2 Linguaggi, interpreti e calcolatori

Indice

2.1 Introduzione

Anche se potrebbe sembrare che l'informatica abbia poco a che fare con le materie umanistiche, basta pensare che esiste una parola in comune tra questi due campi dello scibile umano, apparentemente indipendenti: la parola in questione è "linguaggio". Infatti fondamentalmente i *linguaggi di programmazione* sono dei costrutti linguistici mediante i quali gli esseri umani scrivono, più o meno ad alto livello, le operazioni che deve eseguire la macchina rispecchiando l'implementazione di un algoritmo. Ma questo modo di interpretare le istruzioni che deve eseguire il compilatore, il linguaggio di programmazione appunto, non è direttamente comprensibile alla macchina: è necessario prima o utilizzare un programma che interpreti le sue istruzioni (**interprete**) in modo comprensibile per la macchina, oppure di un traduttore (**compilatore**) che converta il sorgente una volta per tale macchina: ma quale linguaggio può comprendere la macchina e come essa lo interpreta?

Definire come la macchina interpreti il suo codice è compito di un corso di Architettura dei Calcolatori, che va oltre alla trattazione della materia in questione (**realizzazione dell'interprete hardware**): si può pensare tuttavia che il processore, che notoriamente è la parte che si preoccupa della interpretazione delle istruzioni in azione, implementa a livello di circuiteria o tramite un traduttore software in una memoria di sola lettura il metodo di tradurre gli impulsi che devono essere inviati alla macchina. Esiste tuttavia un livello superiore detto ISA, che costituisce il livello di linguaggio macchina verso il quale vengono compilati i programmi in modo che il codice assembly scritto dal programmatore sia più "user-friendly" rispetto al linguaggio immediatamente sottostante: questo livello ed

i successivi che possono essere formati sopra di questo per costruzione sono detti **livelli di astrazione**, in quanto ad ogni livello si può supporre che la macchina sia in grado direttamente di operare quelle istruzioni che sono fornite dal programmatore: possiamo quindi considerare all'interno di un normale computer il livello ISA come quello M_0, ovvero quello oggetto verso il quale vengono tradotto un qualsiasi linguaggio di alto livello, e che quindi comprende il linguaggio $\mathcal{L}_0$: da notare quindi che un qualsiasi macchina fisica, così come una macchina software può essere realizzata in modo tale che interpreti direttamente il linguaggio voluto (es: creare un computer il cui codice oggetto sia direttamente il programma sorgente scritto in linguaggio Java), ma questo pone dei seri limiti alla realizzazione dei livelli successivi di astrazione, in quanto, ad esempio, tale linguaggio non è totalmente adatto ad operare sulla macchina ospite, in quanto ad esempio non presenta alcuni oggetti come i puntatori o di permettere di essere interfacciabile con altri linguaggi di programmazione, quali ad esempio OCaml o Assembly; ed inoltre sarebbe molto scomodo scrivere anche un sistema operativo interamente in questo linguaggio, in quanto non permette un controllo diretto della macchina.

Possiamo tuttavia considerare una qualsiasi macchina astratta M_A che esegue direttamente un linguaggio $\mathcal{L}_A$, sulla quale possiamo scrivere in una astrazione della macchina che crea un livello di linguaggio $\mathcal{L}_{A+1}$. Possiamo quindi considerare che ogni livello successivo utilizza delle peculiarità presenti ad un livello inferiore, in modo da poter rendere più semplici alcune funzionalità il cui accesso sarebbe molto impegnativo ad un livello inferiore: inoltre livelli superiori di astrazione possono nascondere alcune specifiche della macchina sottostante, in modo da evitare all'utente l'accesso diretto ad alcune istruzioni. Un esempio può essere fatto con i protocolli utilizzati nelle Reti di Calcolatori o nei Sistemi Operativi.

Dobbiamo inoltre precisare che ad ogni macchina corrisponde uno ed un solo linguaggi da essa interpretabile, mentre per quanto riguarda un linguaggio, esso può essere istanziato in qualsiasi macchina.

2.2 Compilatori ed interpreti

Definiamo un interprete $\mathcal{I}_{\mathcal{L}_1}^{\mathcal{L}_2}(P^{\mathcal{L}_2}, \mathcal{D})$ un programma scritto in un linguaggio $\mathcal{L}_2$ che prende in input un programma scritto nel linguaggio $\mathcal{L}_1$, facendolo eseguire sui dati $\mathcal{D}$. In questo modo si può dire che:

$$\mathcal{I}_{\mathcal{L}_1}^{\mathcal{L}_2} : \mathcal{L}_1 \times \mathcal{D}ata \rightarrow \mathcal{D}ata$$

$$\mathcal{I}_{\mathcal{L}_1}^{\mathcal{L}_2}(P^{\mathcal{L}_2}, \mathcal{D}) = P^{\mathcal{L}_2}(\mathcal{D})$$

Ovvero l'interprete esegue il programma come se fosse eseguito direttamente sulla macchina nella quale è eseguito direttamente il suo linguaggio $\mathcal{L}_1$. Bisogna quindi sottolineare che detto interprete può essere realizzato sia all'interno della circuiteria di una macchina fisica (**livello hardware**) oppure non fisicamente ma mediante la realizzazione di un algoritmo in un qualsiasi linguaggio, ovvero a **livello software**, in ogni caso si può generalizzare che implementare un nuovo linguaggio significa progettare una macchina che abbia quel linguaggio come nuovo linguaggio macchina. Vediamo brevemente pregi e difetti di questa forma di implementazione:

- A livello hardware si ha un interprete più efficiente dal punto di vista di tempo di esecuzione del programma interpretando, ma risulta parecchio costoso creare da nuovo una nuova apparecchiatura quando si è notato che questa non funziona secondo le specifiche volute. Questo è molto utile quando si vuole operate un tempo reale.
- A livello software invece, come è facilmente intuibile, si ha una esecuzione del programma in un modo molto più rallentata ma è molto più pratico adattarsi a nuovi cambiamenti ricompilando il codice sorgente. In questo modo creiamo un livello di astrazione di una macchina $M_{\mathcal{L}}$ che utilizza i costrutti del linguaggio $\mathcal{L}-1$ al di sopra della macchina che stiamo utilizzando, volendo che questa sia in grado di interpretare il nostro nuovo linguaggio macchina.

Sia che detta realizzazione avvenga a livello hardware, sia che essa avvenga a livello software, si parla comunque di realizzazione di una **macchina astratta** $M_{\mathcal{L}}$ che è in grado di memorizzare al suo interno un linguaggio $\mathcal{L}$ detto **linguaggio macchina** e di interpretarlo, in quanto al suo interno è presente una *memoria* ed un *interprete*. Tale macchina astratta deve essere però in grado di soddisfare alcune caratteristiche fondamentali:

1. Poter *elaborare i dati primitivi*, ovvero dobbiamo essere in grado di effettuare delle operazioni su questi dati e di poterli salvare nelle memorie della nostra macchian per poterli (es.) recuperare per le computazioni successive.
2. Una macchina deve essere in grado di *controllare la sequenza di operazioni* che vengono svolte dalla macchia stessa: ad esempio nella realizzazione della macchina astratta di Tanenbaum (per semplicità tratteremo solamente del Mic-1), esiste un registro di memoria detto Program Counter che consente di tener traccia dell'operazione successiva che deve essere eeseguita.
3. Poter accedere alla memoria trasferendo, modificando e leggendo dati: questo è molto importante per decidere dove trasferire questi dati, che possono eventualmente essere modificati in seguito ad una elaborazione.

Per rendere inoltre possibile l'esecuzione in ogni momento del linguaggio definito all'interno di una macchina M_A senza doverlo ogni volta interpretare, esistono dei programmi chiamati **compilatori**: sono dei programmi scritti in un linguaggio $\mathcal{L}_3$ e che traducono un linguaggio $\mathcal{L}_1$ in uno $\mathcal{L}_2$, indicato da $\mathcal{L}_1 \to \mathcal{L}_2$ ovvero:

$$\mathcal{C}^{\mathcal{L}_3}_{\mathcal{L}_1\to\mathcal{L}_2} : \mathcal{L}_1 \to \mathcal{L}_2$$
$$\mathcal{C}^{\mathcal{L}_3}_{\mathcal{L}_1\to\mathcal{L}_2}(P^{\mathcal{L}_1}) = P^{\mathcal{L}_1} \qquad P^{\mathcal{L}_1}(\mathcal{D}) = P^{\mathcal{L}_2}(\mathcal{D})$$

Mentre si può verificare che un compilatore per ogni programma preso come input, dovrà sempre terminare e quindi è una *funzione totale* che mappa tutti i programmi in altri programmi in quanto la traduzione è un processo di traduzione tra stringhe, l'interprete è invece una funzione parziale in quanto esisteranno sicuramente dei programmi quali i while true che non forniranno alcun risultato utile[1]. Non esistono tuttavia delle soluzioni prettamente compilative o soluzioni prettamente interpretative, in quanto nella maggior parte dei casi avvengono delle situazioni intermedie; di solito, all'esecuzione (e quindi interpretazione) del programma precedentemente compilato si ha la seguente situazione:

[1] Una funzione parziale infatti, essendo definita unicamente per alcuni dati, riporterà per altri un risultato indefinito, ovvero un **while true**

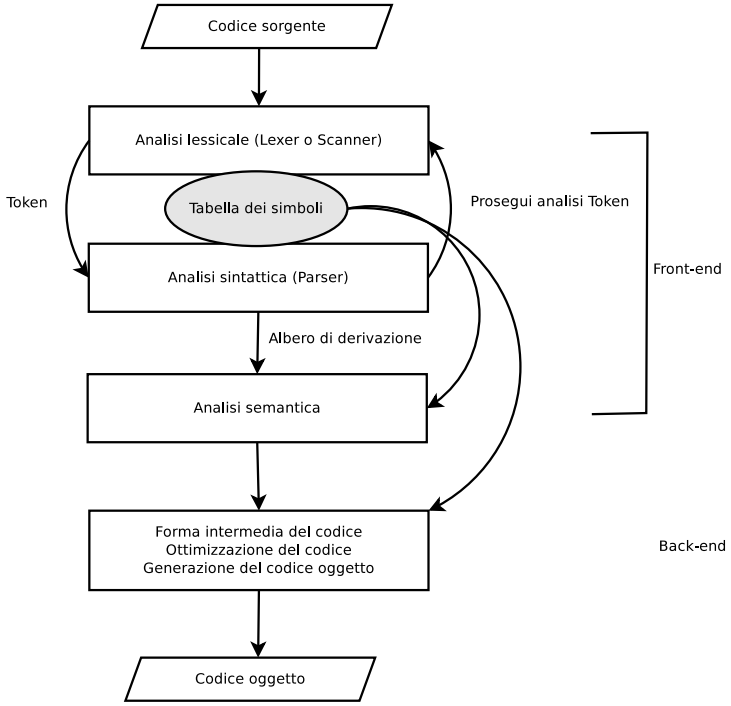

Figura 2.2.1 *Schematizzazione delle varie fasi di compilazione*

$$Real(P^{\mathcal{L}_1}, \mathcal{D}) = \mathcal{I}^{\mathcal{L}_A}_{\mathcal{L}_2}(\mathcal{C}^{\mathcal{L}_A}_{\mathcal{L}_1 \to \mathcal{L}_2}(P^{\mathcal{L}_1}), \mathcal{D})$$

Se il linguaggio $\mathcal{L}_2$ è praticamente simile al linguaggio $\mathcal{L}_A$, allora si ha a che fare con una soluzione di **implementazione compilativa** (come nel caso della compilazione in codice ISA che si preoccuperà di fornire una interpretazione successiva in codice macchina) dove in realità l'interprete non è nient'altro che un *supporto a RunTime* (abbreviato anche come SRT), altrimenti si ha di fronte ad una situazione di **implementazione interpretativa** (come avviene nel caso dell'esecuzione di un interprete del Java ByteCode), quando ovvero la macchina intermedia è effettivamente presente: questa quindi si dovrà in seguito preoccupare di dare alla macchina reale una realizzazione con costrutti simili a quelli che lei possa implementare; questa soluzione in genere viene apportata quando si hanno necessità di portabilità del codice, in modo che questo, una volta compilato, possa girare su diverse macchine progettate con sistemi differenti.

2.2.1 Struttura generale di un compilatore

Seguiamo ora la Figura 2.2.1 nella pagina successiva per vedere, nel suo complesso, come è strutturato un compilatore, e da quali parti questo è formato: noi nello specifico tratteremo dettagliatamente unicamente la parte concernente il **front-end**, che si preoccupa dell'acquisizione in input del codice sorgente, per poi convertirlo in codice oggetto (restituito nel back-end, dove appunto viene fornito l'output della elaborazione), avendo però prima controllato che il sorgente in questione contenga tutte stringhe legali tramite riconoscitori successivi.

- Il primo passaggio di conversione (**analisi lessicale**) è quello della conversione di un insieme di caratteri che vengono identificati con un token, ovvero in unità logicamente significative per la fase successiva, tramite la generazione di automi a stati finiti deterministici che vengono utilizzati per il riconoscimento di tali token da una formula generica specificata. Questi token vengono identificati con un nome univoco ed, eventualmente, possono contenere delle informazioni sulla stringa corrispondente, quale ad esempio il suo valore. *Un errore che può essere identificato a questo stadio di compilazione è il trovare una stringa con un carattere che non appartiene all'alfabeto o comunque al set di caratteri specificato*.
- Tuttavia le grammatiche regolari, quali sono quelle impiegate nell'analisi lessicale, non sono sufficienti per poter descrivere appieno tutti i linguaggi di programmazione: per questo per l'analisi sintattica utilizza si una grammatica libera, ottenendo quindi dall'insieme dei token conformi alla grammatica un albero di derivazione nelle cui foglie sono presenti i suddetti token: con il Pumping Lemma per i linguaggi regolari infatti si vedrà infatti che esistono questi linguaggi "ulteriori" ai regolari, che sono più espressivi in quanto, come vedremo, per essere implementati bisogna prevedere negli automi implementativi la capacità di saper contare (quelli che vedremo essere gli "automi a pila" o **PDA**). *Un errore rilevabile a questo livello di compilazione è l'errata disposizione dei lessemi*.
- L'albero di derivazione così generato viene poi sottoposto, all'interno dell'**analisi semantica**, ai controlli relativi ai vincoli contestuali, dove vengono controllate le *dichiarazioni, i tipi, il numero di parametri delle funzioni*, aggiungendo ai token dell'albero delle informazioni aggiuntive in modo da poter attuare questa forma di controllo. All'interno dei controlli di semantica statica, avviene (es.) *la correzione del codice val = 3; nel linguaggio Java se la variabile non è stata precedentemente dichiarata*. Per quanto riguarda il controllo invece della semantica di un programma, si possono utilizzare due approcci:
  - Con l'approccio *denotazionale*, si cerca di rendere il programma in una notazione funzional-logico-matematica, traducendolo come una funzione che prende un valore in input per restituirne un altro in output
  - Con l'approccio *operazionale*, si fa riferimento ad un formalismo ad un basso livello che è interpretato dalla macchina astratta che esegue quel dato codice

Le definizioni delle forme intermedie, delle ottimizzazioni del codice e della sua generazione sono delle tematiche che dovrebbero essere risolte *ad hoc* in base al linguaggio verso il quale si intende compilare.

2.3 Astrazione

Con **astrazione** intendiamo il meccanismo mediante il quale si identifica, tra tutte le carateristiche presenti, solamente quelle di nostro esclusivo interesse, in modo da potersi focalizzare su queste ed ingorando le altre che vengono messe in secondo piano.

Un primo punto dove possiamo notare questo termine è all'interno della dicitura di *macchina astratta* $\mathcal{M}_{\mathcal{L}}$, con il cui termine intendiamo un qualsiasi insieme di strutture dati e di algoritmi che permettano di memorizzare ed eseguire programmi scritti in $\mathcal{L}$ per la quale è progettata. In seguito possiamo notare che questo termine è utilizzato anche per indicare i diversi livelli di macchine che possono essere implementate l'una sull'altra: ciò è vero perchè abbiamo che *implementare un linguaggio di programmazione* non è nient'altro che creare una macchina astratta che abbia tale linguaggio come linguaggio macchina. Inoltre, se realizziamo tali macchine tramite una realizzazione via software, otteniamo una maggiore flessibilità: oltre al fatto che possiamo riscrivere la sua architettura con costi più limitati

rispetto alla riprogettazione di una circuiteria hardware, possiamo creare diversi livelli di incapsulamento di macchine, partendo da macchine di strati inferiori fornenti un dato linguaggio e dati servizi, per poter ampliare le caratteristiche del primo e le funzionalità dei secondi, in modo da creare più livelli di *interpretazione*

Tuttavia, non è solo all'interno delle macchine astratte che abbiamo l'utilizzo dei meccanismi di astrazione: per quanto riguarda più direttamente i linguaggi di programmazione, abbiamo infatti le seguenti forme di astrazione, entrambe presenti sia:

astrazione sul controllo questa forma di astrazione ha lo scopo di nascondere dettagli procedurali: un esempio sono le **funzioni**, con le quali si nascondono eventuali dettagli nella scrittura del codice, e limitando la conoscenza del contesto al passaggio dei parametri, e la gestione delle **eccezioni**: in quanto queste ultime sono eventi particolari che non possono o non devono essere gestite dal normale flusso di istruzioni, è necessario introdurre un costrutto che permetta il controllo di questi eventi. Si può avere un'altra forma di astrazione sul controllo consentendo l'uso di **linguaggi di ordine superiore**: si definiscono tali quei linguaggi che accettano funzioni di ordine superiore, ovvero che accettano un'altra funzione come parametro o che la possono restituire.

astrazione sui dati in questo caso invece è lo stesso tipo di dato a costituire una forma di astrazione, in quanto non ci interessa in questo contesto come questo sia stato implementato, ma è possibile interagire direttamente con il valore memorizzato tramite una **interfaccia** esterna, che ha il preciso compito di fornire una capsula (occultando quindi l'informazione: si effettua *information hiding*) che, grazie alle operazioni che fornisce (es. i *metodi* delle classi), è possibile accedere o manipolare indirettamente tali valori, i quali opereranno direttamente a livello di *implemetazione* del tale tipo di dato. Un esempio di tali tipi di dato sono appunto i **tipi di dato astratti**, che forniscono un primo passo verso il paradigma orientato agli oggetti. Anche se non è possibile conoscere direttamente come questi tipi di dato siano implementati, è possibile conoscere il loro funzionamento dalla descrizione della loro *specifica*, che appunto descrive la semantica delle operazioni possibili

2.4 Grammatiche e Linguaggi

Una grammatica è costituita da un alfabeto (Σ), ovvero da alcuni simboli che vengono considerati come "unitari" e di base: esso viene utilizzato per definire il piano del *lessico* (ovvero l'insieme delle parole corrette), dal quale ci si può allargare all'ambito *sintattico* (ovvero come disporre tali parole correttamente).

▸ Definizione 2.4.1 (Grammatica (libera dal contesto)). Una grammatica [2] può essere descritta formalmente come una tupla nella forma:

$$\mathcal{G} = (\mathcal{N}t, \mathcal{T}, \mathcal{R}, \mathcal{S})$$

[2] Più precisamente, in questo caso tratteremo unicamente delle grammatiche **libere dal contesto**. Esistono infatti anche altri tipi di grammatiche, che invece dipendono dal nostro contesto, che hanno tutte le produzioni nella forma:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

con $\alpha, \beta \in \mathcal{T}$, $A \in \mathcal{N}t$, $\gamma : \{\mathcal{T} \cup \mathcal{N}t\}^* \neq \varepsilon$ e inoltre $S \rightarrow \varepsilon$ è ammessa sse. S non appare in alcun lato destro delle produzioni fornite. In particolare, queste regole di grammatica vengono utilizzate in fase di analisi di *semantica statica*, ad esempio per controllare che una variabile a cui si sta effettuando un assegnamento, sia stata dichiarata precedentemente, oppure si vuole che le sia stato assegnato un valore del tipo di dato corretto.