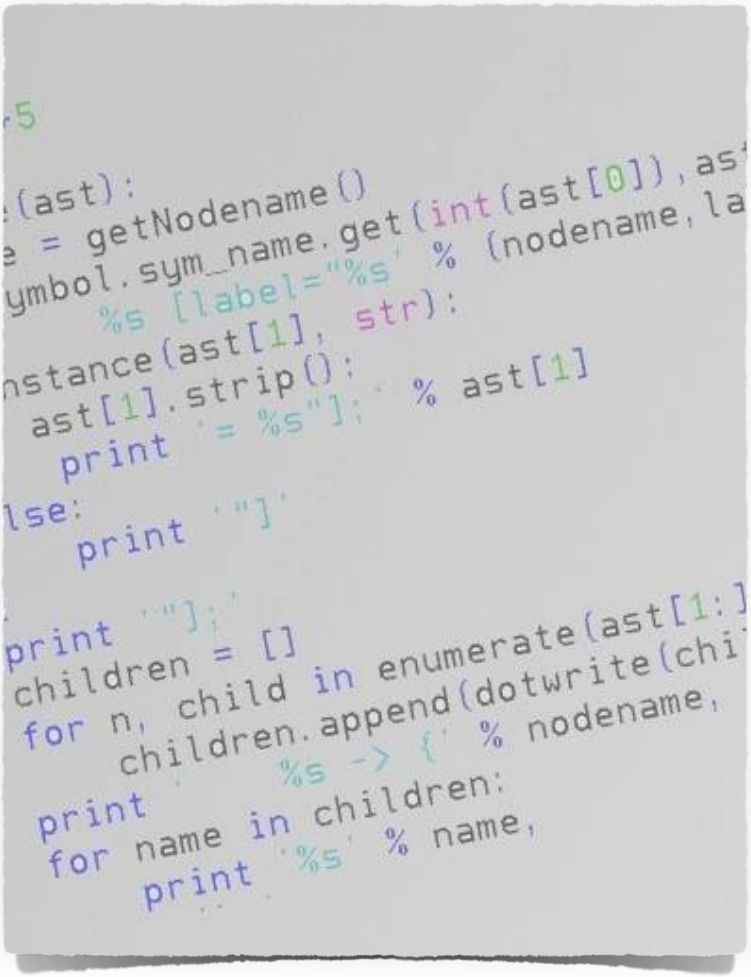
Isabella Mastroeni

LINGUAGGI

Cap. 1 INTRODUZIONE



Motivazione

PERCHÉ STUDIARE LINGUAGGI?

- **Migliorare le proprie capacità nel progettare algoritmi**
- **Per *studiare* e *progettare* nuovi linguaggi**
- **Per individuare il linguaggio *più adeguato* ad una applicazione da sviluppare**

- Per migliorare le capacità di esprimere idee in forma algoritmica. Infatti i programmatori, nel processo di sviluppo del SW, sono spesso vincolati ai limiti del linguaggio. Il linguaggio nel quale sviluppano pone dei limiti al tipo di strutture di controllo, strutture dati e astrazioni che possono usare, questo implica dei limiti agli algoritmi implementati. *La consapevolezza di una più vasta varietà di linguaggi può ridurre queste limitazioni*;

- Per migliorare il proprio bagaglio culturale che permette di scegliere appropriatamente un linguaggio. Infatti solitamente i programmatori tendono ad usare sempre il linguaggio con cui hanno maggiore familiarità indipendentemente dal progetto da sviluppare. *Programmatori più consapevoli potrebbero scegliere un linguaggio con caratteristiche più adatte al problema da risolvere;*

- Per migliorare la capacità di studiare o progettare nuovi linguaggi o nuovi costrutti. Imparare linguaggi nuovi può essere lungo e difficile, soprattutto per chi conosce pochi linguaggi e chi non ha mai studiato i concetti dei linguaggi di programmazione in generale. *Una volta acquisiti questi concetti diventa più facile capire come questi concetti sono incorporati nel linguaggio che si sta studiando;*

- Per comprendere meglio il significato dell'implementazione dei linguaggi di programmazione. Una comprensione degli aspetti implementativi porta ad una maggiore comprensione del perché il linguaggio è stato progettato in un certo modo. Questo permette quindi di usare il linguaggio in modo più intelligente ed efficiente, ovvero nel modo in cui era stato progettato per essere usato;

- Per usare meglio i linguaggi conosciuti. Infatti, conoscendo aspetti di progettazione ed implementativi dei linguaggi, il programmatore ha a disposizione tutti gli strumenti che gli permettono da un lato di *conoscere meglio parti prima sconosciute e inutilizzate del linguaggio*, dall'altro di *evitare errori dovuti ad un uso inconsapevole dei costrutti del linguaggio.*

Storia

COME NASCONO I LINGUAGGI?

- **A quali necessità rispondono**
- **Perché i linguaggi di programmazione si sono evoluti proprio nei linguaggi che conosciamo oggi (ad alto livello)?**
- **Perché contengono determinati costrutti?**

A volte per capire ciò che dobbiamo studiare è necessario fare un passo indietro e capire da dove viene ciò che dobbiamo studiare. Comprendere come e da dove è nata la necessità di progettare quelli che oggi conosciamo come linguaggi di programmazione, ed in particolare i così detti linguaggi ad alto livello, permette sicuramente di capire meglio dove si sta andando, quali possono essere le prossime sfide nell'evoluzione dei linguaggi e quali sono invece gli elementi essenziali di un linguaggio di programmazione

Quindi, per arrivare a comprendere l'importanza dello studio dei linguaggi di programmazione da un punto di vista generale, è importante capire quali necessità hanno portato alla nascita e all'evoluzione dei linguaggi di programmazione.

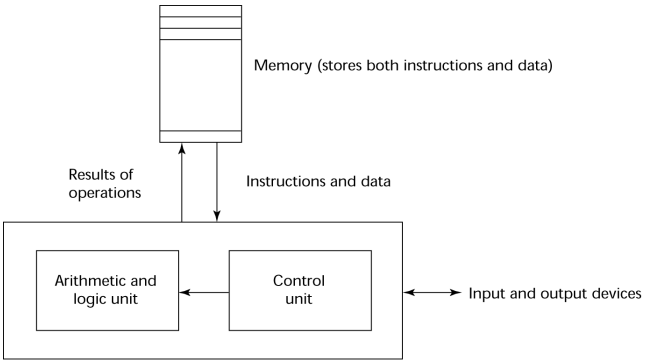
Il principale problema dell'informatica consiste nel voler far eseguire ad una macchina ***algoritmi*** che manipolano ***dati*** :

- La macchina di Babbage, che si può vedere come il primo "computer" nasce per eseguire calcoli matematici complessi in modo automatico;
- Enigma nasce per decifrare un codice mediante un algoritmo di forza bruta che tenta in sequenza tutte le combinazioni).

Ma cosa è una macchina su cui possiamo eseguire programmi che manipolano dati? In generale è una macchina ***programmabile***, ovvero un calcolatore che può eseguire insiemi di istruzioni (passi di calcolo dell'algoritmo) che chiamiamo programmi, ricevuti come input (***macchine universali***).

In particolare i computer moderni hanno una architettura che nasce dall'architettura di Von Neumann. Questa è una tipologia di architettura hardware (HW) per macchine programmabili, con programma memorizzato, dove dati ed istruzioni condividono la stessa area di memoria. Storicamente la prima implementazione risale agli anni '40 (ENIAC - prima macchina general purpose e Turing completa). Quindi, la macchina di Von Neumann è una architettura hardware (HW) con elementi fisici (memoria e unità aritmetiche).

Quindi per operare su questa architettura di che linguaggi abbiamo bisogno? Essenzialmente dobbiamo istruire una CPU (eseguire ***algoritmi***) per operare sui ***dati*** in memoria. È proprio la struttura essenziale di questa architettura (CPU+memoria) ha messo al centro del concetto di linguaggio di programmazione (o almeno dei primi linguaggi poi evoluti in quelli oggi noti come *linguaggi imperativi*) la cella di memoria. Comunemente ad esempio, anche a livello formale,



lo stato di esecuzione di una macchina è descritto attraverso i valori contenuti nelle sue celle di memoria.

Il primo linguaggio che permette di programmare tale architettura è quello che si basa sull'implementazione stessa dell'architettura, ovvero solo funzionamento degli elementi HW che la costituiscono che , comportandosi come interruttori (semplificando molto) possono trovarsi solo in due stati: acceso o spento. La programmazione con schede perforate degli anni '50 si basa esattamente su questo linguaggio a due valori: linguaggio binario.

Di fatto la macchina HW è in grado di interpretare solo il linguaggio ***binario***, ovvero un linguaggio costituito esclusivamente da stringhe di bit.

DAL LINGUAGGIO BINARIO AI LINGUAGGI AD ALTO LIVELLO

Abbiamo già osservato che nell'architettura HW dati e programmi condividono la stessa area di memoria, questo significa anche che a basso livello (HW) sono rappresentati nello stesso modo, con lo stesso linguaggio binario. In altre parole ***dati e istruzioni hanno lo stesso formato***: Una stringa di bit può essere sia un dato che un una o più di istruzioni.

Quindi il linguaggio binario è un linguaggio di programmazione perché permette di programmare una macchina, ma chi lo interpreta deve essere in grado di distinguere stringhe che rappresentano dati da stringhe che rappresentano istruzioni che manipolano dati.

La possibilità di vedere i programmi come dati passati in input ad una macchina (universale) è stato un passaggio fondamentale nella storia dell'informatica per arrivare al concetto di macchina programmabile, ma allo stesso tempo ha comportato la difficoltà intrinseca di riuscire a distinguere dati e istruzioni, in un linguaggio fatto di soli due simboli. Difficoltà che ad esempio rende molto difficile il processo di disassemblaggio (traduzione da binario ad assembly) e decompilazione (da assembly ad alto livello). Infatti, dati e istruzioni, pur avendo la stessa rappresentazione a livello macchina, vanno trattati in modo sostanzialmente diverso: il dato viene manipolato, l'istruzione viene interpretata.

A questo punto dovrebbe essere evidente che usare il linguaggio binario per programmare algoritmi in una macchina non è auspicabile, perché è umanamente difficile poter capire (distinguendo i dati dalle istruzioni) e manipolare le milioni di stringhe binarie che servirebbero per descrivere l'algoritmo.

Per queste ragioni abbiamo bisogno di linguaggi diversi da quello binario, linguaggi che un essere umano possa capire e manipolare (quindi che usa parole e non simboli) ma che allo stesso tempo siano interpretabili da una macchina (ovvero non ambigui come il linguaggio naturale). Questi saranno i linguaggi ad alto livello che in modo poi automatico (mediante compilazione o interpretazione) vengono eseguiti a livello macchina.

COSA SIGNIFICA PROGRAMMARE?

Ritorniamo quindi a cosa significa programmare e quindi cosa è un linguaggio di programmazione. Per quanto detto, un linguaggio di programmazione deve permettermi di ***agire*** su una macchina per manipolare ***dati***, quindi un linguaggio di programmazione deve essere uno ***strumento che permette di descrivere algoritmi e rappresentare dati***.

Abbiamo detto che il linguaggio binario è un linguaggio di programmazione in quanto permette di rappresentare algoritmi (mediante istruzioni) e dati, ma come abbiamo visto non è umanamente utilizzabile o interpretabile, in quanto in principio, ***una istruzione potrebbe essere usata come dato***, e un ***dato interpretato come istruzione***. Questo significa che un linguaggio di programmazione deve pr prima cosa permettere di superare questa difficoltà, fornendo una chiara distinzione tra dato e istruzione.

Algoritmi

Torniamo quindi a ciò che dobbiamo *descrivere*, ovvero gli algoritmi. Un algoritmo è una sequenza di passi il cui significato è il calcolo di una funzione. In altre parole, un algoritmo scompone un calcolo complesso in passi elementari di computazione. Un algoritmo è quindi un concetto astratto che deve trovare forma concreta per essere eseguito, e la sua forma concreta è la definizione di un programma, ovvero una sequenza (finita) di istruzioni.

Ci servono quindi strumenti per scrivere precisamente programmi che descrivono/implementano algoritmi. I ***linguaggi di programmazione*** nascono quindi proprio per questo: Sono ***linguaggi formali le cui frasi sono programmi, ovvero forme concrete di algoritmi eseguibili/interpretabili da una macchina.***

Un algoritmo è una sequenza (finita) di passi primitivi di calcolo descritti mediante una frase ben formata (programma) in un linguaggio di programmazione

Quindi per specificare ed eseguire (ovvero dare forma concreta) ad un algoritmo abbiamo bisogno di scrivere programmi e per scrivere programmi abbiamo bisogno di strumenti che ci permettono di farlo, ovvero di linguaggi di programmazione.

Va osservato che non esiste una corrispondenza precisa tra programmi e algoritmi:

- ***Un programma non è necessariamente un algoritmo,*** in quanto una frase grammaticalmente corretta può non avere significato (torneremo sulla distinzione tra sintassi e semantica), questo avviene anche nel linguaggio naturale in cui possiamo scrivere una frase drammaticamente corretta che però non ha significato.
- ***Lo stesso algoritmo può avere concretizzazioni diverse***, ovvero lo stesso algoritmo può essere implementato da tanti (infiniti) programmi.

A questo punto si aprono diverse questioni a cui daremo risposta durante il corso (se non in altri corsi):

- Dobbiamo stabilire le regole che permettono di costruire frasi grammaticalmente corrette in un dato linguaggio: ***Grammatiche Context-free***
- Dobbiamo stabilire gli elementi base (elementi terminali della grammatica) di cui abbiamo bisogno per definire i programmi in un linguaggio di programmazione: ***Categorie sintattiche***
- Il significato di quale categoria sintattica descrive l'insieme dei passi elementari/primitivi di computazione: ***Comandi***

I programmi quindi concretizzano algoritmi che manipolano ***dati***. Ma cosa sono invece i dati? Sono informazioni memorizzate concretamente in celle di memoria e astratte in elementi che il linguaggio di programmazione può manipolare: le ***variabili***.

I LINGUAGGI DI PROGRAMMAZIONE: COSA SONO?

Il **programma**, abbiamo detto, è la concretizzazione di un **algoritmo** che manipola dati, manca però un altro ingrediente, ovvero la ***trasformazione*** di dati che l'algoritmo vuole eseguire, questa trasformazione è il significato dell'algoritmo, la *semantica* del programma. Quindi, dare semantica ad un programma significa descrivere matematicamente l'effetto dell'esecuzione del programma sui dati.

Il programma (o meglio il linguaggio di programmazione nel quale il programma è scritto) costituisce ciò che chiameremo ***sintassi***, mentre l'effetto dell'esecuzione del programma, la trasformazione dei dati eseguita, costituisce ciò che chiameremo ***semantica***.

Ma quale è la profonda differenza tra programma e trasformazione eseguita, tra sintassi e semantica? Perché abbiamo bisogno di fare questa distinzione, e di creare strumenti, anche profondamente diversi per descrivere i due concetti?

Le trasformazioni di dati sono funzioni sui dati e, in quanto tali, sono notoriamente descrivibili mediante il linguaggio matematico, il quale è sicuramente rigoroso e formale. Questo significa che il linguaggio della matematica può essere considerato un linguaggio di programmazione (di fatto esiste un paradigma detto funzionale che usa le funzioni come strumento di computazione), ma ha tre importanti mancanze collegate tra loro: (1) non sempre permette rappresentazioni finite di oggetti infiniti, (2) non sempre fornisce direttamente un metodo di calcolo di ciò che si vuole rappresentare, (3) non permette di rappresentare tutti i problemi "calcolabili".

Consideriamo il punto (1), questo è importante perché il programma è necessariamente un oggetto finito (abbiamo detto che è sempre concreto, memorizzabile in una macchina), mentre il calcolo di una funzione richiede l'esecuzione di un insieme di passi primitivi, che però possono essere infiniti. Alla base della teoria della calcolabilità troviamo proprio la necessità di ammettere computazioni divergenti per poter catturare tutto ciò che effettivamente un programma può calcolare (si veda la dispensa di Fondamenti).

Questo significa che dobbiamo pensare al ***programma come ad una rappresentazione finita di un insieme, potenzialmente infinito, di passi primitivi di computazione.***

I linguaggi di programmazione devono ***dare specifica finita di oggetti potenzialmente infiniti***

La componente finita è la sintassi mentre l'effetto potenzialmente infinito è la semantica. Purtroppo però, senza la logica proposizionale, la matematica non rappresenta lo strumento migliore perché può fallire nel rappresentare entità infinite. Un tipico esempio di tale fallimento si ha nella rappresentazione di insiemi, la matematica (senza uso della logica) permette la descrizione in forma *esplicita*, ovvero enumerando gli oggetti dell'insieme, che se infinito non può essere enumerato per intero {0,2,4,6,…}. Questa non solo non è una rappresentazione rigorosa, ma non fornisce nemmeno un metodo di calcolo degli oggetti dell'insieme. Chiaramente fornisce quindi una rappresentazione ambigua dell'insieme, perché non sappiamo con certezza (da questa rappresentazione) quale sarà il prossimo elemento, probabilmente 8 ma potrebbe essere qualunque elemento. Per avvicinarci a quello di cui abbiamo bisogno dobbiamo usare la logica proposizionale, ovvero dobbiamo descrivere gli insiemi in forma *implicita*, mediante un predicato che deve essere soddisfatto da tutti e soli gli elementi dell'insieme. In questo modo abbiamo una specifica finita e non ambigua anche per insiemi infiniti: ad esempio {2x | x naturale}. In questo caso x è un identificatore locale (un nome finito che può avere infinite istanze), x naturale è la dichiarazione di una proprietà di x, e 2x è il calcolo per ottenere elementi dell'insieme.

Da queste considerazioni possiamo concludere e osservare che i linguaggi di programmazione ereditano tanti aspetti dalla logica proposizionale e del primo ordine.

Purtroppo, anche la logica preposizionale non basta a descrivere un algoritmo. Consideriamo infatti il punto (2). Quando si parla di algoritmi per calcolare funzioni/trasformazioni, è chiaro che intendiamo definire i *passi* di computazione primitiva necessari per calcolare la trasformazione. Ora vediamo che la logica in questo ci fornisce qualche strumento utile.

Infatti il linguaggio logico in generale è costituito da regole di inferenza e assiomi. Quando forniamo una frase in un linguaggio logico, implicitamente intendiamo dire che esiste un processo che permette di ***dimostrare*** se la frase è un teorema vero (dimostrabile per inferenza dalle regole) o falso (confutabile). Quindi la validità di un teorema nella logica è definibile mediante un algoritmo che utilizza come passo primitivo di computazione l'applicazione di regole di inferenza, tipo il modus ponens. Ovvero programmare, in tal caso, vuol dire fornire una logica nel quale il calcolo da eseguire consiste della dimostrazione/confutazione di un teorema nella logica data.

I linguaggi di programmazione devono ***descrivere i passi di un calcolo***

Possiamo osservare che ogni regola è una rappresentazione finita di un numero potenzialmente infinito di istanze della regola. Ad esempio, se sappiamo che A è un insieme di numeri pari e sappiamo che x sta in A, allora il modus ponens ci permette di dimostrare che x è pari. Quindi sappiamo che "x pari" è corretto e sappiamo come dimostrarlo. Al contrario, quando non riusciamo a trovare le regole che ci permettono di dimostrare un fatto/teorema, allora quel fatto è falso, ovvero non è computabile nella logica. Il forte legame tra il linguaggio logico e i linguaggi di programmazione nasce proprio dal fatto che la logica è stata usata per caratterizzare cosa nella matematica può essere dimostrato in modo automatico (mediante regole e assiomi), ovvero dimostrato da una macchina i cui passi primitivi di computazione sono pattern matching e sostituzione (dalla logica nasce il paradigma logico, i cui linguaggi di programmazione usano esattamente questi strumenti di calcolo per implementare algoritmi).

Ma perché allora non possiamo usare la logica come linguaggio di programmazione?

Arriviamo quindi al punto (3), infatti è stata dimostrata l'incompletezza del linguaggio logico formale (Teorema di incompletezza di Godel), perché esistono sempre oggetti infiniti che non hanno rappresentazione finita nella logica matematica e che quindi non posso essere il risultato di una dimostrazione logica (problemi semi-decidibili e non decidibili).

Il problema è che la logica non è in grado di rappresentare in modo finito dimostrazioni infinite, ovvero dimostrazioni/computazioni che richiedono un numero infinito di passi di esecuzione. La logica, infatti, non è in grado di catturare il concetto di funzione Turing-calcolabile (si veda la dispensa di Fondamenti)

I linguaggi di programmazione devono ***descrivere calcoli potenzialmente infiniti***

Ecco che quindi abbiamo bisogno di un modello di calcolo basato su linguaggi formali/artificiali che permetta di r*appresentare in modo finito un numero potenzialmente infinito di passi di computazione*, e questi sono tutti i modelli di calcolo Turing completi, tra i quali ci sono i linguaggi di programmazione che conosciamo.

Quindi sappiamo che esistono calcoli non rappresentabili neanche dalla logica, e comunque anche la logica, seppur permettendo la rappresentazione finita di oggetti infiniti, e seppur permettendo la descrizione di passi di calcolo, non ha strumenti adeguati per specificare rigorosamente un qualunque processo di computazione.

Linguaggio Matematico: Notazione rigorosa per rappresentare funzioni ma non sempre oggetti infiniti e computazioni, ovvero passi di calcolo.

Linguaggio Logico: Regole e assiomi rendono possibile specificare il processo di computazione (ancora in modo implicito), e permette di rappresentare formalmente oggetti infiniti in modo finito, ma non sempre computazioni infinite.

NASCITA DEI LINGUAGGI DI PROGRAMMAZIONE

Linguaggio di programmazione: Permette di specificare in modo accurato esattamente le primitive del processo di computazione, con la rigorosità e la potenza della logica.

A partire dagli anni '30, ovvero da quando Hilbert pose il problema dell'espressività dei linguaggi formali, i matematici hanno affrontato il problema di formalizzare il concetto di algoritmo, ovvero di descrizione di un processo di computazione potenzialmente infinito, cercando di caratterizzare cosa è possibile specificare in modo algoritmico (finito) e cosa no. Questo ha portato alla costruzione di vari modelli di computazione (MdT, lambda-calcolo, funzioni ricorsive,…), e tutti questi formalismi, sviluppati con strumenti diversi si sono dimostrati essere equivalenti tra loro, nonché equivalenti a ciò che oggi chiamiamo linguaggi di programmazione (tesi di Church, Fondamenti).

Questo ci dice che, almeno per ciò che ad oggi consideriamo essere un processo di calcolo, i linguaggi di programmazione (con tutti i loro limiti) sono lo strumento più potente che possiamo avere a disposizione per implementare algoritmi, ovvero per far eseguire processi di calcolo ad una macchina programmabile.

Se da un lato non esiste una definizione formale di cosa rende un linguaggio formale un linguaggio di programmazione, dall'altro lato chiunque abbia programmato è in grado di capire se un linguaggio è o meno un linguaggio di programmazione. Tale vuoto può solo essere colmato comprendendo a fondo quali elementi costituiscono in generale un linguaggio di programmazione (quelle che chiameremo ***categorie sintattiche***) e quale ruolo ha nel linguaggio ognuno di questi elementi.

Quindi, per prima cosa dobbiamo comprendere cosa ha determinato la struttura dei linguaggi di programmazione moderni, e quindi quali strumenti mi deve offrire un linguaggio per essere un linguaggio di programmazione.

PROGETTARE LINGUAGGI

- **Cosa influisce sulla progettazione dei linguaggi?**
- **Quali caratteristiche deve garantire un linguaggio di programmazione?**

- L'architettura base dei computer ha avuto un effetto determinante nella progettazione dei linguaggi moderni di programmazione. La maggior parte dei linguaggi sviluppati negli ultimi 50 anni sono stati sviluppati attorno all'architettura della macchina di von Neumann: Nascono così i linguaggi imperativi.
- Il contesto in cui il linguaggio deve operare determina le funzionalità che il linguaggio deve gestire efficacemente:
  - Applicazioni scientifiche (Grande numero di computazioni floating-point; uso di array: **Fortran**)
  - Applicazioni economiche (Produrre report, usare numeri decimali e caratteri: **COBOL**)
  - Intelligenza artificiale (Manipolazioni di simboli invece che di caratteri; uso di liste linkate: **LISP**)
  - Programmazione di sistemi (Necessita efficienza per l'uso continuo: **C**)
  - Web Software (Collezione eclettica di linguaggi: markup (**HTML**), scripting (**PHP**), general-purpose PHP), general-purpose (**Java**))
- Nuove metodologie e nuove esigenze hanno portato all'estensione di linguaggi esistenti ma anche allo sviluppo di nuovi paradigmi e nuovi linguaggi di programmazione. Si pensi ad esempio all'evoluzione della programmazione orientata agli oggetti.

ASPETTI DI PROGETTAZIONE: LEGGIBILITÀ

Leggibilità (Readability): sintassi chiara, nessuna ambiguità, facilità di lettura e comprensione dei programmi

È importante poter leggere il codice come un libro, sappiamo che il codice contiene sempre errori, che possono essere scoperti anche successivamente e da altri rispetto a chi ha scritto il codice, quindi altri devono poter leggere e comprendere il codice. Inoltre, altri potrebbero dover estendere il SW per riutilizzare, per aggiungere nuove caratteristiche o semplicemente per manutenerlo. In tutti questi casi la **leggibilità** è fondamentale.

Contribuiscono alla leggibilità:

1. La generale *semplicità* di un linguaggio (pochi ed essenziali costrutti di base). Complicano i linguaggi la possibilità di poter fare la stessa cosa in più modi diversi (x=x+1,x++,…) e l'overloading degli operatori, dove un singolo simbolo di operatore ha più significati.
2. Il significato di un elemento di un linguaggio è *ortogonale* se è indipendente dal contesto di utilizzo all'interno del programma. Ortogonalità e semplicità sono concetti legati: Più è ortogonale la progettazione di un linguaggio, meno eccezioni alle regole ci sono, meno eccezioni significa un più alto grado di regolarità nella progettazione che rende più facile imparare, leggere e capire il linguaggio.
3. La presenza di strumenti per la definizione di tipi di dati e strutture dati è un altro aiuto alla leggibilità (ad es. l'uso di interi come booleani diminuisce la leggibilità)
4. La struttura della sintassi ha effetto sulla leggibilità (composizione flessibile, parole chiave significative, ecc.)

ASPETTI DI PROGETTAZIONE: SCRIVIBILITÀ

Scrivibilità (Writability): facilità di utilizzo di un linguaggio per creare programmi, facilità di analisi e verifica dei programmi

È una misura della facilità di utilizzo di un linguaggio per creare programmi per una specifica classe di problemi. Ciò che solitamente ha effetto sulla leggibilità ha effetto anche sulla scrivibilità.

1. Semplicità e ortogonalità
   - la presenza di tanti costrutti fa si che il programmatore possa non conoscerli tutti e quindi non usarli adeguatamente, quindi pochi costrutti, un piccolo numero di primitive, un piccolo insieme di regole per combinarle;
2. Supporto per l'astrazione
   - Astrazione è un concetto chiave nei moderni linguaggi. Consiste nell'abilità di definire e usare strutture o operazioni complesse in modi che permettono di ignorare i dettagli. Fondamentalmente esistono due tipi di astrazione: processi (uso di sottoprogrammi) e dati.
3. Espressività
   - Può riferirsi a diverse caratteristiche, ad esempio mettere a disposizione un insieme di modi relativamente convenienti per specificare operazioni, oppure applicabilità e numero di operazioni e di funzioni predefinite

ASPETTI DI PROGETTAZIONE: AFFIDABILITÀ E COSTO

Affidabilità (Reliability): conformità alle sue specifiche

Costo: costo complessivo di utilizzo

Un programma è affidabile se soddisfa le specifiche in tutte le condizioni.

1. Il type checking (controllo degli errori di tipo) è importante per l'affidabilità. Spesso eseguito a tempo di compilazione, essendo un procedimento costoso. Inoltre prima gli errori vengono rilevati e meno costoso è ripararli.
2. Gestione delle eccezioni (Gestione di errori run-time per permettere la continuazione dell'esecuzione e l'attuazione di eventuali misure correttive)
3. La presenza di potenziali aliasing (presenza di due o più metodi di riferimento per la stessa locazione di memoria) è sicuramente un potenziale problema.

Sono esempi di costo quelli di…

1. … di addestramento di programmatori per usare il linguaggio
2. …di scrittura di programmi (vicinanza a particolare applicazioni)
3. …di compilazione dei programmi
4. …di esecuzione dei programmi
5. Sistema di implementazione del linguaggio: disponibilità di compilatori liberi
6. Affidabilità: poca affidabilità alza i costi
7. …di mantenimento dei programmi

CLASSIFICAZIONE DEI LINGUAGGI

- PER METODO DI COMPUTAZIONE

  - **Basso livello**: I linguaggi a basso livello hanno caratteristiche specifiche dipendenti dall'architettura che si sta programmando.

    - Il ***linguaggio binario*** abbiamo già detto non ha neanche la distinzione tra dati e programmi;

    - l'***Assembly*** è un linguaggio strutturato (jump e assegnamenti). Si studiano solitamente in contesti collegati allo studio delle architetture e non sono di nostro interesse.

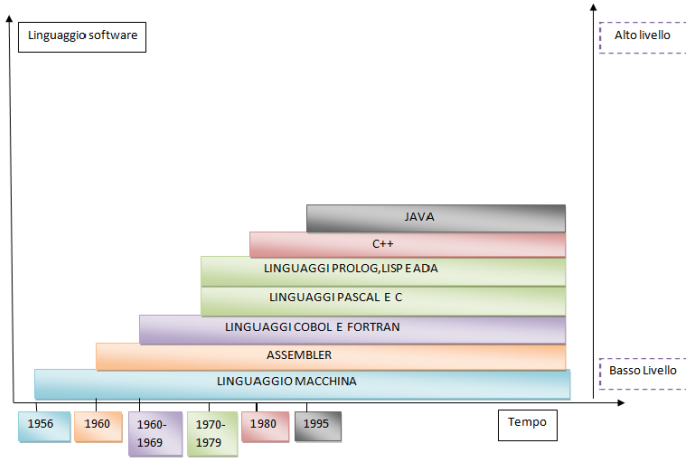
  - **Alto livello**: I linguaggi ad alto livello permettono una programmazione strutturata in cui dati ed istruzioni hanno rappresentazioni diverse.

    - I ***linguaggi imperativi*** descrivono come concetto chiave l'elemento fondamentale dell'architettura di von Neumann, ovvero la cella di memoria. In particolare il concetto di variabile è l'astrazione logica della cella, mentre l'assegnamento è l'operazione primitiva di modifica della cella di memoria e quindi dello stato della macchina. Nei linguaggi imperativi si eseguono assegnamenti, che vengono controllati in modo sequenziale, condizionale e ripetuti.

    - I ***linguaggi funzionali*** sono i più vicini alla matematica, ovvero descrivono i passi di calcolo come funzioni matematiche e quindi compongono e applicano funzioni. Qui una variabile è come nella matematica, semplicemente un nome per qualcosa. Non può cambiare nel tempo, durante la computazione.

    - I ***linguaggi logici*** usano la logica, ovvero pattern matching e unificazione/sostituzione come passo di calcolo primitivo.

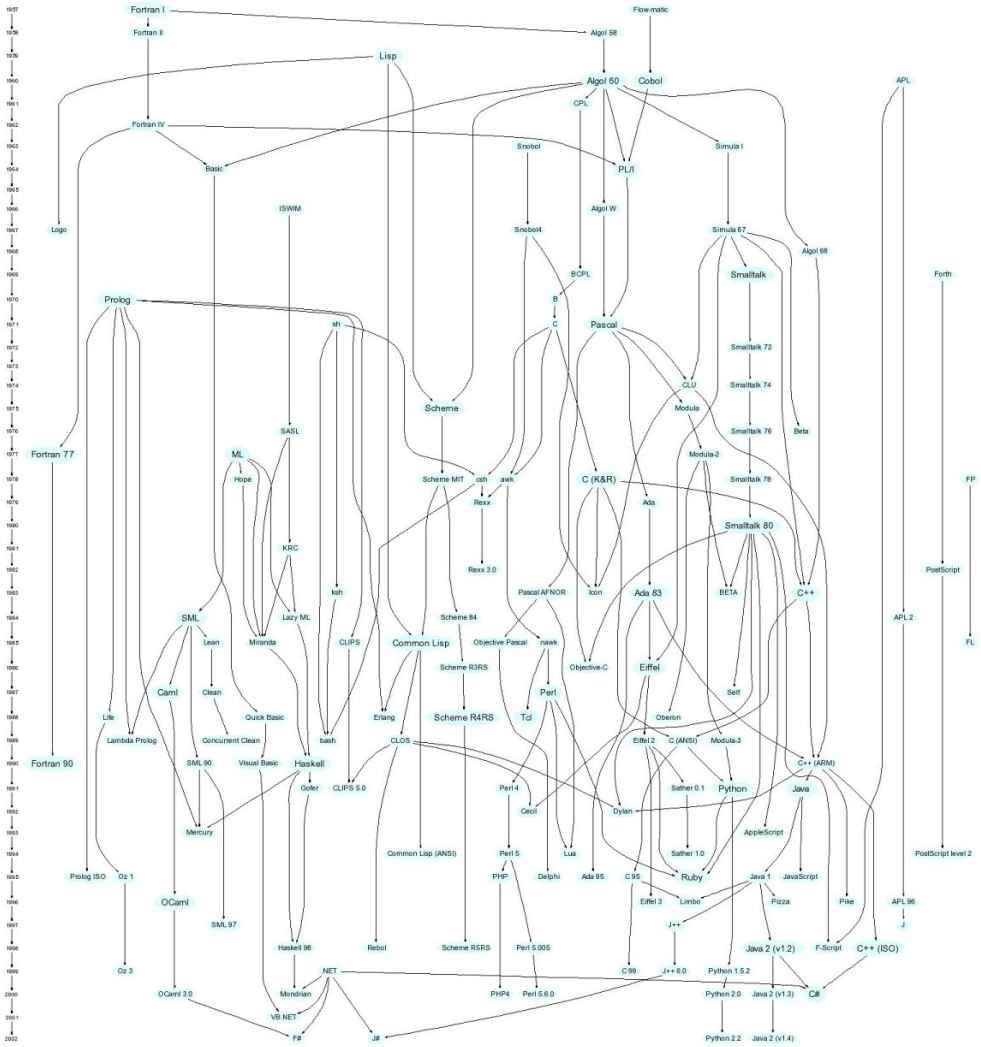
CLASSIFICAZIONE DEI LINGUAGGI



- PER CARATTERISTICHE: Per buona parte degli anni '50 e '60 lo sviluppo dei linguaggi di programmazione si è focalizzato sullo studio delle caratteristiche di base: strutture di base di controllo e per i dati, efficienza nell'esecuzione. Una volta fissate queste caratteristiche di base, negli anni '70 e '80 la maggior parte della concentrazione si è focalizzata su caratteristiche aggiuntive (Modula estende Pascal con i moduli, ADA estende Modula aggiungendo ad esempio concorrenza, gestione eccezioni, ecc). Estensioni significa che le strutture base rimangono le stesse, ma ad esse si aggiungono caratteristiche nuove che permettono di migliorare la soluzione di specifici problemi. Small talk 80 arriva ad aggiungere il concetto di classe/oggetto. Le caratteristiche studiate per estendere le strutture base sono proprio quelle viste (concorrenza, parallelismo, distribuzione, OO…). Questo vale per gli imperativi, ma anche per funzionali e per altri paradigmi, anche se in modi specifici.

Esempi: Sequenziali; Concorrenti; Modulari; Paralleli/ Distribuiti; Orientati agli oggetti/classi;Interpretati/ Scripting.

EVOLUZIONE



IMPLEMENTARE LINGUAGGI

- **Cosa serve per eseguire un programma in un linguaggio di programmazione?**
- **Cosa significa definire un linguaggio di programmazione?**

- Cosa significa implementare un linguaggio di programmazione? Ovvero, cosa dobbiamo fare per poter usare un linguaggio formale per programmare una macchina?
- Dobbiamo per prima cosa **implementarlo**.
- Da cosa dipende l'implementazione? Sicuramente, il modo cui deve essere implementato è direttamente collegato all'architettura che deve programmare, ma un aspetto centrale consiste anche nel progettare i suoi elementi, e questo dipende da ciò che vogliamo rappresentare in modo più o meno efficiente, pur mantenendo lo stesso potere espressivo (Turing calcolabilità).
- Ricordiamo che

Un ***Linguaggio di Programmazione*** L è un insieme di costrutti e regole per descrivere *algoritmi* e *dati.*

Un ***Programma*** è un insieme finito di istruzioni/ costrutti del del linguaggio di programmazione.

IMPLEMENTARE LINGUAGGI

Implementare un linguaggio, significa renderlo comprensibile alla macchina da programmare e che quindi deve eseguirne i programmi. Abbiamo detto che fondamentalmente le macchine su cui sono stati progettati i primi linguaggi, e che ancora oggi sono le più diffuse, si basano sull'architettura di von Neumann, quindi per capire l'implementazione dei linguaggi di programmazione dobbiamo partire dal funzionamento di tali macchine.

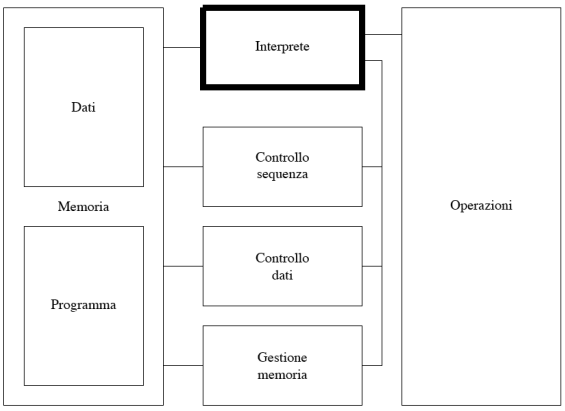
Il funzionamento si basa sulla ripetizione di un ciclo che costituisce ***l'interprete*** del linguaggio che la macchina riconosce:

- Viene letta l'istruzione dalla memoria
- Vengono letti gli operandi
- Viene memorizzato il risultato

Quando si opera con un linguaggio di programmazione su una macchina, in realtà usiamo esattamente le primitive che quella macchina ci mette a disposizione, ovvero le primitive che la macchina è in grado di interpretare in termini di operazioni/azioni che si possono eseguire direttamente nella macchina. Quindi, come già detto, se potessimo lavorare solo con la macchina HW, allora potremmo scrivere programmi solo in linguaggio binario.

Invece, quando usiamo un linguaggio di programmazione ad alto livello lavoriamo su una macchina che interpreta le istruzione del linguaggio di programmazione ed ignoriamo completamente l'esistenza, ad un certo punto, della macchina fisica e del suo linguaggio binario.

MACCHINA ASTRATTA



Implementare un linguaggio significa quindi realizzare la macchina (che chiameremo astratta, non essendo fisica) che interpreta il linguaggio.

Dato un linguaggio L di programmazione, la ***macchina astratta*** M_L per L è un insieme di strutture dati ed algoritmi che permettono di memorizzare ed eseguire i programmai scritti in L.

La collezione di strutture dati ed algoritmi in particolare serve per: acquisire la prossima istruzione; gestire le chiamate ed i ritorni dai sottoprogrammi; acquisire gli operandi e memorizzare i risultati delle operazioni; mantenere le associazioni fra nomi e valori denotati; gestire dinamicamente la memoria. Quindi la macchina astratta è la combinazione di una memoria che immagazzina i programmi e di un interprete che esegue le istruzioni dei programmi.

Il linguaggio L, riconosciuto (interpretato) dalla macchina astratta M_L viene anche chiamato linguaggio macchina. Formalmente è l'insieme di tutte le stringhe interpretabili da M. Ai componenti di M corrispondono opportuni componenti di L_M: Tipi di dati primitivi e costrutti di controllo (per controllare l'ordine di esecuzione e acquisizione e trasferimento dati).

I programmi scritti nel linguaggio macchina di M saranno memorizzati nelle strutture dati della memoria della macchina per distinguerli da altri dati primitivi su cui opera la macchina. Un esempio di macchina astratta è la macchina HW.

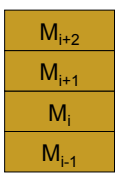
DOVE REALIZZARE LA MACCHINA ASTRATTA

Una qualsiasi macchina astratta per L, per essere eseguita, deve prima o poi utilizzare qualche dispositivo fisico che concretamente esegue le istruzioni di L. Questo significa che in qualunque sistema informatico esiste sempre una macchina HW (realizzata sul sistema fisico) ma non significa che tutte le macchine sono realizzate a livello HW, anzi, per poter controllare la complessità di un sistema informatico, la realizzazione della macchina astratta può avvenire a vari livelli, generalmente sovrapposti, a partire da quello fisico. Quindi le macchine astratte possono essere realizzate a diversi livelli:

1. HW: Sempre possibile e concettualmente semplice. Si realizza mediante dispositivi fisici, e il linguaggio macchina è il linguaggio fisico/binario. L'esecuzione veloce ma i linguaggi ad alto livello sono molto lontani dalle funzionalità elementari a basso livello. Inoltre è difficile da modificare. Viene usata per sistemi dedicati.
2. FW: Consiste nella simulazione delle strutture dati e degli algoritmi in M mediante microprogrammi. Il linguaggio macchina microprogrammato è a basso livello e consiste di microistruzioni che specificano semplici operazioni di trasferimento dati tra registri, da e per la memoria principale ed eventualmente attraverso circuiti logici che realizzano operazioni aritmetiche. I microprogrammi risiedono in speciali aree di memoria di sola lettura. Veloce e più flessibile della realizzazione HW.
3. SW: Realizzazione delle strutture dati e degli algoritmi di M mediante programmi scritti in un altro linguaggio L' che possiamo supporre già implementato. Ovvero, avendo la macchina astratta per L', possiamo realizzare quella per L mediante opportuni programmi scritti in L' che interpretano i costrutti di L simulando le funzionalità di M per L. Riduce la velocità ma aumenta la flessibilità.

LIVELLI DI ASTRAZIONE

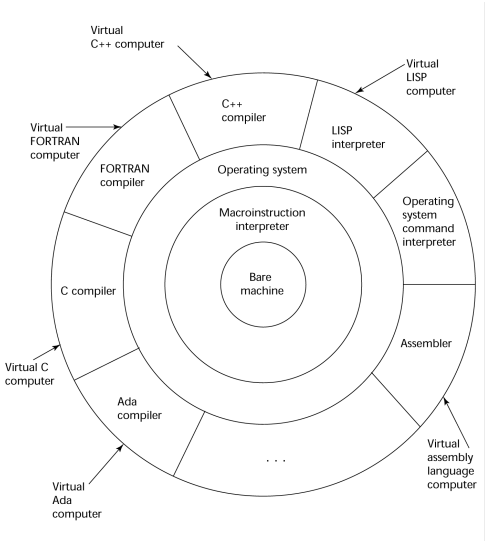
Per permettere l'uso di linguaggi di programmazione ad alto livello, controllando la complessità della macchina fisica e del suo vero linguaggio, quello binario, ai sistemi SW è stata data una struttura suddivisa a livelli di astrazione cooperanti ma sequenziali e indipendenti: Ciascun livello è definito da un linguaggio L che è l'insieme delle istruzioni che il livello mette a disposizione per i livelli successivi/superiori. Tutte le istruzioni di un linguaggio ad un livello i sono implementate da programmi a livello j<i.



M_i: • usa i servizi forniti da M_{i-1} (il linguaggio L_{Mi-i})
• per fornire servizi a M_{i+1}(interpretare L_{Mi+i})
• nasconde (entro certi limiti) la macchina M_{i-1}

Stando al livello i può non essere noto
(e in genere non serve sapere...)
quale sia il livello 0 (hw)

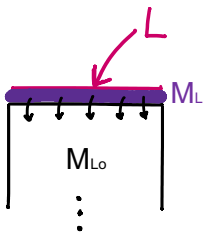
L4	APPL	Linguaggi di programmazione, Basi di dati, interfacce grafiche
L3	SO	Processori, processi, I/O, Memoria: L3.0 = Kernel, L3.1 = Tutte le risorse, L3.2 = Interfaccia (superuser)
L2	ASM	Codici mnemonici per gestire REGS, ALU, MEM (Assembly)
L1	FW	REGS, ALU, MEM (Microprogrammi)
L0	HW	Componenti fisici



COME REALIZZARE LA MACCHINA ASTRATTA

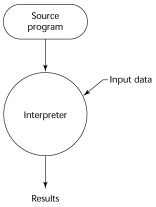
Per implementare un linguaggio L dobbiamo realizzare una macchina astratta M_L, in grado di eseguirlo. M_L è quindi un dispositivo che permette di eseguire programmi scritti in L ma la corrispondenza non è biunivoca: Una macchina astratta corrisponde univocamente ad un linguaggio, il suo linguaggio macchina, ma dato un linguaggio L esistono infinite macchine astratte che hanno come linguaggio macchina esattamente L. Tali macchine differiscono nel modo in cui la macchina astratta viene realizzata e nelle strutture dati utilizzate. Abbiamo già visto, ad esempio, che una macchina astratta (volendo anche per lo stesso linguaggio) si può realizzare a livello HW, FW o SW. La realizzazione della macchina astratta dipenderà quindi dalle funzionalità che le sono messe a disposizione dai livelli sottostanti (tranne che per il livello HW, non considerato in questo corso).

Quindi abbiamo un linguaggio L da implementare e una macchina astratta M_{Lo} a disposizione, che è la macchina ospite (con linguaggio macchina L_O). M_{Lo} è il livello su cui vogliamo implementare L e che mette a disposizione di M_L le sue funzionalità .



Realizzare M_L consiste quindi in realizzare una macchina che "traduce" L in L_O, ovvero che interpreta tutte le istruzioni di L come (sequenza di) istruzioni di L_O. Possiamo distinguere due modalità radicalmente diverse a seconda del fatto che si abbia una traduzione "implicita" realizzata dalla simulazione dei costrutti di M_L (ovvero di L) mediante programmi scritti in Lo (***soluzione interpretativa***), oppure si abbia un una traduzione "esplicita" dei programmi di L in corrispondenti programmi di Lo (***soluzione compilativa***).

SOLUZIONE INTERPRETATIVA: INTERPRETE



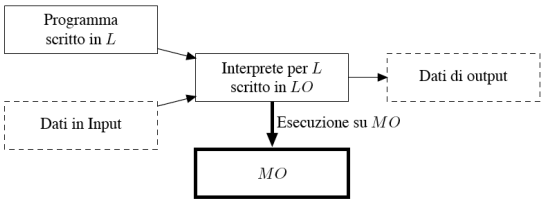
Notazione:
$Prog^L$ = insieme di programmi scritti in L
D = Insieme di dati (input e output)
$P^L \in Prog^L$, in, out$\in$D
$[\![P^L]\!]$:D$\rightarrow$D semantica di P^L tale che $[\![P^L]\!](in)=out$ se P^L eseguito a partire da in restituisce out.

Un interprete è un programma $int^{Lo,L}$ che esegue, sulla macchina astratta per L_O, programmi P^L, scritti in L, su un fissato input in$\in$D. In altre parole, un interprete è una ***macchina universale*** (Fondamenti) che preso un programma e un suo input, lo esegue su quell'input, usando solo funzionalità messe a disposizione dal livello (macchina astratta) sottostante.

Interprete da L a L_O:
Dato $P^L \in Prog^L$ e in$\in$D, un interprete $int^{L,Lo}$ per L su L_O è un programma tale che $[\![int^{L,Lo}]\!]:(Prog^L \times D) \rightarrow D$ e

$$[\![int^{L,Lo}]\!](P^L,in)=[\![P^L]\!](in)$$

Si osserva che un programma è di fatto una sequenza di istruzioni rappresentate da simboli. Questo significa che anche ad alto livello può essere assimilato al concetto di dato, e quindi può essere passato come input.



In questo caso non abbiamo una traduzione esplicita , ma solo un processo di decodifica. L'interprete esegue ogni istruzione di L simulandola usando un certo insieme di istruzioni di Lo. Questa non è una vera traduzione in quanto il codice in Lo viene direttamente eseguito, e non prodotto in output. Di recente con i linguaggi di scripting, la soluzione puramente interpretata è ritornata in uso.

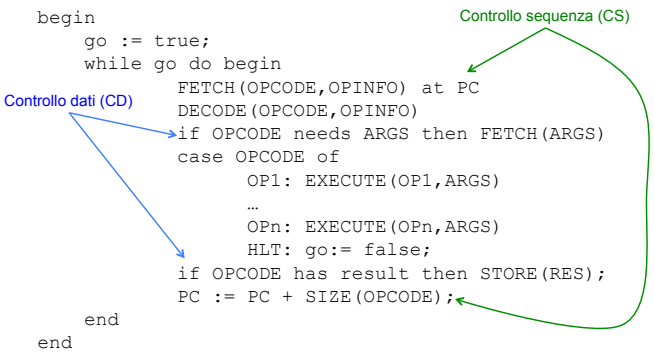
SOLUZIONE INTERPRETATIVA: OPERAZIONI

Esattamente come abbiamo visto nell'architettura di von Neumann, un interprete è di fatto un ciclo che attraverso una serie di operazioni e funzionalità (indipendenti dal linguaggio) esegue per simulazione le istruzioni del linguaggio. Tali operazioni sono:

- ***Elaborazione dei dati primitivi***: I dati primitivi sono dati rappresentabili in modo diretto nella memoria, ad esempio nella macchina fisica i numeri sono dati primitivi. Le operazioni aritmetiche per elaborare questi dati sono direttamente implementate nella struttura della macchina (operazioni primitive);
- ***Controllo di sequenza delle esecuzioni***: Consiste nella gestione del flusso di esecuzione delle istruzioni, non sempre sequenziale. L'interprete dispone infatti di adeguate strutture dati (ad esempio per memorizzare l'indirizzo della prossima istruzione da eseguire) che sono manipolate con operazioni specifiche (ad esempio aggiornamento dell'indirizzo);
- ***Controllo dei dati***: Per eseguire le istruzioni è, in generale, necessario recuperare i dati necessari, mediante l'utilizzo anche di strutture ausiliarie. Servono per controllare gli operandi e, in generale, i dati che devono essere trasferiti dalla memoria all'interprete e viceversa. Riguardano le modalità di indirizzamento della memoria e l'ordine con cui recuperare gli operandi. A volte necessitano di strutture ausiliarie;
- ***Controllo della memoria***: Sia il programma che i dati vanno memorizzati nella macchina, quindi è necessario gestire processi di allocazione della memoria per dati e programmi. Nel caso di macchine astratte vicino all'HW la gestione è abbastanza semplice. Nel caso limite della macchina fisica i dati potrebbero restare sempre nelle stesse locazioni. Nelle macchine astratte (SW) invece esistono costrutti di allocazione e deallocazione di memoria che richiedono opportune strutture dati (pile) e operazioni dinamiche.

SOLUZIONE INTERPRETATIVA: STRUTTURA

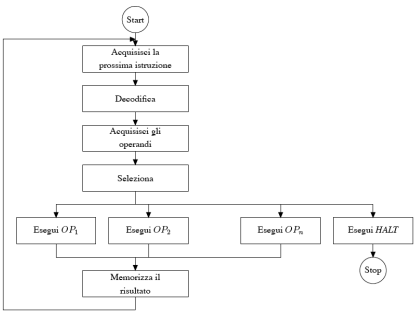
Stabilite le operazioni di cui un interprete ha bisogno vediamo come concretamente opera. Il ciclo di esecuzione comune a tutti gli interpreti è:




```
begin
    go := true;
    while go do begin
            FETCH(OPCODE,OPINFO) at PC
            DECODE(OPCODE,OPINFO)
            if OPCODE needs ARGS then FETCH(ARGS)
            case OPCODE of
                  OP1: EXECUTE(OP1,ARGS)
                  …
                  OPn: EXECUTE(OPn,ARGS)
                  HLT: go:= false;
            if OPCODE has result then STORE(RES);
            PC := PC + SIZE(OPCODE);
    end
end
```

1. Acquisizione prossima istruzione da eseguire
2. Decodifica dell'istruzione per estrarre operazione e operandi
3. Prelievo dalla memoria degli operandi nel numero richiesto e nelle modalità individuate
4. Esecuzione dell'operazione
5. Memorizzazione dell'eventuale risultato nella memoria
6. Ripetizione del fino al raggiungimento di una operazione di halt (terminazione), se raggiungibile.

Dallo pseudocodice mostrato, non risulta evidente la relazione con il controllo memoria, ma osserviamo che se esiste i tale che Op_i modifica la memoria, ad esempio usando i puntatori, allora interviene il controllo della memoria. Invece, dentro le EXECUTE e nel calcolo di PC intervengono le operazioni per l'elaborazione dei dati primitivi.



SOLUZIONE COMPILATIVA: COMPILATORE

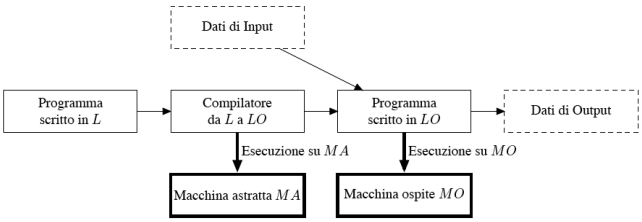
Notazione:
$Prog^L$ = insieme di programmi scritti in L
D = Insieme di dati (input e output)
$P^L \in Prog^L$, in, out∈D
$[\![P^L]\!]: D \rightarrow D$ semantica di P^L

Un compilatore è un programma $comp^{Lo,L}$ che traduce (preservandone la semantica/funzionalità) programmi scritti in L in programmi scritti in L_O, e quindi eseguibili direttamente sulla macchina astratta per L_O. Esattamente come per l'interprete, anche per il compilatore, un programma può essere assimilato al concetto di dato, e quindi passato come input.

Compilatore da L a Lo:
Dato $P^L \in Prog^L$, un compilatore $comp^{L,Lo}$ da L a Lo è un programma tale che $[\![comp^{L,Lo}]\!]: Prog^L \rightarrow Prog^{Lo}$ e

$[\![comp^{L,Lo}]\!](P^L)=P^{Lo}$ tale che
$\forall in \in D.\ [\![P^{Lo}]\!](in)=[\![P^L]\!](in)$

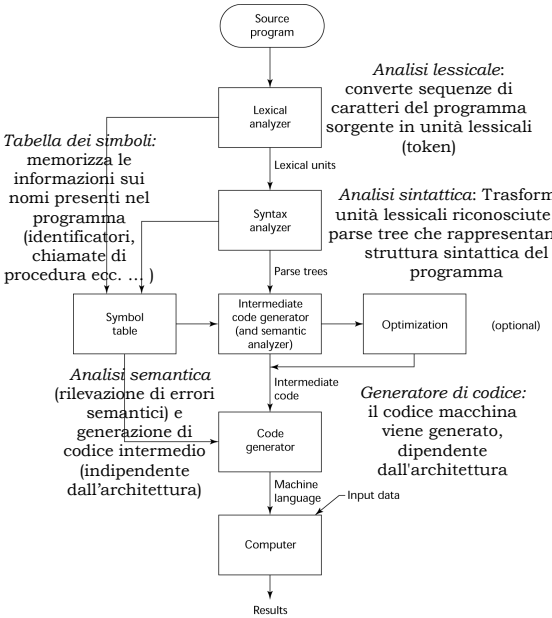
In questo caso abbiamo una traduzione esplicita, in quanto il codice in L viene prodotto come output e non eseguito. Quindi, per poter eseguire il programma P^L con input in dovremo per prima cosa eseguire $comp^{L,Lo}$ con P^L come input. Questa esecuzione avverrà sulla macchina astratta M_A del linguaggio in cui è scritto il compilatore (potenzialmente diverso sia da L che da L_O). Questo produce come risultato un altro programma (compilato) P^{Lo}, scritto in Lo. A questo punto possiamo eseguire P^{Lo} su M_{Lo} (macchina ospite, ovvero macchina astratta per L_O) con input in. Molti linguaggi imperativi ad alto livello (es. C) sono compilati.



SOLUZIONE COMPILATIVA: STRUTTURA

La compilazione deve tradurre un programma da un linguaggio ad un altro preservandone la semantica: dobbiamo avere la certezza che il programma compilato faccia esattamente quello che faceva il programma originale/sorgente. Per questo un compilatore è uno strumento più difficile da progettare rispetto all'interprete. L'esecuzione di un compilatore passa attraverso varie fasi:

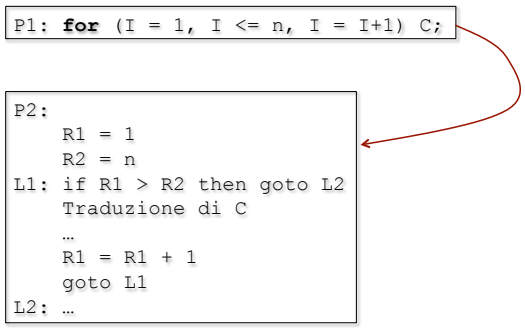
- Analisi lessicale (Scanner): Spezza un programma nei componenti lessicali primitivi (unità logicamente significative), chiamati tokens (identificatori, numeri, parole riservate …). I token formano linguaggi regolari;
- Analisi sintattica (Parser): Crea una rappresentazione ad albero della sintassi del programma dove ogni foglia è un token e le foglie lette da sx a dx costituiscono frasi ben formate del linguaggio. Tale albero costituisce la struttura logica del programma, quando non è possibile costruire l'albero significa che qualche frase è illegale. In tal caso la compilazione si blocca con un errore. Le frasi di token formano linguaggi CF.



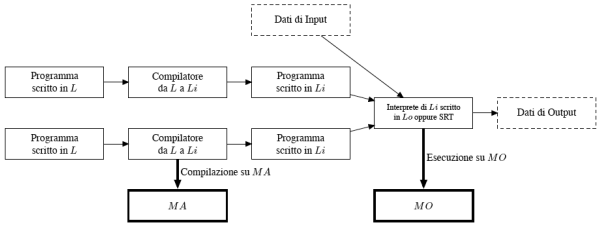
INTERPRETE VS COMPILATORE

Entrambe le soluzioni, nella loro forma pura hanno vantaggi e svantaggi:

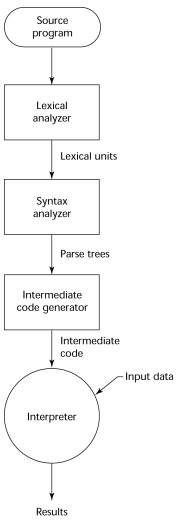
- Implementazione **interpretativa** pura: Interpretare al momento dell'esecuzione permette di interagire in modo diretto con l'esecuzione del programma (importante nel debugging). Un interprete è più veloce da sviluppare rispetto ad un compilatore ed usa meno memoria. D'altra parte i tempi di decodifica si sommano a quelli di esecuzione, ogni volta che una certa istruzione viene eseguita. In sintesi: nessun costo di traduzione, esecuzione lenta, scarsa efficienza della macchina M_L, buona flessibilità e portabilità, facilità di interazione a run-time (es. debugging);
- Implementazione **compilativa** pura: Trascurando i tempi di compilazione, l'esecuzione del programma è più efficiente anche perché nella fase di compilazione il codice viene, in generale, ottimizzato. L'interazione è invece più difficile. Un errore a runtime è difficile da associare all'esatto comando del codice sorgente che lo ha generato, per questo il debugging è più difficile. In sintesi: costi di traduzione, esecuzione veloce, progettazione difficile, dipendente dalla distanza fra L e Lo, buona efficienza (decodifica a carico del compilatore e ottimizzazioni), scarsa flessibilità, perdita di informazione sulla struttura del programma sorgente.



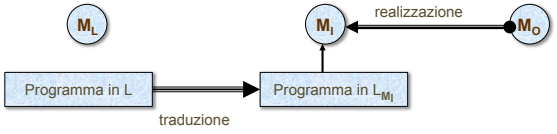
SOLUZIONE REALE: IBRIDA



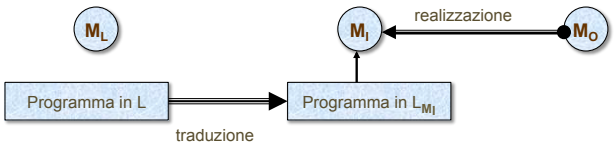
Esiste un compromesso tra compilatore e interprete, ovvero una soluzione ibrida dove il linguaggio ad alto livello viene compilato in un linguaggio a più basso livello che poi viene interpretato. In questo caso, consideriamo il linguaggio L, ad alto livello, per il quale dobbiamo realizzare la macchina astratta M_L. L viene quindi tradotto in un linguaggio intermedio L_{Mi} la cui macchina astratta M_I consiste in un interprete del linguaggio L_{Mi} sulla macchina ospite M_O.



La separazione non è comunque netta: si tende ad interpretare i costrutti lontani da Mo, mentre si compila il resto. Le implementazioni compilative ed interpretate sono quindi gli estremi di quello che avviene in pratica. Osserviamo che l'interprete reale (quando M_I è sostanzialmente diversa da Mo) lavora su una rappresentazione interna dei programmi che non coincide quasi mai con quella esterna. Il passaggio avviene attraverso una traduzione (compilazione) da L ad un linguaggio intermedio (quello interpretato). Ad esempio alcune istruzioni di I/O richiederebbero varie centinaia di istruzioni, per cui si preferisce eseguirle mediante una chiamata ad un programma (o al SO) che al momento dell'esecuzione simula tali istruzioni. Se invece M_I è più vicina a Mo, ovvero se il linguaggio intermedio e il linguaggio macchina non sono molto distanti, per simulare la macchina intermedia può bastare l'interprete della macchina ospite esteso con opportuni programmi detti "supporto a run-time" - SRT (questo avviane ad esempio per il linguaggio C).



SINTESI



Quindi nell'evoluzione dei linguaggi di programmazione esistono fondamentalmente tre situazioni possibili:

- ***Interprete puro***: $M_L = M_I$ (Interprete per L realizzato sulla macchina ospite M_O)
  - Alcune implementazioni (vecchie!) di linguaggi logici e funzionali (LISP, PROLOG)
  - Linguaggi di scripting (JS, PHP, …)
- ***Compilatore***: Macchina intermedia M_I realizzata per estensione sulla macchina ospite M_O (RTS, nessun interprete)
  - Linguaggi imperativi come C, C++, PASCAL
- ***Implementazione mista***: Traduzione dei programmi da L ad un linguaggio intermedio $L_{Mi.}$ I programmi L_{Mi} sono poi interpretati sulla macchina ospite M_O
  - Java (con linguaggio intermedio Java bytecode), Pascal (con linguaggio intermedio P-code)
  - i "compilatori" per linguaggi funzionali e logici (LISP, PROLOG, ML)
  - alcune (vecchie!) implementazioni di Pascal (Pcode)

COMPILATORI = INTERPRETI "SPECIALIZZATI"

Interpreti e compilatori non sono comunque due realizzazioni/implementazioni indipendenti. Esiste infatti un legame semantico tra interprete e compilatore, e per mostrarlo abbiamo bisogno di un altro tipo di programma che non implementa linguaggi, ma trasforma programmi.

Lo ***specializzatore*** valuta il programma su una parte dell' input, ottenendo un programma *specializzato* rispetto a tale input e, per questo, più efficiente. Intuitivamente, se abbiamo un programma in cui una parte dei dati in input è nota e non cambia, allora possiamo trasformare il programma in modo tale che le computazioni sulla parte nota dell'input siano state già svolte. Si tratta quindi di trasformazioni all'interno dello stesso linguaggio per migliorare l'efficienza (nota anche come Valutazione parziale). Lo specializzatore non interpreta quindi un programma, ma ne restituisce uno nuovo a partire da uno esistente e da una parte del suo input. Un tale programma esiste grazie al teorema s-m-n (Fondamenti)

Specializzatore per L:
Dato $P^L \in Prog^L$, uno specializzatore $spec^L$ per L è un programma tale che $[\![spec^L]\!]: (Prog^L \times D) \longrightarrow Prog^L$

$$\forall in,d \in D.[\![spec^L]\!](P^L,d)=Q^L \text{ tale che } [\![P^L]\!](d,in)=[\![Q^L]\!](in)$$

Esiste un risultato formale (I Proiezione di Futamura, Fondamenti) che permette di caratterizzare i compilatori a partire da un interprete e uno specializzatore: ***Specializzando un interprete rispetto al programma otteniamo un compilatore***. Quindi a partire da un interprete si garantisce l'esistenza di un compilatore, e in generale di trasformatori sintattici che preservano la semantica (ottimizzatori, offuscatori…).

Dato $P^L \in Prog^L$, $spec^{L'}$ uno specializzatore per L', $int^{L,Lo}$ un interprete da L a Lo scritto in L', allora

$$P^{L'} = [\![spec^{L'}]\!](int^{L,Lo},P^L)$$

$P^{L'}$ eredita l'algoritmo di P^L, ne esegue fedelmente le operazioni nello stesso ordine, ma con lo stile di programmazione e il linguaggio di int.