stesso. Di conseguenza, l'unica possibilità è che $S = \mathcal{N}$

□

Principio 1: Principio di induzione

Sia P una proprietà che vale per $n = 0$. Dato $n \in \mathbb{N}$, se si verifica che la veridicità di P per n implica che P sia vera anche per $n + 1$, allora P vale per tutto $\mathbb{N}$. In simboli, abbiamo che:

$$\forall P \;\; ((P(0) \land (P(n) \implies P(n+1)))) \implies \forall m \in \mathbb{N} \;\; P(m)$$

Osservazione 1

Il quinto assioma di Peano è equivalente al principio di induzione, poiché basta considerare $S \subseteq \mathbb{N}$ come l'insieme degli elementi per cui vale la proprietà desiderata

Osservazione 2

Dato $k \in \mathbb{N}$, il principio di induzione può essere utilizzato per dimostrare che una proprietà P valga $\forall n \in \mathbb{N} \mid n \geq k$. In altre parole, non è necessario che il principio valga per tutti i naturali a partire da 0.

Dimostrazione.

- Definendo una proprietà Q tale che $P(n) = Q(n - k)$, si ha che:

  $$\forall n - k \in \mathbb{N} \;\; Q(n - k) \iff P(n)$$

  dunque applicare il principio di induzione per P partendo da k equivale ad applicare il principio di induzione per Q partendo da 0, rispettando quindi il quinto assioma di Peano

□

Definizione 2: Insieme unità

Definiamo come **insieme unità** l'insieme $\mathbb{1} = \{()\}$, ossia l'insieme composto da una zerupla

Definizione 3: Funzione nullaria

Definiamo una funzione $f : \mathbb{1} \to S$, dunque avente $\mathbb{1}$ come dominio, come **funzione nullaria** (o funzione costante).

Inoltre, per comodità, indichiamo $f(x)$ direttamente con f, poiché $x = ()$

Esempio:

- Data la funzione $\text{zero} : \mathbb{1} \to \mathbb{N} : x \mapsto 0$, indichiamo $\text{zero}(x)$ direttamente come zero

Osservazione 3

Una funzione nullaria è sempre **iniettiva** in quanto esiste un solo elemento nel dominio.

Definizione 4: Segnatura di una funzione

Data una funzione f definiamo $f : D \to C$ come **segnatura di** f dove D è il **dominio di** f e C è il **codominio di** f

Definizione 5: Algebra

Definiamo come **algebra** (o struttura algebrica) una n-upla $(A, \gamma_1, \ldots, \gamma_n)$ dove A è un insieme non vuoto, detto **dominio**, e $\gamma_1, \ldots, \gamma_n$ sono delle operazioni definite su A stesso.

Esempi:

- La coppia $(\mathbb{N}, \text{succ})$ è un'algebra
- La coppia $(\mathbb{N}, \text{zero})$ è un'algebra

Definizione 6: Segnatura di un'algebra

Data un'algebra $(A, \gamma_1, \ldots, \gamma_n)$, definiamo come **segnatura dell'algebra** l'insieme delle segnature delle operazioni definite su essa

Definizione 7: Segnature equivalenti

Date due algebre $(A, \gamma_1, \ldots, \gamma_n)$ e $(B, \delta_1, \ldots, \delta_n)$, definiamo le segnature di tali algebre come **equivalenti** se per ogni operazione γ definita su A esiste un'operazione δ definita su B per cui invertendo B con A all'interno della segnatura di δ si ottiene la segnatura di γ

Esempio:

- Date le due algebre $(\mathbb{N}, \text{zero}, \text{succ})$ e $(\mathcal{N}, \text{zero}_\mathcal{N}, \text{succ}_\mathcal{N})$, le segnature di tali algebre sono equivalenti poiché:
  - La segnatura di $\text{zero} : \mathbb{1} \to \mathbb{N}$ è equivalente alla segnatura di $\text{zero}_\mathcal{N} : \mathbb{1} \to \mathcal{N}$
  - La segnatura di $\text{succ} : \mathbb{N} \to \mathbb{N}$ è equivalente alla segnatura di $\text{succ}_\mathcal{N} : \mathcal{N} \to \mathcal{N}$

Definizione 8: Algebra induttiva e Costruttori

Definiamo l'algebra $(A, \gamma_1, \ldots, \gamma_n)$ come **induttiva** (o **iniziale**) se:

- $\gamma_1, \ldots, \gamma_n$ sono iniettive
- $\forall i \neq j \quad \mathrm{im}(\gamma_i) \cap \mathrm{im}(\gamma_j) = \varnothing$, ossia le immagini delle operazioni sono due a due disgiunte
- $\forall S \subseteq A \quad (\forall i \in [1, n], a_1, \ldots, a_k \in S \quad \gamma_i(a_1, \ldots, a_k) \in S) \implies S = A$, ossia è soddisfatto il principio di induzione per ogni operazione

Inoltre, definiamo $\gamma_1, \ldots, \gamma_n$ come **costruttori di** A.

Esempi:

- L'algebra $(\mathbb{N}, +)$ non è un'algebra induttiva poiché $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ non è iniettiva
- L'algebra $(\mathbb{N}, \mathrm{succ}, \mathrm{zero})$ è un'algebra induttiva poiché:
  - succ risulta essere iniettiva grazie al secondo assioma di Peano, mentre zero risulta essere iniettiva poiché funzione nullaria
  - $\mathrm{im}(\mathrm{succ}) \cap \mathrm{im}(\mathrm{zero}) = (\mathbb{N} - \{0\}) \cap \{0\} = \varnothing$
  - Sia $S \subseteq \mathbb{N}$ tale che $\forall x \in S \quad \mathrm{succ}(x) \in S$ e $\mathrm{zero} \in S$. Preso $x \in \mathbb{N}$, possiamo esprimere x come $x = \mathrm{succ}(\mathrm{succ}(...(\mathrm{zero})))$.

    Di conseguenza, poiché S è chiuso per succ e zero, otteniamo che:

    * $\mathrm{zero} \in S \implies \mathrm{succ}(\mathrm{zero}) \in S$
    * $\mathrm{succ}(\mathrm{zero}) \in S \implies \mathrm{succ}(\mathrm{succ}(\mathrm{zero})) \in S$
    * $\ldots$
    * $\mathrm{succ}(...(\mathrm{zero})) \in S \implies x = \mathrm{succ}(\mathrm{succ}(...(\mathrm{zero}))) \in S$

    Di conseguenza, otteniamo che $A \subseteq S$ e dunque che $S = A$

Osservazione 4

Equivalentemente, la terza condizione necessaria delle algebre induttive può essere considerata come

$$\nexists S \subsetneq A \mid (S, \gamma_1, \ldots, \gamma_n) \text{ è algebra induttiva}$$

Definizione 9: Omomorfismo

Date due strutture algebriche $(A, \gamma_1, \ldots, \gamma_k)$ e $(B, \delta_1, \ldots, \delta_k)$ dello stesso tipo, definiamo $f : A \to B$ come **omomorfismo** se

$$\forall a_1, \ldots, a_n \in A, i \in [1, k] \quad f(\gamma_i(a_1, \ldots, a_k)) = \delta_i(f(a_1), \ldots, f(a_k))$$

Esempi:

- Date le due algebre $(\mathbb{N}, \text{succ}, +)$ e $(\mathcal{N}, \text{succ}_{\mathcal{N}}, +_{\mathcal{N}})$, affinché la funzione $f : \mathbb{N} \to \mathcal{N}$ sia un omomorfismo è necessario che:

$$f(\text{succ}(n)) = \text{succ}_{\mathcal{N}}(f(n)) \qquad\qquad f(n+m) = f(n) +_{\mathcal{N}} f(m)$$

- Date le due algebre $(\mathbb{R}, +)$ e $(\mathbb{R}_{>0}, \cdot)$, la funzione $\exp : \mathbb{R} \to \mathbb{R}_{>0} : x \mapsto e^x$ è un omomorfismo:

$$\exp(x+y) = e^{x+y} = e^x e^y = \exp(x)\exp(y)$$

Definizione 10: Isomorfismo

Definiamo come **isomorfismo** un omomorfismo biettivo. Inoltre, definiamo due algebre $(A, \gamma_1, \ldots, \gamma_n)$, $(B, \delta_1, \ldots, \delta_n)$ come **isomorfe**, indicato con $A \cong B$, se esiste un isomorfismo tra loro.

Osservazione 5

Data una funzione $f : A \to B$, si ha che:

$$f \text{ è biettiva} \iff \exists f^{-1} : B \to A$$

(*dimostrazione omessa*)

Osservazione 6

Data una funzione $f : A \to B$, si ha che:

$$f \text{ è un isomorfismo} \iff f^{-1} \text{ è un isomorfismo}$$

(*dimostrazione omessa*)

Esempio:

- Date le due algebre $(\mathbb{R}, +)$ e $(\mathbb{R}_{>0}, \cdot)$, la funzione $\exp : \mathbb{R} \to \mathbb{R}_{>0} : x \mapsto e^x$ è un isomorfismo, poiché:
  - $\exp$ è un omomorfismo
  - $\exists \ln : \mathbb{R}_{>0} \to \mathbb{R} \mid \ln(\exp(x)) = x$, dunque f è biettiva.

1.1.1 Lemma di Lambek

Lemma 1

Data un'algebra induttiva $(A, \gamma_1, \ldots, \gamma_n)$, per ogni algebra $(B, \delta_1, \ldots, \delta_n)$ con la stessa segnatura di A si ha che

$$\exists! \text{ omomorfismo } f : A \to B$$

Nota: l'algebra di B non deve necessariamente essere induttiva

(*dimostrazione omessa*)

Teorema 1: Lemma di Lambek (versione ridotta)

Date due algebre induttive $(A, \gamma_1, \ldots, \gamma_n)$ e $(B, \delta_1, \ldots, \delta_n)$ con la stessa segnatura, si ha che $A \cong B$

Dimostrazione.

- Per il lemma precedente, si ha che:

$$\exists! \text{ omomorfismo } f : A \to B$$

$$\exists! \text{ omomorfismo } g : B \to A$$

- Consideriamo quindi la funzione $g \circ f : A \to A : x \mapsto g(f(x))$ e verifichiamo che essa sia un omomorfismo

$$g \circ f(x+y) = g(f(x+y)) = g(f(x)+f(y)) = g(f(x))+g(f(y)) = g \circ f(x)+g \circ f(y)$$

- Notiamo che per ogni algebra esiste sempre l'isomorfismo identità $\mathrm{id} : A \to A : x \mapsto x$ e poiché per il lemma precedente esiste necessariamente un unico omomorfismo tra A e A, ne segue necessariamente che $g \circ f = \mathrm{id}$
- Di conseguenza, si ha che

$$g \circ f = \mathrm{id} \iff g = f^{-1} \implies g, f \text{ biettive} \implies g, f \text{ isomorfismi} \implies A \cong B$$

□

Esempio:

- Date le due algebre induttive $(\mathbb{N}, \mathrm{zero}, \mathrm{succ})$ e $(\mathcal{N}, \mathrm{zero}_{\mathcal{N}}, \mathrm{succ}_{\mathcal{N}})$ sono isomorfe tra loro poiché aventi la stessa segnatura algebrica
- Difatti, come già dimostrato, $\mathbb{N}$ e $\mathcal{N}$ sono solamente due modi diversi per rappresentare lo stesso identico concetto algebrico

1.2 Strutture dati induttive

Definizione 11: Insieme delle liste finite

Definiamo `List<T>` come l'insieme delle liste finite di elementi di T

Esempio:

- Dato `List<Int>`, si ha che $[3 \to 5 \to 1] \in$ `List<Int>`

Proposizione 2: Algebra induttiva delle liste finite

La tripla $(\texttt{List<T>}, \text{empty}, \text{cons})$, dove:

- $\text{empty} : \mathbb{1} \to \texttt{List<T>} : x \mapsto [\ ]$ è la funzione nullaria che restituisce la **lista vuota**
- $\text{cons} : \texttt{List<T>} \times T \to \texttt{List<T>} : x, ([x_1 \to \ldots \to x_n]) \mapsto [x \to x_1 \to \ldots x_n]$ è la funzione di **costruzione delle liste**

è un'algebra induttiva

Dimostrazione.

1. La funzione empty risulta essere iniettiva poiché nullaria.

   Dati $\ell_1, \ell_2 \in \texttt{List<T>}$ e $x_1, x_2 \in T$, supponiamo che:

   $$\text{cons}(y_1, \ell_1) = \text{cons}(y_2, \ell_2) = [x_1 \to x_2 \to \ldots \to x_n]$$

   Per definizione stessa di cons, si ha che:

   $$\text{cons}(y_1, \ell_1) = \text{cons}(y_2, \ell_2) = [x \to x_1 \to \ldots \to x_n]$$

   $$\implies y_1 = y_2 = x, \ell_1 = \ell_2 = [x_1 \to \ldots \to x_n]$$

   dunque anche cons risulta iniettiva

2. $\text{im}(\text{empty}) \cap \text{im}(\text{cons}) = \{[\ ]\} \cap (\texttt{List<T>} - \{[\ ]\}) = \varnothing$

3. Sia $S \subseteq \texttt{List<T>}$ tale che $\forall x \in T, \ell \in \texttt{List<T>}\ \ \text{cons}(x, \ell) \in S$ e $\text{empty} \in S$.

   Preso $\ell := [x_1 \to x_2 \to \ldots \to x_n] \in \texttt{List<T>}$, possiamo esprimere ℓ come

   $$\ell = \text{cons}(x_1, \text{cons}(x_2, ...cons(x_n, \text{empty})))$$

   Di conseguenza, poiché S è chiuso per cons e empty e poiché empty $\in S$, otteniamo che ogni valore della catena sia contenuto in S, implicando che $x \in S$ e quindi che $\texttt{List<T>} \subseteq S$, concludendo che $S = \texttt{List<T>}$

□

Osservazione 7

La tripla $(\texttt{List<T>}_\infty, \text{empty}, \text{cons})$, dove $\texttt{List<T>}_\infty$ è l'insieme delle liste infinite di elementi di T **non è un'algebra induttiva**, poiché $\texttt{List<T>} \subsetneq \texttt{List<T>}_\infty$ e poiché $(\texttt{List<T>}, \text{empty}, \text{cons})$ è un'algebra induttiva

Osservazione 8

Tramite i costruttori di un'algebra induttiva è possibile definire le ulteriori operazioni "aggiuntive" di tale algebra

Esempio:

- Data l'algebra induttiva $(\texttt{List<T>}, \text{empty}, \text{cons})$, definiamo la seguente operazione

$$\text{concat} : \texttt{List<T>} \times \texttt{List<T>} \to \texttt{List<T>}$$

dove:

$$\begin{cases} \text{concat}(\text{empty}, \ell) = \ell \\ \text{concat}(\text{cons}(n, \ell), \ell') = \text{cons}(n, \text{concat}(\ell, \ell')) \end{cases}$$

- Ad esempio, in $\texttt{List<Int>}$, abbiamo che:

$$\text{concat}([1 \to 5], [7 \to 2]) = \text{concat}(\text{cons}(1, [5], [7 \to 2])) = \text{cons}(1, \text{concat}([5], [7 \to 2])) =$$
$$\text{cons}(1, \text{concat}(\text{cons}(5, \text{empty}), [7 \to 2])) = \text{cons}(1, \text{cons}(5, \text{concat}(\text{empty}, [7 \to 2]))) =$$
$$\text{cons}(1, \text{cons}(5, [7 \to 2])) = \text{cons}(1, [5 \to 7 \to 2]) = [1 \to 5 \to 7 \to 2]$$

Definizione 12: Insieme degli alberi binari finiti

Definiamo $\texttt{BinTree}$ come l'insieme degli alberi binari finiti

Proposizione 3: Algebra induttiva degli alberi binari finiti

La tripla $(\texttt{BinTree}, \text{leaf}, \text{branch})$, dove:

- $\text{leaf} : \mathbb{1} \to \texttt{BinTree} : x \mapsto \circ$ è la funzione nullaria che restituisce una **foglia**
- $\text{branch} : \texttt{BinTree} \times \texttt{BinTree} \to \texttt{BinTree} : (t_{sx}, t_{dx}) \mapsto t$ è la funzione di **costruzione dei rami**, ossia tale che

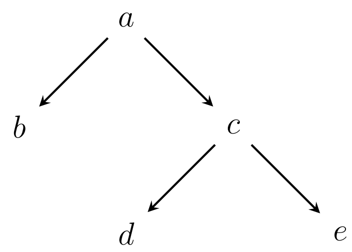
$$\begin{tikzcd} & t \arrow[dl] \arrow[dr] & \\ t_{sx} & & t_{dx} \end{tikzcd}$$

è un'algebra induttiva

(*dimostrazione omessa*)

Esempio:

- Il seguente albero



  corrisponde a:

$$a = \text{branch}(\text{leaf}, \text{branch}(\text{leaf}, \text{leaf}))$$

1.2.1 Induzione strutturale

Definizione 13: Induzione strutturale

Definiamo come **induzione strutturale** il metodo dimostrativo generalizzante il principio di induzione e basato sulle proprietà di un'algebra induttiva.

In particolare, viene ipotizzato che una proprietà P valga per ogni argomento di ogni costruttore dell'algebra e tramite il terzo assioma viene dimostrato che tale proprietà valga per tutti gli elementi dell'algebra stessa

Teorema 2: Relazione tra nodi e foglie

Dato $t \in \texttt{BinTree}$ avente n foglie, il numero di nodi di t è pari a $2n - 1$

Dimostrazione per induzione strutturale.

- Definiamo l'operazione

$$\text{leaves} : \texttt{BinTree} \to \mathbb{N} : t \mapsto \text{Numero di foglie in } b$$

  dove:

$$\begin{cases} \text{leaves}(\text{leaf}) = 1 \\ \text{leaves}(\text{branch}(b_1, b_2)) = \text{leaves}(b_1) + \text{leaves}(b_2) \end{cases}$$

- Dato $t \in \texttt{BinTree}$, sia k il numero di nodi di t e sia $n = \text{leaves(t)}$

  Caso base. Se $t = \text{leaf}$, allora t è composto da $k = 1$ nodi e $n = \textbf{leaves}(\text{leaf}) = 1$ foglie. Difatti, si ha che $k = 1 = 2n - 1$

  Ipotesi induttiva. Ogni argomento t' di ogni costruttore possiede $k' = 2\textbf{leaves}(t') - 1$ nodi

Passo induttivo. Se $t \neq \text{leaf}$, allora $\exists t_1, t_2 \in \texttt{BinTree} \mid t = \text{branch}(t_1, t_2)$ dove t_1 e t_2 possiedono rispettivamente k_1 e k_2 nodi. Inoltre, si ha che $k = k_1 + k_2 + 1$

In quanto t_1 e t_2 sono argomenti del costruttore branch, per ipotesi induttiva si ha che:

$$k = k_1 + k_2 + 1 = 2\text{leaves}(t_1) - 1 + 2\text{leaves}(t_2) - 1 + 1 = 2(\text{leaves}(t_1) + \text{leaves}(t_2)) - 1 =$$

$$= 2(\text{leaves}(\text{branch}(t_1, t_2))) - 1 = 2(\text{leaves}(t)) - 1$$

□

1.3 Sintassi astratta

Definizione 14: Linguaggio

Definiamo come **linguaggio** un insieme di stringhe

Definizione 15: Grammatica

Definiamo come **grammatica** un insieme di regole, dette **termini**, che definiscono come poter manipolare le stringhe di un linguaggio.

La **forma di Backus-Naur** è una notazione utilizzata per descrivere grammatiche ed è definita come:

`<symbol>` ::= `_expression_`

dove:

- `<symbol>` è una simbolo non-terminale espresso dalla grammatica
- L'operatore ::= indica che ciò che si trova alla sua sinistra possa essere sostituito con ciò che si trova alla sua destra
- `<_expression_>` consiste in una o più sequenze di simboli terminali o non-terminali dove ogni sequenza è separata da una barra verticale (ossia |) indicante una scelta possibile per l'operatore ::=

Esempio:

- Consideriamo il linguaggio L espresso dalla grammatica:

$$M, N ::= 0 \mid 1 \mid \dots \mid M + N \mid M * N$$

  Tale grammatica indica che i simboli non-terminali M e N possono essere sostituiti con:

  - Un numero naturale
  - Un'espressione $M + N$ o $M * N$ dove M e N sono due ulteriori simboli terminali o non-terminali

- Ad esempio, abbiamo che la stringa `"5 + 7"` sia ben definita dalla grammatica, mentre la stringa `"5 + +"` non lo sia

Definizione 16: Sintassi astratta

La **sintassi astratta** di un linguaggio è una definizione induttiva di un insieme T di termini, permettendo di definire strutture algebriche senza dover necessariamente definire concretamente le sue operazioni

Esempio:

- Consideriamo ancora il linguaggio L definito dalla grammatica
$$M, N ::= 0 \mid 1 \mid \ \dots \ \mid M + N \mid M * N$$
- Definiamo quindi la funzione $\mathrm{eval} : L \to \mathbb{N}$ in grado di valutare le espressioni del linguaggio:
$$\begin{aligned}\mathrm{eval}(\texttt{"0"}) &= 0\\ \mathrm{eval}(\texttt{"1"}) &= 1\\ &\dots\\ \mathrm{eval}(\texttt{"M + N"}) &= \mathrm{eval}(\texttt{"M"}) + \mathrm{eval}(''N'')\\ \mathrm{eval}(\texttt{"M * N"}) &= \mathrm{eval}(\texttt{"M"}) + \mathrm{eval}(''N'')\end{aligned}$$
- Notiamo quindi che la grammatica definisca in modo astratto (ma concretamente tramite eval) le seguenti operazioni:
$$\begin{aligned}0 : \mathbb{1} \to \mathbb{N} : x \mapsto 0\\ 1 : \mathbb{1} \to \mathbb{N} : x \mapsto 1\\ \dots\\ \mathrm{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (m, n) \mapsto m + n\\ \mathrm{times} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (m, n) \mapsto m \cdot n\end{aligned}$$
- Notiamo però che le operazioni plus e times non risultano essere né iniettive né con immagini disgiunte. Di conseguenza, la funzione eval non ci permette di definire un'algebra induttiva.
- Tuttavia, per tale linguaggio è comunque possibile definire (in qualche modo, ad esempio fissando una precedenza per le operazioni rompendo proprietà come l'associatività e la commutatività) una funzione che possa descrivere un'algebra induttiva.

Teorema 3: Algebra induttiva dei termini

Dato un linguaggio L con una sintassi astratta con termini definiti in T, esiste sempre un'algebra induttiva (T, α). Di conseguenza, **tutte le proprietà** di un linguaggio sono dimostrabili tramite l'induzione strutturale sulla sua algebra dei termini.

(*dimostrazione omessa*)

2

Paradigma funzionale

2.1 *Exp*: un semplice linguaggio funzionale

Definizione 17: Il linguaggio *Exp*

Definiamo come Exp il linguaggio rappresentato dalla seguente grammatica:

$$M, N ::= k \mid x \mid M + N \mid let\ x = M \text{ in } N$$

dove:

- $k \in \{0, 1, \ldots\}$ ossia è una **costante**
- $x \in Var = \{x, y, z, \ldots\}$ ossia è una **variabile**
- $+ : Exp \times Exp \to Exp$ la quale **somma le due espressioni**
- $let : Var \times Exp \times Exp \to Exp$ la quale **assegna** alla variabile x l'espressione M all'interno della **valutazione** di N. Inoltre, x prende il nome di variabile locale all'interno di N.
- $Val = \{0, 1, \ldots\}$ è l'**insieme dei valori** in cui un'espressione può essere valutata

Esempi:

- L'espressione $let\ x = 3 \text{ in } x + 1$ indica che la variabile x assuma valore 3 all'interno della valutazione di $x + 1$. Di conseguenza, il risultato della valutazione dell'espressione è 4
- L'espressione $let\ x = 3 \text{ in } 7$ viene valutata come 7
- L'espressione $let\ y = 9 \text{ in } (let\ x = (let\ y = 2 \text{ in } y + 1) \text{ in } x + y)$ viene valutata come 12 (si consiglia di cercare di capire come le clausole interne sovrascrivano i valori delle clausole esterne. Se ciò risultasse complesso, più avanti verranno forniti strumenti matematici per valutare in modo corretto le clausole *let* annidate)

Definizione 18: Scope di una variabile

Data un'espressione e una variabile x, definiamo come **scope di** x la porzione la porzione dell'espressione all'interno della quale una variabile può essere riferita, ossia per cui ne è definito il valore.

Una variabile il cui valore non è assegnato in una porzione dell'espressione viene detta **variabile libera**

Definizione 19: Variabile libera

Data un'espressione $expr \in Exp$, definiamo $x \in expr$ come **libera** se x non ha un valore assegnato durate la valutazione di $expr$.

Esempio:

- L'espressione $let\ x = (let\ y = 2 \text{ in } y+1) \text{ in } x+y$ non è coerente con la grammatica di Exp, poiché y non è definito durante la valutazione di $x+y$. Di conseguenza, non è possibile valutare tale espressione.

Proposizione 4: Variabili libere in *Exp*

Dato il linguaggio Exp, la funzione

$$\text{free} : Exp \to \mathcal{P}(Var)$$

restituisce l'insieme di tutte le **variabili libere** di un'espressione dove:

$$\begin{cases} \text{free}(k) = \varnothing \\ \text{free}(x) = \{x\} \\ \text{free}(M+N) = \text{free}(M) \cup \text{free}(N) \\ \text{free}(let\ x = M \text{ in } N) = \text{free}(M) \cup (\text{free}(N) - \{x\}) \end{cases}$$

Nota: $\mathcal{P}(Var)$ è l'insieme delle parti di Var, ossia l'insieme contenente tutti i suoi sottoinsiemi possibili

Esempio:

- Riprendendo l'esempio precedente, notiamo che:

$$\text{free}(let\ x = (let\ y = 2 \text{ in } y+1) \text{ in } x+y) =$$

$$\begin{gathered} = \text{free}(let\ y = 2 \text{ in } y+1) \cup (\text{free}(x+y) - \{x\}) = \\ = \text{free}(let\ y = 2 \text{ in } y+1) \cup ((\text{free}(x) \cup \text{free}(y)) - \{x\}) = \\ = \text{free}(let\ y = 2 \text{ in } y+1) \cup ((\{x\} \cup \{y\}) - \{x\}) = \\ = \text{free}(let\ y = 2 \text{ in } y+1) \cup \{y\} = \end{gathered}$$

$$= (\text{free}(2) \cup (\text{free}(y+1) - \{y\})) \cup \{y\} =$$
$$= ((\text{free}(y)) - \{y\}) \cup \{y\} =$$
$$= \{y\}$$

dunque l'espressione è invalutabile

Definizione 20: Insieme degli ambienti

Dato il linguaggio *Exp*, definiamo come **insieme degli ambienti di** *Exp*, indicato con *Env*, l'insieme delle funzioni parziali (ossia non necessariamente definite su tutto il dominio) che associano ogni variabile al proprio valore:

$$Env = \{f \mid f : Var \xrightarrow{fin} Val\}$$

Definizione 21: Concatenazione di ambienti

Dato il linguaggio *Exp*, definiamo l'operazione di **concatenazione di ambienti**, ossia:

$$\cdot : Env \times Env \to Env$$

dove:

$$(E_1 E_2)(x) = \begin{cases} E_2(x) & \text{se } x \in dom(E_1) \\ E_1(x) & \text{altrimenti} \end{cases}$$

Nota: tale operazione può essere interpretata come una sovrascrittura in E_1 di tutte le variabili definite in E_2

Esempio:

- Dati gli ambienti $E_1 = \{(x,4),(y,3)\}$ e $E_2 = \{(x,5)\}$, si ha che

$$(E_1 E_2)(x) = 5$$

$$(E_1 E_2)(y) = 3$$

Proposizione 5: Regola di inferenza

Data la proposizione:

$$\text{Premessa 1} \wedge \ldots \wedge \text{Premessa n} \implies \text{Conclusione}$$

definiamo come **regola di inferenza** la notazione alternativa:

$$\frac{\text{Premessa 1} \quad \ldots \quad \text{Premessa n}}{\text{Conclusione}}$$

Definizione 22: Semantica operazionale

Data la seguente relazione detta **semantica operazionale**, ossia:

$$\leadsto \subseteq Env \times Exp \times Val$$

definiamo come **giudizio operazionale** la tripla $(E, M, v) \in \leadsto$ descritta dalla notazione

$$E \vdash M \leadsto v$$

la quale viene letta come "nell'ambiente E, M viene valutato come v".

Proposizione 6: Regole operazionali di *Exp*

Definiamo come **regole operazionali** le regole di inferenza che dettano le valutazioni effettuate dalla semantica operazionale:

- Per le **costanti** si ha che:

$$\forall E \in Env \quad E \vdash k \leadsto k$$

- Dato $E \in Env$, per le **variabili** si ha che:

$$x \in dom(E) \wedge E(x) = v \implies E \vdash x \leadsto v$$

- Dato $E \in Env$, per la **somma** si ha che:

$$u = v + v' \implies \frac{E \vdash M \leadsto v \quad E \vdash N \leadsto v'}{E \vdash M + N \leadsto u}$$

- Per l'espressione ***let*** si ha che:

$$\frac{E \vdash M \leadsto v \quad E\{(x, v)\} \vdash N \leadsto v'}{E \vdash let\ x = M \text{ in } N \leadsto v'}$$

Osservazione 9: Ambiente iniziale

A meno che non vi siano variabili esternamente assegnate, all'interno di un'espressione l'**ambiente iniziale** corrisponde sempre a $\varnothing \subseteq Env$.

Osservazione 10: Variabili invalutabili

Dato un ambiente $E \in Env$, se $x \notin dom(E)$, ossia se x non è definita nell'ambiente E, allora x è una **variabile libera** e dunque è **invalutabile** in E, ossia:

$$\nexists v \in Val \text{ t.c. } E \vdash x \leadsto v$$

Esempio:

- L'espressione $x + 4$ è invalutabile, poiché $x \notin dom(\varnothing)$, dunque:

$$\nexists v' \in Val \text{ t.c. } v = v' + 1 \wedge \frac{\varnothing \vdash x \rightsquigarrow v' \quad \varnothing \vdash 1 \rightsquigarrow 1}{\varnothing \vdash x + 1 \rightsquigarrow v}$$

- L'espressione $let\ x = 1 \text{ in } x + 4$ è valutabile, poiché $x \in dom(\{(x, 1)\})$, dunque:

$$\frac{\varnothing \vdash 1 \rightsquigarrow 1 \qquad \dfrac{\{(x,1)\} \vdash x \rightsquigarrow 1 \quad \{(x,1)\} \vdash 4 \rightsquigarrow 4}{\{(x,1)\} \vdash x + 1 \rightsquigarrow 5}}{\varnothing \vdash let\ x = 1 \text{ in } x + 4 \rightsquigarrow 5}$$

Definizione 23: Albero di derivazione

Definiamo come **albero di derivazione** l'albero generato dalla valutazione concatenata di più regole di inferenza.

Esempio:

- L'espressione $let\ y = 3 \text{ in } (let\ x = 7 \text{ in } x + y)$ viene valutata dal seguente albero di derivazione:

$$\frac{\varnothing \vdash 3 \rightsquigarrow 3 \qquad \dfrac{\{y, 3\} \vdash 7 \rightsquigarrow 7 \qquad \dfrac{\{(y,3),(x,7)\} \vdash x \rightsquigarrow 7 \quad \{(y,3),(x,7)\} \vdash y \rightsquigarrow 3}{\{(y,3),(x,7)\} \vdash x + y \rightsquigarrow 10}}{\{(y,3)\} \vdash let\ x = 7 \text{ in } x + y \rightsquigarrow 10}}{\varnothing \vdash let\ y = 3 \text{ in } (let\ x = 7 \text{ in } x + y) \rightsquigarrow 10}$$

- Notiamo quindi come, per valutare l'intera espressione, ci basti in realtà valutare i termini "più in alto" dell'albero di derivazione

2.2 Valutazione Eager vs Lazy

Consideriamo la seguente espressione per il linguaggio *Exp*:

$$let\ x = \sqrt{397^5 + \int_3^{15} y^2\, dy} + \log_{\sqrt{37}}(479) \text{ in } 3$$

Notiamo come nonostante l'espressione assegnata ad x sia di grandi dimensioni, richiedendo un enorme albero di derivazione, la valutazione dell'espressione sia totalmente indipendente da tale valutazione in quanto la variabile x non venga neanche utilizzata per la valutazione del secondo termine dell'espressione *let*.

Utilizzando le regole di valutazione previste dalla metodologia di valutazione, detta *eager* (trad: *affrettata*), vista nella sezione precedente, andremmo a valutare delle espressioni del tutto inutili.

Una metodologia di valutazione alternativa, detta *lazy*, è costituita da regole operazionali atte al *ritardare* la valutazione dei termini fino a quando non sia strettamente necessario.

Definizione 24: Valutazione eager

Definiamo una modalità di valutazione come **eager** (o valutazione *call-by-name* o *call-by-value*) se la valutazione di una sua espressione viene effettuata non appena essa viene legata ad una variabile, associandone immediatamente il risultato alla variabile stessa.

Definizione 25: Valutazione lazy

Definiamo una modalità di valutazione come **lazy** (o valutazione *call-by-need*) se la valutazione di una sua espressione viene effettuata solo quando si richiede il valore di un'espressione che da essa dipende.

Proposizione 7: Linguaggio *Exp* lazy

L'uso di una valutazione lazy necessita la ridefinizione dell'insieme *Env* e di alcune regole operazionali definite per la valutazione eager:

- L'insieme *Env* viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Exp\}$$

- Dato $E \in Env$, per le variabili si ha che:

$$x \in dom(E) \wedge E(x) = M \implies \frac{E \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v}$$

- Per l'espressione *let* si ha che:

$$\frac{E\{(x, M)\} \vdash N \rightsquigarrow v}{E \vdash let\ x = M \text{ in } N \rightsquigarrow v}$$

Osservazione 11

È necessario puntualizzare che non sempre la valutazione lazy sia più ottimale della eager

Esempio:

- Consideriamo la seguente espressione

$$let\ x = M \text{ in } x + x$$

- Utilizzando la valutazione eager otteniamo il seguente albero di derivazione:

$$\dfrac{\dfrac{\ldots}{\varnothing \vdash M \rightsquigarrow v'} \qquad \dfrac{\{(x,v')\} \vdash x \rightsquigarrow v' \quad \{(x,v')\} \vdash x \rightsquigarrow v'}{\{(x,v')\} \vdash x + x \rightsquigarrow v}}{\varnothing \vdash \mathit{let}\ x = M \text{ in } x + x \rightsquigarrow v}$$

  dove $v = v' + v'$

- Utilizzando la valutazione lazy, invece, otteniamo il seguente albero di derivazione:

$$\dfrac{\dfrac{\dfrac{\dfrac{\ldots}{\{(x,M)\} \vdash M \rightsquigarrow v'}}{\{(x,M)\} \vdash x \rightsquigarrow v'} \qquad \dfrac{\dfrac{\ldots}{\{(x,M)\} \vdash M \rightsquigarrow v'}}{\{(x,M)\} \vdash x \rightsquigarrow v'}}{\{(x,M)\} \vdash x + x \rightsquigarrow v}}{\varnothing \vdash \mathit{let}\ x = M \text{ in } x + x \rightsquigarrow v}$$

  dove $v = v' + v'$

- Notiamo quindi che l'espressione M venga valutata una sola volta nella valutazione eager ma due volte nella valutazione lazy

2.3 Scoping Statico vs Dinamico

Consideriamo la seguente espressione:

$$\mathit{let}\ x = 3 \text{ in } (\mathit{let}\ y = x \text{ in } (\mathit{let}\ x = 7 \text{ in } y + x))$$

Prima di tutto, valutiamo tale espressione tramite valutazione eager:

$$\dfrac{\varnothing \vdash 3 \rightsquigarrow 3 \qquad \dfrac{\{(x,3)\} \vdash x \rightsquigarrow 3 \qquad \dfrac{E \vdash 7 \rightsquigarrow 7 \qquad \dfrac{E\{(x,7)\} \vdash y \rightsquigarrow 3 \quad E\{(x,7)\} \vdash x \rightsquigarrow 7}{E\{(x,7)\} \vdash y + x \rightsquigarrow 10}}{\{(x,3),(y,3)\} \vdash \mathit{let}\ x = 7 \text{ in } y + x \rightsquigarrow 10}}{\{(x,3)\} \vdash \mathit{let}\ y = x \text{ in } (\mathit{let}\ x = 7 \text{ in } y + x) \rightsquigarrow 10}}{\varnothing \vdash \mathit{let}\ x = 3 \text{ in } (\mathit{let}\ y = x \text{ in } (\mathit{let}\ x = 7 \text{ in } y + x)) \rightsquigarrow 10}$$

dove $E := \{(x,3),(y,3)\}$

Valutiamo ora invece tale espressione utilizzando una valutazione lazy:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E\{(x,7)\} \vdash 7 \rightsquigarrow 7}{E\{(x,7)\} \vdash x \rightsquigarrow 7}}{E\{(x,7)\} \vdash y \rightsquigarrow 7} \qquad \dfrac{E\{(x,7)\} \vdash 7 \rightsquigarrow 7}{E\{(x,7)\} \vdash x \rightsquigarrow 7}}{E\{(x,7)\} \vdash y + x \rightsquigarrow 14}}{\{(x,3),(y,x)\} \vdash \mathit{let}\ x = 7 \text{ in } y + x \rightsquigarrow 14}}{\{(x,3)\} \vdash \mathit{let}\ y = x \text{ in } (\mathit{let}\ x = 7 \text{ in } y + x) \rightsquigarrow 14}}{\varnothing \vdash \mathit{let}\ x = 3 \text{ in } (\mathit{let}\ y = x \text{ in } (\mathit{let}\ x = 7 \text{ in } y + x)) \rightsquigarrow 14}$$

dove $E := \{(x,3),(y,x)\}$

Notiamo quindi che le due valutazioni abbiano prodotto un risultato diverso. Tuttavia, vorremmo che le due valutazioni siano differenti solo a livello *"implementativo"*, ossia che venga solo ritardata la valutazione dei termini. Difatti, tale problematica non è dovuta alla metodologia di valutazione utilizzata ma bensì dal tipo di *scoping*.

Definizione 26: Scoping statico

Definiamo un linguaggio come linguaggio a **scoping statico** se durante la valutazione di un'espressione viene utilizzato l'ambiente definito al tempo in cui viene interpretata (ma non valutata) l'espressione stessa.

Definizione 27: Scoping dinamico

Definiamo un linguaggio come linguaggio a **scoping statico** se durante la valutazione di un'espressione viene utilizzato l'ambiente definito al tempo di valutazione stesso.

Difatti, nell'esempio precedente ci troviamo in due situazioni:

- Nella valutazione eager, la variabile y viene valutata con l'ambiente $\{(x, 3), (y, x)\}$ definito al tempo in cui viene interpretata l'espressione $let\ y = x \text{ in } \dots$ (scoping *statico*)
- Nella valutazione lazy, la variabile y viene valutata con l'ambiente $\{(x, 3), (y, x), (x, 7)\}$ definito al tempo della sua valutazione (scoping *dinamico*)

Per tanto, è necessario precisare che le due precedenti versioni viste del linguaggio Exp siano rispettivamente la versione **eager statica** e la versione Exp **lazy dinamica**.

Proposizione 8: Linguaggio Exp lazy statico

L'uso di una semantica lazy statica necessita la ridefinizione dell'insieme Env e di alcune regole operazionali definite per la semantica lazy dinamica:

- L'insieme Env viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Exp \times Env\}$$

- Dato $E \in Env$, per le variabili si ha che:

$$x \in dom(E) \wedge E(x) = (M, E') \implies \frac{E' \vdash M \leadsto v}{E \vdash x \leadsto v}$$

- Per l'espressione *let* si ha che:

$$\frac{E\{(x, (M, E))\} \vdash N \leadsto v}{E \vdash let\ x = M \text{ in } N \leadsto v}$$

Valutiamo quindi l'espressione precedente utilizzando una semantica lazy statica:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\varnothing \vdash 3 \rightsquigarrow 3}{E \vdash x \rightsquigarrow 3}}{E'' \vdash y \rightsquigarrow 3} \quad \dfrac{E' \vdash 7 \rightsquigarrow 7}{E'' \vdash x \rightsquigarrow 7}}{E'\{(x,(7,E'))\} \vdash y + x \rightsquigarrow 10}}{E\{(y,(x,E))\} \vdash \mathit{let}\ x = 7 \text{ in } y + x \rightsquigarrow 10}}{\{(x,(3,\varnothing))\} \vdash \mathit{let}\ y = x \text{ in } (\mathit{let}\ x = 7 \text{ in } y + x) \rightsquigarrow 10}}{\varnothing \vdash \mathit{let}\ x = 3 \text{ in } (\mathit{let}\ y = x \text{ in } (\mathit{let}\ x = 7 \text{ in } y + x)) \rightsquigarrow 10}$$

dove $E := \{(x,(3,\varnothing))\}$, $E' := E\{(y,(x,E))\}$ e $E'' := E'\{(x,(7,E'))\}$. Notiamo quindi che la valutazione nel caso di *Exp* lazy statico coincida con la valutazione nel caso di *Exp* eager statico.

Osservazione 12: Linguaggio *Exp* eager dinamico

All'interno del linguaggio *Exp* **non vi è distinzione** tra semantica eager statica e eager dinamica, poiché nessuna delle valutazioni dei termini della grammatica di *Exp* viene influenzata dal tipo di scoping.

Per tanto, all'interno di *Exp* parliamo direttamente di **semantica eager**

Definizione 28: Equivalenza tra semantiche operazionali

Sia L un linguaggio. Date due semantiche operazionali definite su L, definiamo tali semantiche come **equivalenti** se ogni espressione di L restituisce lo stesso risultato per entrambe le semantiche a seguito della valutazione

Teorema 4: Equivalenze semantiche di *Exp*

Dato il linguaggio *Exp*, si ha che:

$$\mathit{Exp} \text{ eager } \equiv \mathit{Exp} \text{ lazy statico } \not\equiv \mathit{Exp} \text{ lazy dinamico}$$

Osservazione 13

In base alla semantica utilizzata, possono generarsi problemi diversi durante le valutazioni

Esempio:

- Consideriamo la seguente espressione $\mathit{let}\ x = x \text{ in } x$
- Utilizzando una semantica eager statica o lazy statica, otteniamo che il termine interno del *let* sia invalutabile
- Utilizzando una semantica lazy dinamica, la valutazione entrerà in un loop infinito (si consiglia di provare ad scrivere l'albero di derivazione)

2.4 *Fun*: un linguaggio con funzioni

Definizione 29: Il linguaggio *Fun*

Definiamo come Fun il linguaggio rappresentato dalla seguente grammatica:

$$M, N ::= k \mid x \mid M + N \mid let\ x = M \text{ in } N \mid fn\ x \Rightarrow M \mid MN$$

dove:

- $k \in \{0, 1, \ldots\}$ ossia è una **costante**
- $x \in Var = \{x, y, z, \ldots\}$ ossia è una **variabile**
- $+ : Fun \times Fun \to Fun$ la quale **somma le due espressioni**
- $let : Var \times Fun \times Fun \to Fun$ la quale **assegna** alla variabile x l'espressione M all'interno della **valutazione** di N. Inoltre, x prende il nome di variabile locale all'interno di N
- $fn : Var \times Fun \to Fun$ la quale restituisce una **funzione** avente un parametro il quale influenza l'espressione valutata dalla funzione
- Data l'espressione $fn\ x \Rightarrow M$, definiamo la coppia $(x, M) \in Var \times Fun$ come **chiusura** di tale espressione
- $\cdot : Fun \times Fun \to Fun$ la quale **applica** il termine sinistro al termine destro. In particolare, è <u>necessario</u> che il termine sinistro sia una funzione
- $Val = \{0, 1, \ldots\} \cup (Var \times Fun)$ è l'**insieme dei valori** in cui un'espressione può essere valutata, ossia costanti e chiusure

Esempi:

- L'espressione $(fn\ x \Rightarrow x + 1)\ 7$ viene valutata come 8, poiché la funzione sinistra $fn\ x \Rightarrow x + 1$ viene applicata al termine destro 7 (dunque 7 viene utilizzato come argomento della funzione per il parametro x)
- L'espressione $(fn\ x \Rightarrow x\ 3)\ 7$ è invalutabile, poiché l'argomento 7 viene passato come parametro x della funzione, ma all'interno di quest'ultima non è possibile valutare $x\ 3$ visto che 7 non è applicabile a 3
- L'espressione $(fn\ x \Rightarrow x\ 3)(fn\ x \Rightarrow x+1)$ viene valutata come 4, poiché l'argomento $fn\ x \Rightarrow x + 1$ viene passato come parametro x della funzione $fn\ x \Rightarrow x\ 3$, per poi valutare l'applicazione $x\ 3$ passando l'argomento 3 come parametro per la funzione contenuta in x (ossia $fn\ x \Rightarrow x + 1$).

  <u>Informalmente</u>, possiamo dire che:

  $$(fn\ x \Rightarrow x\ 3)(fn\ x \Rightarrow x + 1) \longrightarrow (fn\ x \Rightarrow x + 1)\ 3 \longrightarrow 4$$

Osservazione 14

Nel caso in cui si abbia un'espressione con doppio operatore di applicazione MNL, essa verrà valutata come $(MN)L$

Esempio:

- Le due espressioni $(fn\ x \Rightarrow x\ 3)(fn\ x \Rightarrow x+1)\ 7$ e $[(fn\ x \Rightarrow x\ 3)(fn\ x \Rightarrow x+1)]\ 7$ sono equivalenti

Definizione 30: Curryficazione

Definiamo come **curryficazione** la contrazione sintattica $fn\ x_1x_2\dots x_n \Rightarrow M$ equivalente alla seguente espressione:

$$fn\ x_1 \Rightarrow (fn\ x_2 \Rightarrow \dots (fn\ x_n \Rightarrow M)\dots)$$

Esempi:

- L'uncurryficazione dell'espressione $(fn\ xy \Rightarrow yx)\ 7\ (fn\ x \Rightarrow x+1)$ corrisponde a:

  $$(fn\ x \Rightarrow fn\ y \Rightarrow yx)\ 7\ (fn\ x \Rightarrow x+1)$$

  e viene pertanto valutata come 8. Difatti, <u>informalmente</u>, possiamo dire che:

  $$(fn\ x \Rightarrow fn\ y \Rightarrow yx)\ 7\ (fn\ x \Rightarrow x+1) \longrightarrow (fn\ y \Rightarrow y\ 7)(fn\ x \Rightarrow x+1) \longrightarrow 8$$

Osservazione 15

Trattandosi di un'estensione del linguaggio *Exp*, il linguaggio *Fun* **eredita le regole operazionali** delle semantiche di *Exp*

Proposizione 9: Linguaggio *Fun* eager dinamico

La semantica eager dinamica del linguaggio *Fun* prevede l'aggiunta di alcune regole operazionali:

- L'insieme *Env* viene ridefinito come:

  $$Env = \{f \mid f : Var \xrightarrow{fin} Val\}$$

- Dato $E \in Env$, per le funzioni si ha che:

  $$E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M)$$

- Dato $E \in Env$, per le applicazioni si ha che:

  $$\frac{E \vdash M \rightsquigarrow (x, L) \quad E \vdash N \rightsquigarrow v' \quad E\{(x, v')\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$$

Proposizione 10: Linguaggio Fun eager statico

La semantica eager statica del linguaggio Fun prevede l'aggiunta di alcune regole operazionali:

- L'insieme Env viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Val \times Env\}$$

- Dato $E \in Env$, per le funzioni si ha che:

$$E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M, E)$$

- Dato $E \in Env$, per le applicazioni si ha che:

$$\frac{E \vdash M \rightsquigarrow (x, L, E') \quad E \vdash N \rightsquigarrow v' \quad E'\{(x, v')\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$$

Lemma 2

A differenza del linguaggio Exp, per la sua estensione Fun si ha che:

$$Fun \text{ eager dinamico } \not\equiv Fun \text{ eager statico}$$

Dimostrazione.

- Consideriamo l'espressione $let\ x = 7 \text{ in } ((fn\ y \Rightarrow let\ x = 3 \text{ in } yx)(fn\ z \Rightarrow x))$
- Utilizzando la semantica eager dinamica, l'albero di derivazione corrisponde a:

$$(*) \qquad \dfrac{E' \vdash 3 \rightsquigarrow 3 \qquad \dfrac{E'' \vdash y \rightsquigarrow (z, x) \quad E'' \vdash x \rightsquigarrow 3 \quad E''\{(z, 3)\} \vdash x \rightsquigarrow 3}{E'' \vdash yx \rightsquigarrow 3}}{E' \vdash let\ x = 3 \text{ in } yx \rightsquigarrow 3}$$

$$\dfrac{\varnothing \vdash 7 \rightsquigarrow 7 \qquad \dfrac{E \vdash fn\ y \Rightarrow let\ x = 3 \text{ in } yx \rightsquigarrow (y, let\ x = 3 \text{ in } yx) \quad E \vdash fn\ z \Rightarrow x \rightsquigarrow (z, x) \quad (*)}{E \vdash (fn\ y \Rightarrow let\ x = 3 \text{ in } yx)(fn\ z \Rightarrow x) \rightsquigarrow 3}}{\varnothing \vdash let\ x = 7 \text{ in } ((fn\ y \Rightarrow let\ x = 3 \text{ in } yx)(fn\ z \Rightarrow x)) \rightsquigarrow 3}$$

dove $E := \{(x, 7)\}$, $E' := E\{(y, (z, x))\}$ e $E'' := E'\{(x, 3)\}$

- Utilizzando la semantica eager statica, invece, l'albero di derivazione corrisponde a:

$$(*) \qquad \dfrac{E' \vdash 3 \rightsquigarrow 3 \qquad \dfrac{E'' \vdash y \rightsquigarrow (z, x, E) \quad E'' \vdash x \rightsquigarrow 3 \quad E\{(z, 3)\} \vdash x \rightsquigarrow 7}{E'' \vdash yx \rightsquigarrow 7}}{E' \vdash let\ x = 3 \text{ in } yx \rightsquigarrow 7}$$

$$\dfrac{\varnothing \vdash 7 \leadsto 7 \qquad \dfrac{E \vdash fn\ y \Rightarrow let\ x = 3 \text{ in } yx \leadsto (y, let\ x = 3 \text{ in } yx, E) \qquad E \vdash fn\ z \Rightarrow x \leadsto (z, x, E) \quad (*)}{E \vdash (fn\ y \Rightarrow let\ x = 3 \text{ in } yx)(fn\ z \Rightarrow x) \leadsto 7}}{\varnothing \vdash let\ x = 7 \text{ in } ((fn\ y \Rightarrow let\ x = 3 \text{ in } yx)(fn\ z \Rightarrow x)) \leadsto 7}$$

dove $E := \{(x, 7)\}$, $E' := E\{(y, (z, x, E))\}$ e $E'' := E'\{(x, 3)\}$

- Poiché l'espressione restituisce due valutazioni diverse, le due semantiche non sono equivalenti

□

Proposizione 11: Linguaggio *Fun* lazy dinamico

La semantica lazy dinamica del linguaggio *Fun* prevede l'aggiunta di alcune regole operazionali:

- L'insieme Env viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Fun\}$$

- Dato $E \in Env$, per le funzioni si ha che:

$$E \vdash fn\ x \Rightarrow M \leadsto (x, M)$$

- Dato $E \in Env$, per le applicazioni si ha che:

$$\frac{E \vdash M \leadsto (x, L) \qquad E'\{(x, N)\} \vdash L \leadsto v}{E \vdash MN \leadsto v}$$

Proposizione 12: Linguaggio *Fun* lazy statico

La semantica lazy statica del linguaggio *Fun* prevede l'aggiunta di alcune regole operazionali:

- L'insieme Env viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Fun \times Env\}$$

- Dato $E \in Env$, per le funzioni si ha che:

$$E \vdash fn\ x \Rightarrow M \leadsto (x, M, E)$$

- Dato $E \in Env$, per le applicazioni si ha che:

$$\frac{E \vdash M \leadsto (x, L, E') \qquad E'\{(x, N, E)\} \vdash L \leadsto v}{E \vdash MN \leadsto v}$$

Osservazione 16

Come per il linguaggio *Exp*, per la sua estensione *Fun* si ha che:

$$Fun \text{ lazy dinamico } \not\equiv Fun \text{ lazy statico}$$

Definizione 31: Espressione ω

Dato il linguaggio *Fun*, definiamo come **espressione omega**, indicata con ω, la seguente espressione:

$$\omega := (fn\ x \Rightarrow xx)(fn\ x \Rightarrow xx)$$

In particolare, l'espressione ω è **invalutabile per qualsiasi semantica**

Esempio:

- Analizziamo l'albero di derivazione di ω utilizzando una semantica eager statica:

$$(*) \qquad \varnothing \vdash x \rightsquigarrow (x, xx, \varnothing) \quad \varnothing \vdash x \rightsquigarrow (x, xx, \varnothing) \quad \frac{(*)}{(x, \{(x, xx, \varnothing)\}) \vdash xx \rightsquigarrow v}$$

$$\frac{\varnothing \vdash fn\ x \Rightarrow xx \rightsquigarrow (x, xx, \varnothing) \quad \varnothing \vdash fn\ x \Rightarrow xx \rightsquigarrow (x, xx, \varnothing) \quad \dfrac{(*)}{(x, \{(x, xx, \varnothing)\}) \vdash xx \rightsquigarrow v}}{\varnothing \vdash (fn\ x \Rightarrow xx)(fn\ x \Rightarrow xx) \rightsquigarrow v}$$

- Notiamo quindi che affinché la valutazione del termine $(x, \{(x, xx, \varnothing)\}) \vdash xx \rightsquigarrow v$ richieda che esso stesso venga valutato, creando così un albero di derivazione infinito.

Lemma 3

Dato il linguaggio *Fun*, si ha che:

$$Fun \text{ eager statico } \not\equiv Fun \text{ lazy statico}$$

$$Fun \text{ eager dinamico } \not\equiv Fun \text{ lazy dinamico}$$

Dimostrazione.

- Consideriamo l'espressione $let\ x = \omega$ in 42. Utilizzando una semantica eager (statica o dinamica), verrebbe richiesta immediatamente la valutazione del termine ω, il quale tuttavia è invalutabile. Utilizzando una semantica lazy (statica o dinamica), invece, il termine ω non verrà mai valutato, restituendo 42 come risultato.

□

Teorema 5: Equivalenze semantiche di *Fun*

Dato il linguaggio *Fun*, **non esistono due semantiche equivalenti**

Proposizione 13: Variabili libere in *Fun*

Dato il linguaggio *Fun*, la funzione free : $Fun \to \mathcal{P}(Var)$ è definita come:

$$\begin{cases} \text{free}(k) = \varnothing \\ \text{free}(x) = \{x\} \\ \text{free}(M+N) = \text{free}(M) \cup \text{free}(N) \\ \text{free}(let\ x = M \text{ in } N) = \text{free}(M) \cup (\text{free}(N) - \{x\}) \\ \text{free}(fn\ x \Rightarrow M) = \text{free}(M) - \{x\} \\ \text{free}(MN) = \text{free}(M) \cup \text{free}(N) \end{cases}$$

2.4.1 *Fun* in Standard ML

La grammatica prevista dal linguaggio *Fun* mostrato fino ad ora è utilizzabile all'interno del **linguaggio SML (Standard Model Language)**, il quale prevede una sintassi leggermente diversa:

- L'operatore $let\ x = M$ in N corrisponde a `let val x = M in N end`
- L'operatore $fn\ x \Rightarrow M$ corrisponde a `fn x => M;`
- L'operatore MN corrisponde a `MN` (potrebbe essere necessario introdurre uno spazio tra `M` ed `N` affinché l'interprete riesca a distinguere i due termini)
- La **curryficazione non è prevista**
- L'espressione va terminata da un **punto e virgola**
- La semantica utilizzata è **eager statica**

Ad esempio, l'espressione:

$$let\ x = 7 \text{ in } ((fn\ y \Rightarrow let\ x = 3 \text{ in } yx)(fn\ z \Rightarrow x))$$

corrisponde al comando:

```
let val x = 7 in (fn y => let val x = 3 in y x end) end;
```

Inoltre, il linguaggio SML permette di assegnare variabili, alle quali possono essere assegnate anche funzioni. Ad esempio, definendo:

```
val id = fn x => x;
```

il seguente comando restituisce 7:

```
id 7;
```

Per utilizzare il linguaggio SML, si consiglia l'uso del programma `smlnj` o dell'emulatore online `SOSML`.

2.5 Lambda calcolo

Definizione 32: Lambda calcolo

Il **lambda calcolo** è un sistema formale in logica matematica per esprimere il calcolo basato sull'**astrazione** e l'applicazione di **funzioni**.

Nella forma più semplice di lambda calcolo, i termini sono costruiti utilizzando solo le seguenti regole:

- Una **variabile** è rappresentata da un carattere (es: x)
- Una **funzione** è rappresentata da una **lambda astrazione**, ossia una stringa composta dal simbolo λ seguito dai parametri della funzione separati con un punto dal corpo della funzione stessa (es: $\lambda x.M$)
- L'**applicazione** di una funzione M ad un argomento N viene rappresentata come $M\ N$

Esempi:

- La lambda astrazione $\lambda x.x + 1$ corrisponde alla funzione $f(x) = x + 1$
- La lambda astrazione $\lambda xy.x + y$ corrisponde alla funzione $f(x, y) = x + y$
- La lambda astrazione $(\lambda x.x)\ 3$ corrisponde all'applicazione della funzione $f(x) = x$ all'argomento 3, restituendo quindi 3
- La lambda astrazione $(\lambda x.x)(\lambda x.x)$ restituisce $\lambda x.x$
- La lambda astrazione $\lambda xy.x(xy)$ applica due volte sull'argomento y la funzione x passata anch'essa come argomento

Osservazione 17: Curryficazione in lambda calcolo

La lambda astrazione $\lambda x_1 \dots x_n.M$ è la contrazione sintattica della seguente lambda astrazione:

$$\lambda x_1.\ \dots \lambda x_n.M$$

Definizione 33: Sostituzione

Definiamo come **sostituzione**, indicata con $M[N/x]$, l'operazione tramite cui all'interno di un'espressione M tutte le occorrenze di una variabile x vengono rimpiazzate con il termine N

Esempi:

- La sostituzione $(xy)[\lambda z.z/x]$ corrisponde a $((\lambda z.z)y)$
- La sostituzione $(fn\ x \Rightarrow xy)[x/y]$ corrisponde a $(fn\ x \Rightarrow xx)$

Osservazione 18: Cattura di variabili

L'operazione di sostituzione potrebbe legare una variabile precedentemente libera o viceversa. Tale fenomeno viene detto **cattura di variabili** ed è necessario accertarsi che esso non si verifichi affinché la sostituzione sia corretta

Esempio:

- L'espressione $(\lambda y.M)[N/x]$ è equivalente all'espressione $\lambda y.(M[N/x])$ solo se $y \notin \text{free}(N)$. Difatti, la sostituzione $(\lambda y.x)[y/x]$ risulta essere "scorretta" in quanto $(\lambda y.y)$ ha una valutazione differente rispetto all'espressione originale

Definizione 34: Alfa conversione

Definiamo come **alfa conversione**, indicata con $\longrightarrow_\alpha$, la regola secondo cui all'interno di una lambda astrazione $\lambda x.M$ ogni occorrenza della variabile x (incluso il parametro) possa essere rimpiazzata dalla variabile y:

$$\lambda x.M \ \longrightarrow_\alpha \ \lambda y.(M[y/x])$$

Esempi:

- Data la lambda astrazione $\lambda x.(xy)$, si ha che:

$$\lambda x.xy \ \longrightarrow_\alpha \ \lambda z.zy$$

- Data la lambda astrazione $\lambda x.x(\lambda z.zw)$, si ha che:

$$\lambda x.x(\lambda z.zw) \ \longrightarrow_\alpha \ \lambda z.z(\lambda z.zw)$$

Definizione 35: Alfa equivalenza

Due lambda astrazioni $\lambda x.M$ e $\lambda y.N$ vengono dette **alfa equivalenti**, indicato con $\equiv_\alpha$, se:

$$\lambda x.M \equiv_\alpha \lambda y.N \iff \lambda x.M \ \longrightarrow_\alpha \ \lambda y.N \wedge \lambda y.N \ \longrightarrow_\alpha \ \lambda x.M$$

Esempi:

- Date le due lambda astrazioni $\lambda x.(xy)$ e $\lambda z.zy$, si ha che:

$$\lambda x.xy \ \longrightarrow_\alpha \ \lambda z.zy \wedge \lambda z.zy \ \longrightarrow_\alpha \ \lambda x.xy \implies \lambda x.xy \equiv_\alpha \lambda z.zy$$

- Date le due lambda astrazioni $\lambda x.x(\lambda z.zw)$ e $\lambda z.z(\lambda z.zw)$, si ha che:

$$\lambda x.x(\lambda z.zw) \ \longrightarrow_\alpha \ \lambda z.z(\lambda z.zw)$$

$$\lambda z.z(\lambda z.zw) \not\longrightarrow_\alpha \lambda x.x(\lambda z.zw)$$

dunque ne concludiamo che:

$$\lambda x.x(\lambda z.zw) \not\equiv_\alpha \lambda z.z(\lambda z.zw)$$

Definizione 36: Beta conversione

Definiamo come **beta conversione** (o *beta riduzione*), indicata con $\longrightarrow_\beta$, la regola secondo cui all'interno di una lambda espressione $(\lambda x.M)N$ ogni occorrenza della variabile x all'interno di M possa essere rimpiazzata dal termine N:

$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

Osservazione 19

La beta riduzione corrisponde esattamente ad singolo **passo computazionale**

Esempi:

- Data la lambda espressione $(\lambda x.xy)(\lambda z.z)$, si ha che:

$$(\lambda x.xy)(\lambda z.z) \longrightarrow_\beta (\lambda z.z)y \longrightarrow_\beta y$$

Definizione 37: Eta conversione

Definiamo come **eta conversione**, indicata con $\longrightarrow_\eta$, la regola secondo cui la lambda espressione $(\lambda x.Mx)$ possa essere rimpiazzata con il termine M solo se $x \notin \text{free}(M)$:

$$x \notin \text{free}(M) \implies \lambda x.Mx \longrightarrow_\eta M$$

Esempi:

- Consideriamo la lambda espressione $\lambda x.(\lambda y.y)x$.
- Poiché:

$$\begin{gathered}\text{free}(\lambda x.(\lambda y.y)x) = \text{free}((\lambda y.y)x) - \{x\} = \\ = [\text{free}(\lambda y.y) \cup \text{free}(x)] - \{x\} = [(\text{free}(y) - \{y\}) \cup \{x\}] - \{x\} = \\ = [(\{y\} - \{y\}) \cup \{x\}] - \{x\} = \varnothing\end{gathered}$$

  dunque $x \notin \text{free}(\lambda x.(\lambda y.y)x)$, si ha che:

$$\lambda x.(\lambda y.y)x \longrightarrow_\eta \lambda y.y$$

Teorema 6: Lambda calcolo turing completo

Il lambda calcolo è **turing completo**, ossia è in grado di simulare una qualsiasi computazione possibile

(*dimostrazione omessa*)

2.5.1 *Fun* vs Lambda calcolo

Avendo trattato le componenti principali del lambda calcolo, possiamo rappresentare quest'ultimo tramite la seguente grammatica:

$$M, N ::= x \mid fn\ x \Rightarrow M \mid MN$$

notiamo come il linguaggio *Fun* corrisponda ad un **sovra-linguaggio** del lambda calcolo stesso. Difatti, essendo il lambda calcolo già **turing completo**, alcuni termini del linguaggio *Fun* risultano "*ridondanti*".

In particolare, le seguenti due espressioni:

$$let\ x = M \text{ in } N \qquad\qquad (fn\ x \Rightarrow N)M$$

risultano essere **operativamente equivalenti**, ossia vengono sempre valutate nello stesso risultato indipendentemente dalla semantica utilizzata (sebbene esse differiscano in termini di "implementazione" delle loro regole operazionali, dunque <u>non</u> sono effettivamente la stessa espressione).

Osservazione 20

La lambda astrazione $\lambda x_1.\ \dots\ .x_n.M$, corrisponde all'espressione:

$$fn\ x_1 \dots x_n \Rightarrow M$$

In modo analogo a Von Neumann, il matematico Church diede una propria definizione alternativa dei **numeri naturali**: il numero $n \in \mathbb{N}$ corrisponde all'applicazione per n volte di un'operazione x su un valore y.

In particolare, notiamo che tale definizione data da Church possa essere espressa in termini di **lambda calcolo**. Ad esempio, il numero naturale 3 corrisponderà alla lambda astrazione $\lambda xy.x(x(xy))$

Proposizione 14: Numeri naturali di Church

I numeri naturali di Church, indicati con $\mathcal{N}_\lambda$, definiti come:

$$0_{\mathcal{N}_\lambda} := \lambda xy.y$$

$$1_{\mathcal{N}_\lambda} := \lambda xy.xy$$
$$2_{\mathcal{N}_\lambda} := \lambda xy.x(xy)$$
$$3_{\mathcal{N}_\lambda} := \lambda xy.x(x(xy))$$
$$\dots$$

dove $\text{succ}_{\mathcal{N}_\lambda} : \mathcal{N}_\lambda \to \mathcal{N}_\lambda : n \mapsto n \cup \{n\}$, soddisfano gli assiomi di Peano

(*dimostrazione omessa*)

Utilizzando la definizione di Church dei numeri naturali, è possibile definire un modello di calcolo **interamente basato sul lambda calcolo** dove ogni operazione possibile è definibile in termini di lambda astrazioni che lavorano sui numeri di Church (i quali a loro volta sono delle lambda astrazioni).

Di conseguenza, potremmo effettivamente ridurre la grammatica dell'intero linguaggio *Fun* in quella del lambda calcolo.

Procediamo quindi definendo i numeri di Church all'interno del linguaggio *Fun*:

- `zero` $:= fn\ x \Rightarrow fn\ y \Rightarrow y$ oppure $fn\ xy \Rightarrow y$
- `one` $:= fn\ x \Rightarrow fn\ y \Rightarrow xy$ oppure $fn\ xy \Rightarrow xy$
- `two` $:= fn\ x \Rightarrow fn\ y \Rightarrow x(xy)$ oppure $fn\ xy \Rightarrow x(xy)$
- `three` $:= fn\ x \Rightarrow fn\ y \Rightarrow x(x(xy))$ oppure $fn\ xy \Rightarrow x(x(xy))$
- ...

Definiamo inoltre una funzione `eval` in grado di convertire un numero di Church nel suo equivalente nei numeri naturali:

$$\texttt{eval} := fn\ z \Rightarrow z(fn\ x \Rightarrow x+1)\ 0$$

Ad esempio, l'espressione `eval two` viene valutata come:

$$\begin{gathered}
\texttt{eval two} \ \longrightarrow_\beta \\
\{fn\ z \Rightarrow z(fn\ x \Rightarrow x+1)\ 0\}[fn\ x \Rightarrow fn\ y \Rightarrow x(xy)] \ \longrightarrow_\beta \\
\{[fn\ x \Rightarrow fn\ y \Rightarrow x(xy)](fn\ x \Rightarrow x+1)\ 0\} \ \longrightarrow_\beta \\
\{[fn\ y \Rightarrow (fn\ x \Rightarrow x+1)((fn\ x \Rightarrow x+1)y)]\ 0\} \ \longrightarrow_\beta \\
[(fn\ x \Rightarrow x+1)\{(fn\ x \Rightarrow x+1)\ 0\}] \ \longrightarrow_\beta \\
[(fn\ x \Rightarrow x+1)\ 1] \ \longrightarrow_\beta \\
2
\end{gathered}$$

A questo punto, definiamo la funzione `succ` che restituisce il successore del numero di Church dato in input:

$$\texttt{succ} := fn\ z \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow zx(xy))$$

Ad esempio, l'espressione `succ one` viene valutata come:

$$\begin{gathered}
\texttt{succ one} \ \longrightarrow_\beta \\
[fn\ z \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow zx(xy))](fn\ x \Rightarrow fn\ y \Rightarrow xy) \ \longrightarrow_\beta \\
[fn\ x \Rightarrow fn\ y \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow xy)x(xy)] \ \longrightarrow_\beta \\
[fn\ x \Rightarrow fn\ y \Rightarrow (fn\ y \Rightarrow xy)(xy)] \ \longrightarrow_\beta \\
fn\ x \Rightarrow fn\ y \Rightarrow x(xy) \ \longrightarrow_\beta \\
\texttt{two}
\end{gathered}$$

Successivamente, definiamo le seguenti ulteriori funzioni matematiche:

- La funzione `sum` che somma due numeri di Church:

$$\texttt{sum} := fn\ z \Rightarrow fn\ w \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow zx(wxy))$$

  oppure:

$$\texttt{sum} := fn\ z \Rightarrow fn\ w \Rightarrow z\ \texttt{succ}\ w$$

- La funzione `prod` che moltiplica due numeri di Church:

$$\texttt{prod} := fn\ z \Rightarrow fn\ w \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow z(wx)y)$$

  oppure:

$$\texttt{prod} := fn\ z \Rightarrow fn\ w \Rightarrow z(\texttt{sum}\ w)\ \texttt{zero}$$

- La funzione `power` che eleva un numero di Church ad un altro numero di Church:

$$\texttt{prod} := fn\ z \Rightarrow fn\ w \Rightarrow wz$$

Oltre ai numeri naturali, il lambda calcolo ci permette di descrivere anche la **logica booleana** di Church, dove i due valori `True` e `False` sono definiti come:

$$\texttt{True} := fn\ x \Rightarrow fn\ y \Rightarrow x$$

$$\texttt{False} := fn\ x \Rightarrow fn\ y \Rightarrow y$$

Come per i numeri di Church, definiamo una funzione `evalBool` in grado di convertire un booleano di Church in nel suo equivalente booleano:

$$\texttt{evalBool} := fn\ z \Rightarrow z\ true\ false$$

dove *true* e *false* sono i normali valori booleani

Infine, definiamo i seguenti operatori logici:

- L'operatore `ITE` (abbreviativo di `If-Then-Else`) che dati una condizione z e due booleani di Church u, v, valuta u se z è *true* oppure valuta v se z è *false*:

$$\texttt{ITE} := fn\ z \Rightarrow fn\ u \Rightarrow fn\ v \Rightarrow z\ u\ v$$

- L'operatore `If` che dati una condizione z ed un booleano di Church u, valuta u se z è *true*:

$$\texttt{If} := fn\ z \Rightarrow fn\ u \Rightarrow z\ u\ \texttt{True}$$

- L'operatore `Not` che restituisce il negato di un booleano di Church:

$$\texttt{Not} := fn\ z \Rightarrow fn\ x \Rightarrow fn\ y \Rightarrow z\ y\ x$$

- L'operatore `Or` che restituisce l'or logico tra due booleani di Church:

$$\texttt{Or} := fn\ z \Rightarrow fn\ w \Rightarrow \texttt{If}(\texttt{Not}\ z)w$$

- L'operatore `And` che restituisce l'and logico tra due booleani di Church:

$$\texttt{And} := fn\ z \Rightarrow fn\ w \Rightarrow \texttt{Not}(\texttt{If}\ z\ (\texttt{Not}\ w))$$

Di seguito viene fornito il codice SML per poter lavorare con il modello di calcolo appena definito:

```
(* Numeri di Church *)

val zero = fn x => fn y => y;
val one = fn x => fn y => x y;
val two = fn x => fn y => x(x y);
val three = fn x => fn y => x(x(x y));

val eval = fn z => z (fn x => x+1) 0;

val succ = fn z => fn x => fn y => z x (x y);
val sum = fn z => fn w => fn x => fn y => z x (w x y);
val prod = fn z => fn w => fn x => fn y => z (w x) y;
val power = fn z => fn w => w z;

(* Booleani di Church *)

val True = fn x => fn y => x;
val False = fn x => fn y => y;

val evalBool = fn z => z true false;

val ITE = fn z => fn u => fn v => z u v;
val If = fn z => fn u => z u True;

val Not = fn z => fn x => fn y => z y x;
val Or = fn z => fn w => If (Not z) w;
val And = fn z => fn w => Not (If z (Not w));

(* Esempi *)

eval (sum (power two three) (prod two three));
evalBool (And (ITE True False True) False);
```