Capitolo 1

Linguaggi e macchine astratte

Nel corso della nostra trattazione cercheremo di dare risposta ad alcune domande fondamentali:

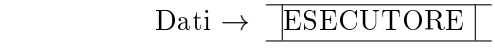
- Quali istruzioni esegue un elaboratore?
- Quali problemi può risolvere un elaboratore?
- Esistono problemi che un elaboratore non può risolvere?
- Che ruolo ha il linguaggio di programmazione?

Introduciamo alcune definizioni:

- **algoritmo**: è una sequenza **finita** di passi che risolve **in tempo finito** una *classe* di problemi.
- **Codifica**: è la scrittura di un algoritmo attraverso un insieme ordinato di **frasi** (**istruzioni**) di un **linguaggio di programmazione** che specificano le azioni da compiere.
- **Programma**: è un testo scritto in accordo alla **sintassi** e alla **semantica** di un linguaggio di programmazione.

  Nota: il programma spesso sarà un algoritmo, ma può anche non esserlo e ciò accade quando un programma non termina, cioè quando prevede una sequenza di passi infinita. Un esempio è dato dai sistemi operativi che restano in esecuzione fino a quando non è l'utente a stopparli. In generale possiamo dire che non sono algoritmi i programmi che devono *agire*, più che *calcolare un risultato*.

Escludiamo per ora l'eccezione appena presentata e consideriamo quindi un programma come la formulazione testuale di un algoritmo che risolve un dato problema (o meglio una data classe di problemi). Partendo dai dati in ingresso, svolgendo in sequenza tutte le azioni previste dall'algoritmo, si ottiene il risultato dell'elaborazione. Ciò presuppone l'esistenza di un'entità che effettivamente esegua i passi, quindi di un **automa esecutore**. Definiamo l'automa esecutore come una macchina astratta capace di eseguire le azioni specificate dall'algoritmo. L'automa esecutore riceve in ingresso oltre ai dati anche l'algoritmo che dovrà essere espresso in qualche linguaggio. L'automa dovrà perciò essere in grado di interpretare il linguaggio.



Affinché possa essere realizzabile l'automa deve soddisfare i seguenti vincoli:

- se l'automa dev'essere composto da un **numero finito di parti**;
- ingresso e uscita devono essere denotabili attraverso un **insieme dinito di simboli**.

Non abbiamo ancora detto nulla sull'automa in sè, su come dev'essere progettato, su come debba **computare**. Esistono vari approcci, vari **sistemi formali** che definiscono in modo diverso la **computabilità**. Riepiloghiamo di seguito i principali sistemi formali:

- **gerarchia di macchine astratte**: dall'automa a stati finiti alla macchina di Turing ed è il modello di computazione su cui ci concentreremo noi;
- **approccio funzionale**: basato sul concetto di funzione matematica;
- **sistemi di riscrittura**: che descrivono l'automa come un insieme di regole di riscrittura (o inferenza) che trasformano frasi in altre frasi.

1.1 La gerarchia di macchine

La gerarchia di macchien è così composta (indicando le macchine dalla meno potente alla più potente:

1. reti combinatorie;
2. automi a stati finiti;
3. macchina a stack;
4. macchine di Turing.

Diversi macchine hanno diverse capacità risolutive. Se neanche la più potente riesce a risolvere un problema, allora potrebbe non essere risolubile.

1.1.1 La macchina base, combinatoria

Si trata di una rete definita formalmente dalla tripla: `I, O, mfn` dove gli elementi che la compongono sono:

- `I`: insieme **finito** dei simboli di ingresso;
- `O`: insieme **finito** dei simboli di uscita;
- `mfn: I -> O`: funzione macchina

Esempi di queste reti sono le **porte logiche**. Se prendessimo come esempio una porta logica a due ingressi, avremmo $I = \{\{0,1\} \times \{0,1\}\}$ e $O = \{0,1\}$.
La macchina combinatoria è una rete semplicissima e in quanto tale presenta dei limiti:

- risolvere problemi con tale macchina comporta l'**enumerazione completa** di tutte le possibili configurazioni di ingresso ed indicare il corrispondente valore di uscita;
- si tratta di un dispositivo puramente combinatorio, quindi **inadatto per problemi che richiedano memoria interna**.

1.1.2 L'automa a stati finiti

Abbiamo detto che uno dei limiti delle macchine base è l'impossibilità di risolvere problemi che richiedano memoria. Con gli automi a stati finiti si introduce proprio il concetto di memoria grazie ad un numero finito di **stati interni**.
L'automa a stati finiti è definito dalla quintupla: `I, O, S, mfn, sfn`. Questi simboli indicano:

- `I`: come per le macchine base è l'insieme finito dei simboli di ingresso;
- `O`: come per le macchine base è l'insieme dinito dei simboli di uscita;
- `S`: è l'insieme finito degli **stati**;
- `mfn`: $I \times S \to O$ è la funzione di macchina;
- `sfn`: $I \times S \to S$ è la **funzione di stato futuro**.

L'uscita dipende ora dallo stato, pertanto si evince la capacità della macchina di tenere in memoria informazioni che, al pari dei dati in ingresso, influenzano il risultato della computazione.
Anche l'automa a stati finiti, ovviamente, ha un limite: ha **memoria finita**, quindi è inadatto a risolvere problemi che non consentono di limitare a priori le sequenze di ingresso da memorizzare. Un problema che possono risolvere le macchine a stati finiti è quello del bilanciamento delle parentesi, tuttavia possono farlo soltanto se il numero di parentesi non eccede il massimo numero previsto in fase di progettazione dell'automa. Si dovrebbe quindi conoscere a priori il massimo numero di parentesi che si possono verificare per poter progettare l'automa senza rischiare malfunzionamenti dovuti a input eccessivamente lunghi.

1.1.3 La macchina di Turing

Saltiamo per il momento le macchine a stack e concentriamoci sul livello immediatamente seguente nella gerarchia: le macchine di Turing, l'automa più potente a disposizione.
La macchina di Turing introduce un supporto di memorizzazione esterna, un nastro illimitatamente espandibile, ed è definita dalla quintupla: `A, S, mfn, sfn, dfn`. Questa volta gli alfabeti di ingresso e di uscita (sempre finiti) sono considerati coincidenti ed indicati con `A`. `mfn` ed `sfn` sono le funzioni che abbiamo già visto prima (uscita e stato futuro), mentre la `dfn` è la funzione di indirizzamento sul nastro: $A \times S \to D = \{Left, Right, None\}$.
Risolvere un problema con la MdT richiede quindi di:

1. definire un'opportuna rappresentazione dei dati iniziali sul nastro;
2. definire la parte di controllo (cioè le tre funzioni.

Un esempio di problema risolvibile tramite le macchine di Turing è il riconoscimento dei palindromi.
Secondo **Church-Turing** non esiste una macchina capace di risolvere una classe di problemi piú ampia rispetto alla macchina di Turing, pertanto se un problema non è risolvibile con essa probabilmente non può essere risolto in altro modo.
Se l'algoritmo non è cablato all'interno della MdT, ma va recuperato dal nastro, allora si ha una macchina **general purpose**, anche detta **Macchina di Turing Universale** (UTM). Si tratterebbe quindi di una macchina programmabile. L'UTM sarebbe un interprete di linguaggio in quanto interpreterebbe il programma presentato sul nastro (che è espresso in un dato linguaggio). La UTM rivestirebbe quindi il ruolo dell'automa esecutore presentato prima, svolgendo soltanto le operazioni di `fetch`, `decode` ed `execute`.

UTM e macchina di Von Neumann

Volendo fare un paragone fra l'UTM e la macchina di Von Neumann, potremmo individuare le seguenti corrispondenze:

- leggere/scrivere un simbolo dal/sul nastro corrisponde a leggere/scrivere dalla/sulla memoria;
- transitare in un nuovo stato interno corrisponde ad una nuova configurazione dei registri della CPU;
- spostarsi sul nastro corrisponde a scegliere una cella di memoria su cui operare.

Dunque effettivamente si potrebbe dire che c'è una forte corrispondenza fra le due macchine, tuttavia l'interazione con il mondo esterno nella UTM manca completamente. L'UTM opera soltanto da e verso il nastro. La Universal Turing Machine è quindi una **macchina di pura computazione**.
Computazione e **interazione** sono dimensioni ortogonali che vengono modellate separatamente ed espresse (potenzialmente) da linguaggi distinti, il primo dei quali è detto **linguaggio di computazione**, il secondo di **coordinazione**. Mentre il linguaggio di computazione definisce le primitive per elaborare le informazioni, il linguaggio di coordinazione contiene le primitive di input/output di informazioni da e verso il mondo esterno. All'interno di questo linguaggio troviamo il **linguaggio di comunicazione** che definisce quali informazioni vadano trasmesse.

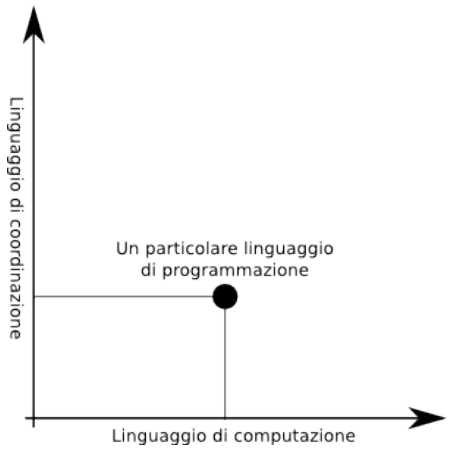


Figura 1.1: Computazione e interazione

1.1.4 Il Push Down Automaton

È un automa più limitato rispetto alla MdT (e infatti la precede nella gerarchia) e la differenza consiste nella presenza di una memoria più limitata.