Linguaggi

VR443470

marzo 2023

Indice

1 Introduzione

1.1 Nascita dei linguaggi

Il problema **principale dell'informatica** consiste nel voler far eseguire ad una macchina **algoritmi** che manipolano **dati**. Con il termine "macchina" si intende un dispositivo **programmabile**, ovvero un calcolatore, che può eseguire un insieme di istruzioni chiamate programmi, ricevuti come input (**macchine universali**).

In particolare, i computer moderni hanno un'architettura che nasce da quella di Von Neumann.

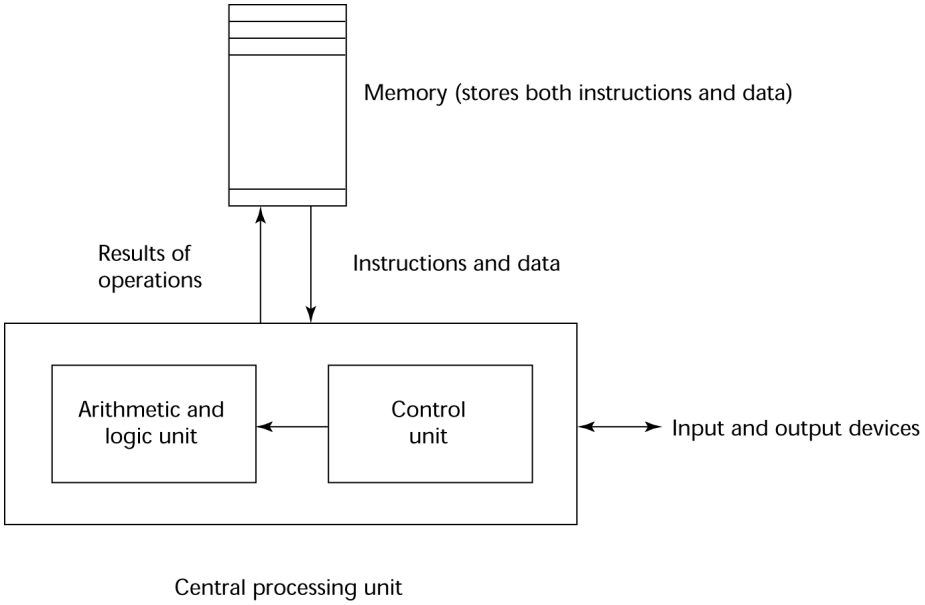


Figura 1: Architettura di Von Neumann.

Per operare su tali macchine, fu necessario inserire una CPU per eseguire algoritmi e operare sui dati in memoria. Il primo linguaggio che consente di programmare tale architettura è quello basato sull'implementazione dell'architettura stessa, ovvero la programmazione con schede perforate per esempio.

1.2 Definizioni

1.2.1 Algoritmo

Un **algoritmo** **è una sequenza finita di passi primitivi di calcolo descritti mediante una frase ben formata (programma) in un linguaggio di programmazione**. In altre parole, un algoritmo scompone un calcolo complesso in passi elementari di computazione. Esso è dunque un concetto astratto che trova la sua forma concreta in un programma che è la sequenza finita di istruzioni.

Data la definizione, è necessario precisare che:

- **Un programma non è necessariamente un algoritmo**, poiché una frase grammaticalmente corretta potrebbe non avere significato;
- **Lo stesso algoritmo può avere concretizzazioni diverse**, ovvero lo stesso algoritmo può essere implementato da infiniti programmi.

1.2.2 Dati

I programmi sono la concretizzazione degli algoritmi, i quali manipolano i **dati**. Essi sono informazioni memorizzate sia concretamente in celle di memoria, sia astrattamente in elementi che il linguaggio di programmazione può manipolare, ovvero le **variabili**.

1.2.3 Sintassi e semantica

La trasformazione (scrittura) di un programma in un determinato linguaggio di programmazione rappresenta la **sintassi**, mentre l'effetto della sua esecuzione e la trasformazione dei dati eseguita, costituisce la **semantica**.

1.2.4 Linguaggio matematico e logico

Il **linguaggio matematico** è una notazione rigorosa per rappresentare funzioni, ma non sempre oggetti infiniti e computazioni, cioè passi di calcolo.

Il **linguaggio logico** sono regole e assiomi che rendono possibile specificare il processo di computazione, in modo implicito, e consente di rappresentare formalmente oggetti infiniti in modo finito, ma non computazioni infinite.

1.2.5 Linguaggio di programmazione

Un **linguaggio di programmazione** consente di specificare in modo accurato esattamente le primitive del processo di computazione, con la rigorosità e la potenza della logica.

1.2.6 Programma

Un **programma** è un insieme finito di istruzioni e costrutti del linguaggio di programmazione.

1.3 Aspetti di progettazione

1.3.1 Leggibilità

La **leggibilità** (*readability*) è la sintassi chiara, l'assenza di ambiguità, la facilità di lettura e la comprensione dei programmi.

I fattori che contribuiscono alla leggibilità sono:

1. La **semplicità di un linguaggio**, per esempio pochi ed essenziali costrutti base. Infatti, un linguaggio inizia ad essere complicato quando per poter fare la stessa cosa si possono seguire molti percorsi diversi. Un altro fattore di complicazione è l'overloading degli operatori, ovvero quando il simbolo di un operatore ha molteplici significati.

2. L'**ortogonalità** della progettazione di un linguaggio. Un elemento di un programma è ortogonale se è indipendente dal contesto di utilizzo all'interno di esso. Più un programma è ortogonale, meno eccezioni alla regola esistono.

3. **Presenza di strumenti per la definizione di tipi di dati e strutture dati**. Ad esempio l'uso di booleani al posto dei valori interi.

4. **Struttura della sintassi**, come parole chiave significative ad esempio.

1.3.2 Scrivibilità

La **scrivibilità** (*writability*) è la facilita di utilizzo di un linguaggio per creare programmi, la facilità di analisi e la verifica dei programmi. I fattori che influenza la leggibilità sono gli stessi della scrivibilità.

In breve i fattori che che contribuiscono alla scrivibilità sono:

1. **Semplicità e ortogonalità**. La presenza di pochi costrutti consente al programmatore di conoscerli in gran parte e di sfruttare al massimo il linguaggio. Stessa cosa per il numero di primitive.

2. **Supporto per l'astrazione**. La possibilità di utilizzare strutture o operazioni complesse in modi che permettono di ignorare i dettagli. Esistono due tipi di astrazione: processi e dati.

3. **Espressività**. Si riferisce a molte caratteristiche, per esempio mettere a disposizione un insieme di modi relativamente convenienti per specificare operazioni.

1.3.3 Affidabilità e costo

Per **affidabilità** (*reliability*) si intende la conformità alle sue specifiche. Mentre per **costo**, letteralmente il costo complessivo di utilizzo.

Un **programma** viene categorizzato come **affidabile** se soddisfa le seguenti condizioni:

1. ***Type checking***, ovvero il controllo degli errori di tipo. Viene eseguito spesso a tempo di compilazione poiché risulta costoso.
2. **Gestione delle eccezzioni**. Gestire gli errori run-time per consentire la continuazione dell'esecuzione e l'attuazione di eventuali misure correttive.
3. **Presenza di potenziali aliasing**. La presenza di due o più metodi di riferimento per la stessa locazione di memoria è un problema.

Mentre le **specifiche di costo** riguardano:

- L'addestramento di programmatore per usare il linguaggio
- Scrittura di programma
- Compilazione dei programmi
- Esecuzione dei programmi
- Sistema di implementazione del linguaggio, ovvero la disponibilità di compilatori liberi
- Poca affidabilità fanno lievitare i costi
- Mantenimento dei programmi

1.4 Classificazione dei linguaggi

I linguaggi possono essere classificati per: **metodo di computazione** e **per caratteristiche**.

1.4.1 Metodo di computazione

I linguaggi possono essere a:

- **Basso livello**. Questi linguaggi hanno caratteristiche strettamente dipendente all'architettura su cui si sta programmando. Per esempio:
  - Linguaggio binario che non fa distinzione tra dati e programmi;
  - Assembly, linguaggio strutturato molto basso, vicino al linguaggio macchina.
- **Alto livello**. Questi linguaggi consentono una programmazione strutturata in cui dati ed istruzioni hanno rappresentazioni diverse. Esistono tre tipi:
  - **Linguaggi imperativi** che descrivono come **concetto chiave** l'elemento fondamentale dell'architettura di Von Neumann, ovvero la **cella di memoria**.
    Il concetto di variabile rappresenta l'astrazione logica della cella.
    Il concetto di assegnamento rappresenta l'operazione primitiva di modifica della cella di memoria e dunque dello stato della macchina.
    Nei linguaggi imperativi, gli assegnamenti vengono controllati in modo sequenziale, condizionale e ripetuti.
  - **Linguaggi funzionali** sono molto vicini alla matematica. Essi descrivono i passi di calcolo come funzioni matematiche. Il *core* principale si concentra sulla composizione e applicazione di funzioni.
    Una variabile viene intesa come un'incognita matematica e sostituita come se fosse un *placeholder* all'interno del linguaggio. Infatti, essa non può cambiare nel tempo durante la computazione.
  - **Linguaggi logici** usano la logica, ovvero eseguono pattern matching. Come passo di calcolo primitivo utilizzano l'unificazione o la sostituzione.

1.4.2 Per caratteristiche

La classificazione per caratteristiche è una metodologia utilizzata principalmente all'inizio dell'era informatica per studiare le caratteristiche di base quali: strutture di base di controllo, strutture per i dati, efficienze nell'esecuzione.

Andando avanti con il tempo, la classificazione si è focalizzata su caratteristiche aggiuntive, ovvero le strutture di base rimangono le stesse, ma ad esse vengono aggiunte nuove caratteristiche. In questo modo viene migliorata la soluziona di specifici problemi.

1.5 Implementazione dei linguaggi

L'**implementazione di un linguaggio** ha un collegamento stretto con il funzionamento della macchina su cui deve essere eseguito. Infatti, l'implementazione riguarda le metodologie per rendere comprensibile alla macchina da programmare il linguaggio scelto. Per farlo, è necessario introdurre il funzionamento di una macchina basata sull'architettura di Von Neumann. Essa si basa sulla ripetizione di un ciclo che costituisce l'**interprete** del linguaggio che la macchina riconosce:

- Lettura dell'istruzione dalla memoria (*fetch*)
- Decodifica dell'istruzione (*decode*)
- Lettura di eventuali operandi
- Memorizzazione ed esecuzione del risultato (*exec*)

Figura 2: Ciclo di esecuzione delle istruzioni.

1.5.1 Macchina astratta

L'implementazione di un linguaggio **significa** considerare una macchina astratta poiché lavorando ad alto livello, le istruzioni vengono interpretate e dunque si ignorano momentaneamente il linguaggio binario e la macchina fisica.

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 programmi scritti in L.

La collezione di strutture dati ed algoritmi è necessario per:

- Acquisire la prossima istruzione
- Gestire le chiamate e 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

In altre parole, una **macchina astratta** è la combinazione di una memoria che immagazzina i programmi e di un interprete che esegue istruzioni dei programmi.

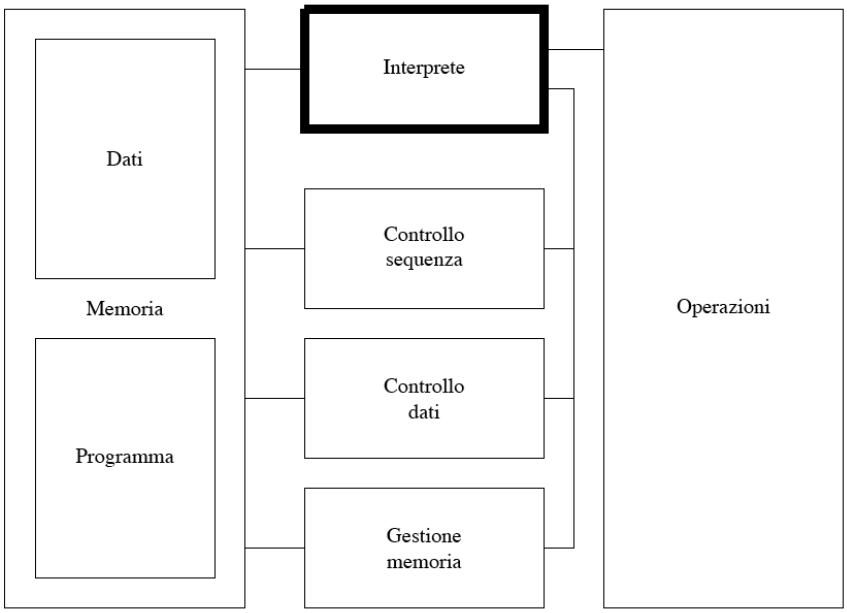


Figura 3: Rappresentazione di una macchina astratta.

Il **linguaggio** L riconosciuto (interpretato) dalla macchina astratta M_L viene chiamato **linguaggio macchina**. Formalmente, è l'insieme di tutte le stringhe interpretabili dalla macchina astratta M.

1.5.2 Realizzazione di una macchina astratta a vari livelli

Qualsiasi macchina astratta, per essere eseguita, deve prima o poi utilizzare qualche dispositivo hardware. Questo però non significa che tutte le macchine sono realizzate a livello hardware. Infatti, la realizzazione di una macchina astratta può avvenire tre categorie:

- **Realizzazione hardware (HW)**. Sempre possibile e concettualmente semplice. Il linguaggio macchina è il linguaggio fisico/binario e si realizza mediante dispositivi fisici. Data la sua lontananza dai linguaggi ad alto livello, la loro programmazione risulta complessa. Questo è uno dei tanti motivi per cui viene usata solo per sistemi dedicati.
- **Realizzazione firmware (FW)**. Le strutture dati e gli algoritmi vengono simulati nella macchina mediante microprogrammi. Il linguaggio macchina è a basso livello e consiste in microistruzioni che specificano le operazioni di trasferimento dati tra registri. Il vantaggio è dato dalla velocità e la flessibilità maggiore rispetto all'hardware.
- **Realizzazione software (SW)**. Le strutture dati e gli algoritmi vengono realizzati tramite un linguaggio implementato. In questo modo è possibile scrivere programmi che interpretano i costrutti del linguaggio macchina simulando le funzionalità della macchina. La velocità viene diminuita ma aumenta molto la flessibilità.

1.5.3 Realizzazione software e livelli di astrazione

Con la realizzazione software, vengono utilizzati linguaggi di programmazione ad alto livello poiché essi implementano una struttura suddivisa a livelli di astrazione. Ogni livello coopera in modo sequenziale ma allo stesso tempo è indipendente.

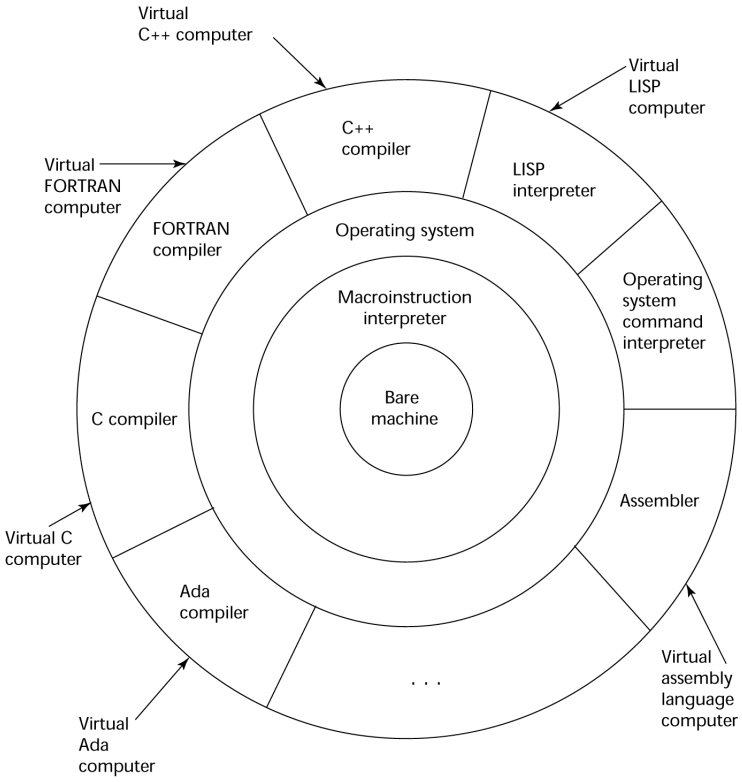
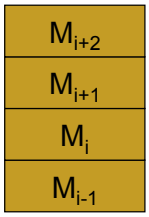


Figura 4: Livelli di astrazione utilizzati dai linguaggi ad alto livello.

Quindi, la macchina può essere vista come una stratificazione di livelli di astrazione:



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)

1.6 Realizzazione di una macchina a livello software/firmware

Sia L un linguaggio da implementare e sia M_{L_0} una macchina astratta a disposizione che ha come linguaggio macchina L_0. La macchina astratta M_{L_0} è il livello su cui si vuole implementare il linguaggio L e che mettette a disposizione di M_L le sue funzionalità.

Dunque, la realizzazione di una macchina astratta M_L consiste nel realizzare una macchina che "traduce" il linguaggio L (ad alto livello per esempio) in linguaggio macchina L_0, ovvero che interpreta tutte le istruzioni di L come istruzioni di L_0. La traduzione avviene tramite due metodi a scelta:

- **Soluzione interpretativa**. Simulazione dei costrutti della macchina astratta M_L da realizzare, mediante programmi scritti in L_0;
- **Soluzione compilativa**. Traduzione esplicita dei programmi di L in corrispondenti programmi di L_0.

1.6.1 Soluzione interpretativa: interprete

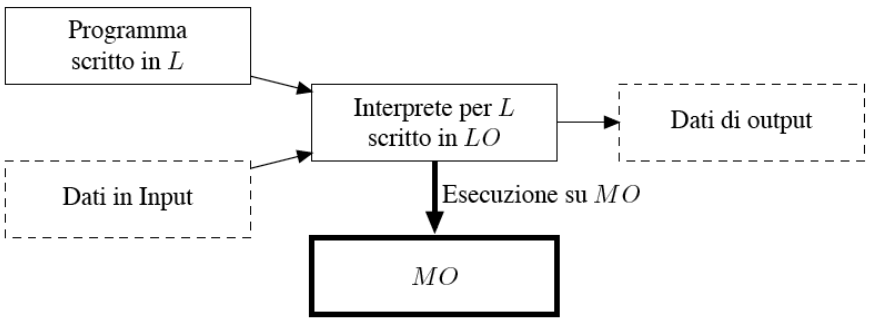
La soluzione interpretativa prevede l'utilizzo di un interprete per la realizzazione di una macchina astratta. Un **interprete** è un programma $\text{int}^{\text{L}_0,\text{L}}$ che esegue, sulla macchina astratta per L_0, programmi P^L, scritti nel linguaggio di programmazione L, su un input fissato appartenente all'insieme di dati (input e output). In breve, un interprete è una **macchina universale** che preso un programma e un suo input, lo esegue su quell'input usando solo funzionalità messe a disposizione dal livello (macchina astratta) sottostante.

Notazioni

- $Prog^L$ è l'insieme di programmi scritti nel linguaggio di programmazione L;
- D è l'insieme di dati, ovvero input e output;
- P^L è il programma scritto nel linguaggio di programmazione L;
- Relazioni ovvie: $P^L \in Prog^L$ e $in, out \in D$;
- $[\![P^L]\!] : D \longrightarrow D$ è la notazione utilizzata per indicare che l'esecuzione del programma scritto nel linguaggio di programmazione L con input in è uguale all'output out. Quindi $[\![P^L]\!](in) = out$.

Un **interprete formalmente** (**definizione**) è esprimibile nel seguente modo. Si consideri un interprete da L a L_0: dato $P^L \in Prog^L$ e $in \in D$, un interprete int^{L,L_0} per L su L_0 è un programma tale che $[\![int^{L,L_0}]\!] : \left(Prog^L\right) \longrightarrow D$ e dunque $[\![int^{L,L_0}]\!]\left(P^L, in\right) = [\![P^L]\!](in)$.

Anche un programma può essere utilizzato come dato di input in un altro programma. Si osservi il seguente diagramma:



Si noti come un programma scritto nel linguaggio L, insieme ad eventuali altri input, viene interpretato da un programma creato appositamente per eseguire questo compito su L, ma scritto in L_0. L'**esecuzione comporta una decodifica e non una traduzione esplicita**. Infatti, l'interprete simula ogni istruzione di L utilizzando un certo insieme di istruzioni di L_0. Questa è la base dei linguaggi di scripting.

1.6.2 Soluzione interpretativa: operazioni e struttura

Un interprete può eseguire una serie di **operazioni**:

- **Elaborazione dei dati primitivi**. I dati primitivi sono dati rappresentabili in modo diretto nella memoria, per esempio i numeri. Le elaborazioni di essi, sono implementate direttamente nella struttura della macchina;
- **Controllo di sequenza delle esecuzioni**. Non è altro che la gestione del flusso di esecuzione delle istruzioni, le quali non sempre sono sequenziali, tramite alcune strutture dati;
- **Controllo dei dati**. Recupero dei dati necessari per eseguire le istruzioni. I dati possono riguardare le modalità di indirizzamento della memoria e l'ordine con cui recuperare gli operandi;
- **Controllo della memoria**. È necessaria una gestione della memoria per allocare dati e programmi. Cambia a seconda del tipo di realizzazione della macchina astratta:
  - Realizzazione hardware (HW): la gestione è semplice poiché nella peggiore delle ipotesi, i dati potrebbero essere rimasti sempre nelle stesse locazioni.
  - Realizzazione software (SW): la gestione è complessa ed esistono costrutti di allocazione e deallocazione che richiedono alcune strutture dati (e.g. pile) e operazioni dinamiche.

Date le operazioni elencate, il **ciclo di esecuzione di un interprete** è il seguente:

```
1  begin
2    go := true;
3    while go do begin
4      FETCH(OPCODE, OPINFO) at PC
5      DECODE(OPCODE, OPINFO)
6      if OPCODE needs ARGS then FETCH (ARGS)
7      case OPCODE of
8        OP1: EXECUTE(OP1, ARGS)
9        ...
10       OPn: EXECUTE(OPn, ARGS)
11       HLT: go := false;
12     if OPCODE has result then STORE(RES);
13     PC := PC + SIZE(OPCODE);
14   end
15 end
```

- Righe 1-3: finché go ha il valore true, il codice viene eseguito;
- Riga 4: estrazione dell'istruzione riferita ad OPCODE (controllo sequenza 1 su 2);
- Righe 5: decodifica dell'istruzione estratta alla riga precedente;
- Riga 6: prelievo dalla memoria gli operandi richiesti da OPCODE e nelle modalità individuate (controllo dati 1 su 2);
- Righe 7-11: esecuzione delle operazioni;

- Riga 12: se l'operazione ha un risultato da salvare, allora viene salvato in memoria (controllo dati 2 su 2);
- Riga 13: viene incrementato il *program counter* (PC) per eseguire la prossima istruzione (controllo sequenza 2 su 2).

Il ciclo continua ad essere eseguito finché la variabile `go` ha valore `true`. Inoltre, le istruzioni riferite al program counter sono chiamate istruzioni di **controllo sequenza** (CS) perché manipolano e accedono al program counter. Mentre le operazioni di memorizzazione/estrazione sugli argomenti si chiamano **controllo dati** (CD).

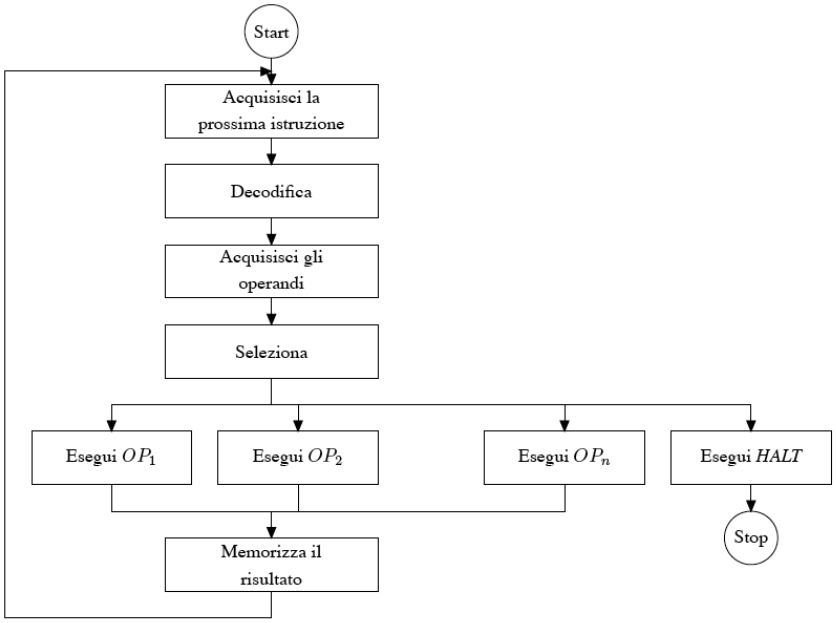


Figura 5: Diagramma a blocchi della struttura di un interprete.

1.6.3 Soluzione interpretativa: pro e contro

- **Pro:**
  - **Facilità di interazione *run-time***. Interpretazione al momento dell'esecuzione consente di interagire direttamente con l'esecuzione del programma (*debugging*);
  - Velocità nello sviluppo applicativo di un interprete, quindi **tempi ridotti per la sua creazione**;
  - Utilizzo della **memoria ridotto** rispetto ad un compilatore.
- **Contro:**
  - Tempi di decodifica sommati a quelli d'esecuzione ogni volta che un'istruzione viene eseguita, si traduce in un'**esecuzione lenta** e quindi una scarsa efficienza della macchina.

1.6.4 Soluzione compilativa: compilatore

Un **compilatore** è un programma $\text{comp}^{L_0,L}$ che **traduce**, preservando semantica e funzionalità, programmi scritti nel linguaggio di programmazione L in programmi scritti in L_0, e quindi eseguibili direttamente sulla macchina astratta per L_0. Come l'interprete, anche il compilatore accetta un programma come input poiché viene considerato come dato.

Notazioni

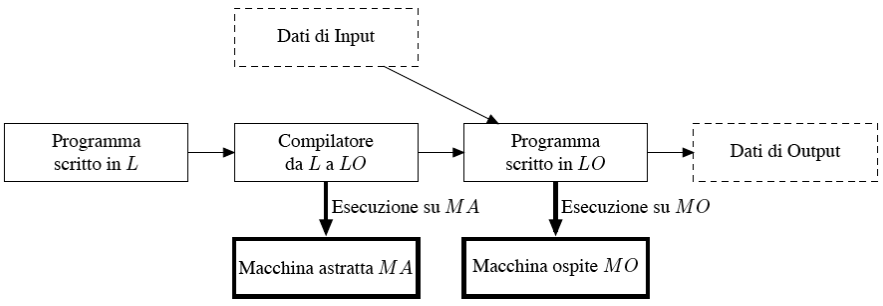
- $Prog^L$ è l'insieme di programmi scritti nel linguaggio di programmazione L;
- D è l'insieme di dati, ovvero input e output;
- P^L è il programma scritto nel linguaggio di programmazione L;
- Relazioni ovvie: $P^L \in Prog^L$ e $in, out \in D$;
- $[\![P^L]\!] : D \longrightarrow D$ rappresenta la semantica di P^L.

Un **compilatore formalmente** (**definizione**) è esprimibile nel seguente modo. Dato $P^L \in Prog^L$, un **compilatore** comp^{L,L_0} da L a L_0 è un programma tale che $[\![comp^{L,L_0}]\!] : Progr^L \longrightarrow Progr^{L_0}$ e:

$$[\![comp^{L,L_0}]\!]\left(P^L\right) = P^{L_0} \text{ tale che } \forall in \in D.[\![P^{L_0}]\!](in) = [\![P^L]\!](in)$$

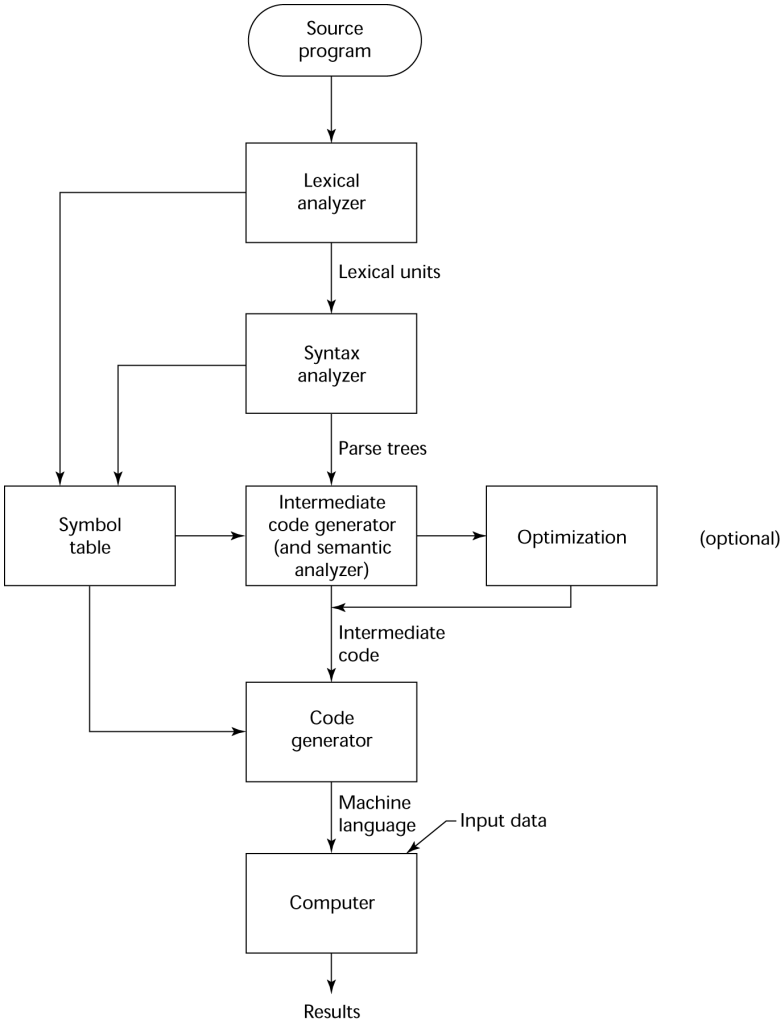
Ovvero che l'esecuzione della compilazione del linguaggio L a L_0 con input il programma scritto in L, l'output sia uguale al programma scritto nel linguaggio L_0; tale che per ogni input appartenente all'insieme dei dati, l'esecuzione del programma scritto in L_0 con input in, sia uguale all'esecuzione del programma scritto in L con input in.

Con un compilatore, la **traduzione** è **esplicita** poiché il codice in L viene prodotto come output e non eseguito. Quindi, per eseguire il programma P^L con input in, è necessario prima eseguire $comp^{L,L_0}$ con P^L come input. L'esecuzione avverrà sulla macchina astratta M_A del linguaggio in cui è scritto il compilatore. Il risultato dunque è un altro programma (compilato) P^{L_0}, scritto in L_0. Solo a questo punto è possibile eseguire P^{L_0} su M_{L_0} con input in. Un **esempio** di linguaggio compilato è il C.



1.6.5 Soluzione compilativa: struttura

La compilazione deve tradurre un programma da un linguaggio ad un altro preservandone la semantica: si deve avere la certezza che il programma compilato faccia esattamente quello che faceva il sorgente. L'**esecuzione** di un compilatore si articola in varie fasi:



- **Analisi lessicale** (*Lexical analyzer*), divide il programma in componenti sintattici primitivi chiamati **tokens** (identificatori, numeri, parole riservate). I *tokens* sono coloro che formano i linguaggi regolari.
  In altre parole, l'**analisi lessicale converte caratteri del programma sorgente in unità lessicali**;

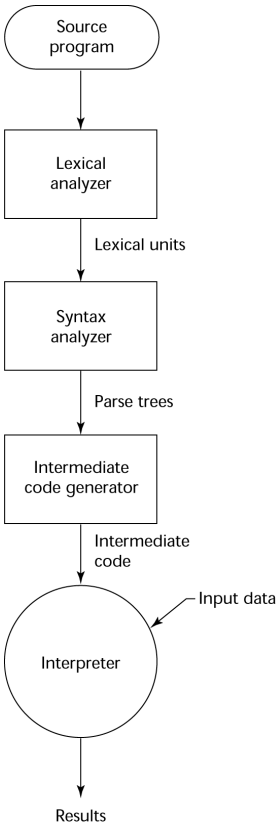
- **Analisi sintattica** (*Syntax analyzer*), crea una rappresentazione ad albero della sintassi del programma. Ogni foglia è un *token* e le foglie lette da sinistra verso destra costituiscono frasi ben formate del linguaggio. Inoltre, l'albero costituisce la struttura logica del programma e dunque nel momento in cui non fosse possibile costruire l'albero, significherebbe che qualche frase è illegale. Questo genere di evento si traduce in un errore di compilazione. Le frasi di token formano linguaggi CF.
  In altre parole, l'**analisi sintattica trasforma unità lessicali in *parse tree* che rappresentano la struttura sintattica del programma**.
- **Tabella dei simboli** (*Symbol table*), memorizza le informazioni sui nomi presente nel programma, come gli identificatori, le chiamate di procedura, ecc.
- **Analisi semantica** (*Semantic analyzer*), consente di rilevare errori semantici, grazie all'analisi semantica, e di generare codice intermedio che ha la caratteristica di essere indipendente dall'architettura (compito del *Intermediate code generator*).
- **Ottimizzazione** (*Optimization*), opzionale, consente di ottimizzare il codice.
- **Generatore di codice** (*Code generator*), viene generato codice macchina che ha la caratteristica di essere dipendente dall'architettura.

1.6.6 Soluzione compilativa: pro e contro

- **Pro:**
  - **Esecuzione molto efficiente**, il codice viene anche ottimizzato;
- **Contro:**
  - **Interazione *run-time* molto difficile**;
  - Un **errore** a *run-time* è **difficile da associare** all'esatto comando del codice sorgente (debugging complesso);

1.6.7 Soluzione reale: ibrido

Nella realtà 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.



Il **procedimento** è il seguente.
Si consideri il linguaggio ad alto livello L per il quale si deve realizzare la macchina astratta M_L.
Il linguaggio 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 netta poiché vengono interpretati i costrutti lontani da M_O, mentre viene compilato il resto. Il passaggio chiave è la traduzione (compilazione) da L ad un linguaggio intermedio (quello interpretato). Questo accade spesso nella realtà, specialmente con le *system call* del sistema operativo. In parole povere, si cerca di trovare una connessione a metà strada.

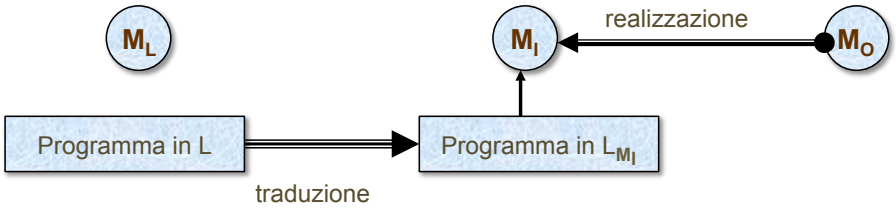


Figura 6: Soluzione ibrida con le *system call*.

1.7 Sintesi

Nell'evoluzione dei linguaggi di programmazione, esistono fondamentalmente tre situazioni possibili:

- **Interprete puro** (paragrafo 1.6.1), $M_L = M_I$ (interprete per L realizzato sulla macchina ospite M_O). Per esempio i linguaggi logici e funzionali, e di scripting (JS, PHP, ...)
- **Compilatore** (paragrafo 1.6.4), macchina intermedia M_I realizzata per estensione sulla macchina ospite M_O. Per esempio i linguaggi imperativi come C, C++, Pascal
- **Implementazione mista** (paragrafo 1.6.7), traduzione dei programmi da L ad un linguaggio intermedio L_{Mi}. I programmi L_{Mi} sono poi interpretati sulla macchina ospite M_O. Per esempio, Java con il suo linguaggio intermedio Java bytecode, Pascal con il suo linguaggio intermedio P-code.

2 Descrivere i linguaggi

Un **linguaggio di programmazione** è un linguaggio naturale, ma con alcune semplificazioni e **non ambiguo**. Quindi, per descriverli è necessario affrontare alcune tematiche:

- Grammatica, o meglio la **sintassi**: costituisce l'**insieme delle regole che consentono di costruire frasi corrette**. Viene quindi individuato l'alfabeto con cui sono costruite le frasi, le parole che compongono le frasi e infine viene eseguito un controllo della grammatica per verificare se le frasi le rispettano (il compito di verificare è assegnato al ***parsing***);
- **Semantica**: nei linguaggi rappresenta la **relazione tra segni** (frasi legali) **e significati** (entità autonome che esistono indipendentemente dai segni utilizzati). Per esempio, la semantica di un programma può essere la funzione matematica calcolata dal programma. Solitamente la semantica viene specificata descrivendo gli effetti della sintassi su una rappresentazione astratta della macchina, chiamata **stato**;
- **Pragmatica**: analisi delle **frasi che hanno lo stesso significato, ma possono essere utilizzate diversamente** in modo dipendente dal contesto linguistico;
- **Implementazione**: aspetti riguardanti la tecnica di implementazione utilizzata, i vincoli dell'architettura o della macchina, l'interfaccia del sistema operativo, gestione degli errori.
  In sintesi, sono tutti gli aspetti che hanno effetto sul funzionamento del linguaggio ma che dipendono dalla macchina su cui esso viene eseguito.

2.1 Sintassi

2.1.1 Definizione e notazione

Le regole sintattiche (**sintassi**) del linguaggio specificano quali stringhe di caratteri sono legali nel linguaggio.

La terminologia linguistica utilizzata è la seguente:

- Una **parola** è una stringa di caratteri su un alfabeto;
- Una **frase** è una sequenza, ben formata, di parole;
- Una **linguaggio** è un insieme di frasi.

Invece, la **terminologia tecnica** utilizzata nel mondo dell'informatica è la seguente:

- Le **parole** vengono chiamate **lessemi**. Un lessema è una **parola con un significato specifico**, nella grammatica corrisponde ad un terminale. Rappresenta anche l'**unità minima sintattica**, ovvero quella a più basso livello di un linguaggio di programmazione (e.g. `begin`). **Per esempio**, in $index = 2$, sia $index$, $=$, che 2 sono lessemi;
- Le **frasi** vengono chiamate ***token***. Essi corrispondono agli elementi delle categorie sintattiche del linguaggio di programmazione, e nella grammatica corrispondono alle sequenze generate dai simboli non terminali. **Per esempio**, con $index = 2$, il token è l'intera definizione;
- Il **programma** è una sequenza/composizione sequenziale, nella grammatica, di frasi ben formate;
- I linguaggi diventa il **linguaggio di programmazione**, ovvero tutti gli strumenti formali che lo definiscono.

2.1.2 Descrivere la sintassi

Nei linguaggi di programmazione, il **linguaggio dei lessemi** è in generale sempre un linguaggio **regolare**, ovvero **riconosciuto** da un automa a stati finiti. Si definisce **riconoscitore**, uno **strumento di riconoscimento che legge in input stringhe sull'alfabeto del linguaggio e decide se la stringa appartiene o meno al linguaggio**.

Per esempio, l'analisi sintattica, la quale riconosce lessemi, di un compilatore.

Sia L un linguaggio su un alfabeto Σ, per **costruire un riconoscitore** è necessario avere un meccanismo R in grado di leggere (input) le stringhe di caratteri e dire (output) se essa appartiene oppure no al linguaggio L. Si ricorda, che questo è possibile perché il linguaggio è regolare.

Il **linguaggio dei *token***, e quindi dei programmi, è in generale un **linguaggio *context-free* (CF)**, quindi viene **generato** da una grammatica CF. Si definisce **generatore**, uno **strumento che genera stringhe di un linguaggio**. Inoltre, un generatore può determinare se la sintassi di una particolare chiave è sintatticamente corretta confrontandola con la struttura del generatore (*parser*).

2.2 Grammatiche *context-free*

Una **grammatica libera dal contesto** (o *context-free*, CF) è una quadrupla $G = \langle V, T, P, S \rangle$, dove:

- V è un insieme finito di variabili, chiamati anche simboli non terminali. Rappresenta le **categorie sintattiche**, ovvero gli **elementi della frase**;
- T è un insieme finito di simboli terminali ($V \cap T = \emptyset$). Rappresenta il **vocabolario**, ovvero la **collezione di lessemi**;
- P è un insieme finito di produzioni; ogni produzione è della forma $A \to \alpha$, dove:
  - $A \in V$ è una variabile
  - $\alpha \in (V \cup T)$

  Questo insieme rappresenta la **collezione di regole di formazione/-composizione**;
- $S \in V$ è una variabile speciale, chiamata simbolo iniziale. Rappresenta la **categoria delle frasi**.

La proprietà CF è anche uno svantaggio per i linguaggi di programmazione, infatti tali grammatiche non riescono a catturare vincoli contestuali. Per esempio, poter utilizzare una variabile se e solo se questa è stata precedentemente dichiarata/definita.

2.3 Notazione BNF

La **BNF** è un metalinguaggio, ovvero un **linguaggio usato per descrivere altri linguaggi**. Venne introdotto perché in passato accadeva che chi progettava ogni implementazione di un compilatore, dava significati diversi agli stessi costrutti. Questo si traduceva che programmi scritti in un linguaggio di programmazione, sviluppati per macchine diverse, non erano confrontabili.

Questa notazione è utilizzata per descrivere grammatiche CF, dove si usano intere parole come simboli terminali, i non terminali sono identificati racchiudendoli tra parentesi angolate <>, e per le produzioni si utilizza il simbolo ::= al posto della freccia.

$$< A > \quad ::= \quad \alpha < B > \gamma \quad | \quad \alpha$$

$$< B > \quad ::= \quad \varepsilon \quad | \quad \beta_1 \quad | \quad \beta_2$$

Il suo significato è esattamente identico a quello delle grammatiche. Quindi:

- **Non terminali** sono astrazioni utilizzate per rappresentare classi di strutture sintattiche;
- **Terminali** sono lessemi;
- Ogni regola ha una parte **sinistra** contenente un **non terminale**;
- Ogni regola ha una parte **destra** contenente una **stringa di terminali e non terminali**.

Esiste anche la **variante EBNF** che aggiunge la possibilità di rappresentare le **opzioni**:

- Le **parentesi quadre** [ ] indicano nessuna o un'occorrenza del contenuto;
- Le **parentesi graffe** { } indicano nessuna o più occorrenze di quanto contenuto.
- La **virgola** , consente di esprimere più opzioni in or logico.

Un esempio:

$$< A >::= \alpha \ [\beta_1, \beta_2] \ \gamma \quad | \quad \alpha \, \gamma \quad | \quad \gamma$$

2.4 Descrivere un semplice linguaggio imperativo

Un **linguaggio può essere descritto in modo informale** descrivendo quali caratteristiche inserire e il loro significato:

- Niente dichiarazioni;
- Solo variabili ed espressioni booleane;
- Assegnamento e composizione sequenziale;
- Comando condizionale;
- Comando iterativo (*loop*).

Nonostante possa essere sufficiente per scrivere un programma corretto e interpretabile da un altro programmatore, esso non è adeguato per costruire un compilatore o un interprete.

Per costruire un compilatore o un interprete è necessario scendere ad un livello più formale.

È necessario che un linguaggio sia descritto a più livelli di astrazione ed ogni strato sia descritto da una grammatica.

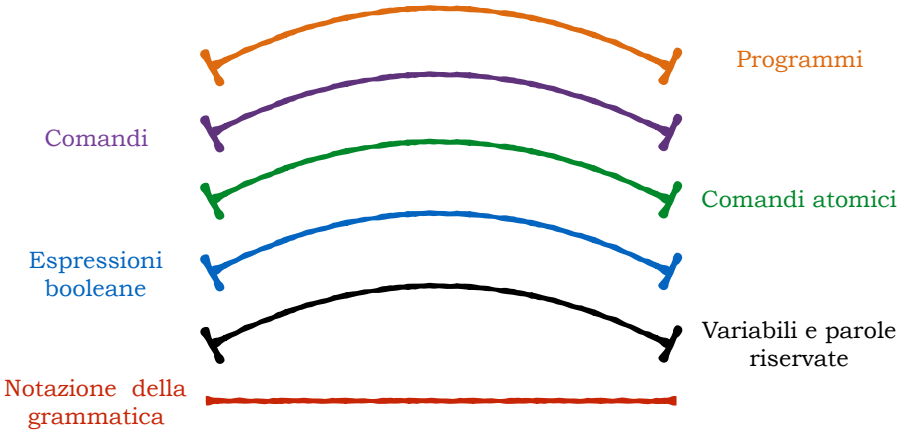


Figura 7: Linguaggio descritto a più livelli di astrazione.

Un esempio di descrizione a più livelli:

$$
\begin{array}{lllll}
< \text{program} > & S & ::= & C & \\
< \text{com} > & C & ::= & A \mid C;C \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C & \\
< \text{atomic com} > & A & ::= & v ::= B & \\
< \text{b} - \text{expr} > & B & ::= & \text{true} \mid \text{false} \mid v \mid (\text{not } B) \mid (B \text{ and } B) \mid (B \text{ or } B) &
\end{array}
$$

2.5 Analisi semantica

Esistono dei vincoli che dipendono dal contesto. Per esempio, l'espressione:

$$I ::= R + 3$$

Potrebbe essere sintatticamente corretta ma illegale nel contesto in cui si trova. Infatti, se il linguaggio fosse fortemente *tipato* e prima non ci fosse una dichiarazione di R e di I, il comando sarebbe illegale.

Quindi, stringhe sintatticamente corrette per una certa grammatica sono legali sono in determinati contesti. Questo **vincolo sintattico** non può essere descritto mediante le grammatiche CF.

Esistono due **soluzioni**:

- Utilizzare grammatiche contestuali (CSG). Tuttavia, non esistono algoritmi lineari per il riconoscimento di stringhe generate, quindi non esistono algoritmi efficienti per effettuare il parser delle stringhe di una CSG.
- Utilizzare controlli *ad hoc*.

A causa dei problemi con le grammatiche contestuali, la scelta ricade nell'utilizzare una specifica del linguaggio mediante una grammatica CF (**sintassi**) e successivamente, come parte della semantica (**semantica statica**) specificare i vincoli contestuali.

2.6 Semantica dinamica

La **semantica** ricerca esattezza e flessibilità:

- **Esattezza**: descrizione precisa e non ambigua di ogni costrutto sintatticamente corretto, per sapere cosa accadrà durante l'esecuzione
- **Flessibilità**: nessuna anticipazione delle scelte che devono essere demandate dall'implementazione.

La **semantica dinamica** risponde ad alcune problematiche, come il significato di alcuni comandi, che cosa fanno i costrutti, generatori di compilatori. Il suo **utilizzo è fondamentale poiché specifica gli effetti della sintassi sulla rappresentazione astratta della macchina**, quest'ultima chiamata **stato**. Quindi, per ogni costrutto, va descritto il significato della sua esecuzione come trasformazione di stato. Infatti, l'astrazione della macchina nel concetto di stato consente di dare al costrutto un significato puro, indipendente dalla macchina, quindi senza restrizioni.

La **semantica** **attribuisce un significato ad ogni frase sintatticamente corretta**. Con **significato** **si intendono le entità autonome che esistono indipendentemente dai segni che vengono utilizzati per descriverle**.

Esiste un **legame forte tra sintassi e semantica**:

- La **sintassi** è utilizzato come metodo finito per rappresentare un insieme infinito di programmi, la cui sola cosa analizzabile è la struttura;
- La **semantica** è utilizzata come metodo finito, infatti segue la struttura della sintassi, per dare significato a tutti gli elementi dell'insieme infinito dei programmi.

2.7 Induzione matematica e strutturale

La semantica segue la struttura della sintassi, mentre quest'ultima è definita descrivendo gli elementi base e componendo questi elementi, attraverso regole, in elementi composti. Questa forma di definizione si chiama **induzione** e più formalmente è: data un insieme A ed una relazione binaria $<\subseteq A \times A$ ben fondata (senza catene discendenti infinite), se $A = Nat$ si ha **induzione matematica**; se $A = L(G)$ è un linguaggio generato da una grammatica G, allora si ha **induzione strutturale**.

Il **principio di induzione strutturale** si basa sulla seguente definizione. Per dimostrare che una proprietà è valida per tutti gli elementi di una categoria sintattica:

1. **Base induttiva**: si dimostra la proprietà per tutti gli elementi base della categoria, quelli che non hanno nessun elemento come componente;
2. **Passo induttivo**: si dimostra la proprietà per tutti gli elementi composti assumendo che la proprietà sia verificata da tutti i loro componenti immediati.

Ecco un esempio.

Dimostrazione induttiva. Dimostrare che:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \text{con } n \geq 1$$

La **base induttiva** è con n pari ad 1, quindi sostituendo:

$$n = 1 \qquad \sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$$

Per il **passo induttivo** si suppone che sia vero per n considerando dunque come passo induttivo l'equazione iniziale. Si dimostra per $n+1$:

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

$$\sum_{i=1}^{n+1} i \triangleq \underbrace{\sum_{i=1}^{n} i + (n+1)}_{\text{per definizione}} = \underbrace{\frac{n(n+1)}{2}}_{\text{ip. induttiva}} + n + 1$$

$$= \frac{n(n+1) + 2n + 2}{2} = \frac{n^2 + n + 2n + 2}{2}$$

$$= \frac{(n+1)n + (n+1)2}{2} = \frac{(n+1)(n+2)}{2}$$

QED

2.8 Un significato, tante rappresentazioni

Durante l'implementazione di un algoritmo, ci sono alcuni aspetti che un programmatore deve tenere in considerazione:

- Il **comportamento dell'I/O** è un aspetto che interessa l'**implementatore**, ovvero colui che descrive funzionalità attraverso le trasformazioni di stato della macchina.
- La funzione descritta dall'algoritmo è un aspetto che interessa il **progettista**, ovvero colui che progetta costrutti del linguaggio per consentire l'implementazione di certe funzionalità.
- Le **proprietà e invarianti** sono di interesse dello sviluppatore, ovvero colui che è focalizzato sull'utilizzo e sulla combinazione dei costrutti così di preservare invarianti o da garantire proprietà desiderate.

Di fatto tutte queste rappresentazioni, e i loro punti di vista, sono equivalenti, guardano solo il problema da diversi punti. Per questo motivo, nascono **diversi tipi di semantica**:

- **Semantica denotazionale**: descrive funzionalità. Studia gli effetti dell'esecuzione e cerca proprietà del programma studiando proprietà della funzione calcolata;
- **Semantica assiomatica**: descrive proprietà. Necessaria per fare deduzioni logiche, a partire da assiomi dati, su parti del programma (dimostrazioni di correttezza);
- **Semantica operazionale**: descrive trasformazioni di stato. Opera con l'obbiettivo di analizzare il processo per arrivare ad un risultato finale. In parole povere, ha l'obbiettivo di analizzare come i risultati finali vengono prodotti (implementazione di un interprete).

2.8.1 Semantica denotazionale

La **semantica denotazionale** è un modello matematico dei programmi basato sulla ricorsione. È la semantica più "astratta" con cui descrivere i programmi.

Il **processo di costruzione** della semantica denotazionale **per un linguaggio** si articola in due passaggi fondamentali:

1. Definizione di un oggetto matematico per ogni entità del linguaggio;
2. Definizione di una funzione che esegue un *mapping* delle istanze delle entità del linguaggio in istanze dei corrispondenti oggetti matematici.

Il modello matematico utilizzato è quello delle funzioni matematiche ricorsive, ovvero un programma corrispondente ad una funzione tra stati della macchina:

$$E : \text{Prog} \longrightarrow ((\text{Var} \to \text{Val}) \longrightarrow (\text{Var} \to \text{Val}))$$

L'equivalenza di programma si dimostra mediante equivalenza tra funzioni matematiche.

Dunque la **semantica denotazionale è un oggetto puramente matematico** che descrive gli effetti semplicemente manipolando oggetti matematici.
In particolare, vengono **analizzati gli effetti dell'esecuzione del programma**, descrivendo l'effetto dell'esecuzione di una sequenza di comandi separati da ";" attraverso la composizione degli effetti dei singoli comandi da sinistra verso destra.
Con **effetto di ogni comando** si intende la funzione che dato uno stato produce un nuovo stato.

Il suo utilizzo è **utile** per:

- Dimostrare la correttezza dei programmi;
- Ragionare formalmente sui programmi;
- Aiutare la progettazione dei linguaggi;
- Generare i compilatori.s

Purtroppo, a causa della sua elevata complessità, risulta **poco utile** per gli utilizzatori dei linguaggi.

Per **esempio**, si definisce la funzione di valutazione $E[\![P]\!]\sigma$ che valuta il programma P sulla memoria σ, restituendo in output σ' che corrisponde alla memoria σ modificata dal programma P.

Il programma P è formato nel seguente modo:

```
P:
  z := 2;
  y := z;
  y := y+1;
  z := y;
```

La semantica denotazionale consente questa rappresentazione:

$$
\begin{aligned}
E[\![P]\!](z=\bot, y=\bot) &= (E[\![z := y]\!] \circ E[\![y := y+1]\!] \circ E[\![y := z]\!] \circ E[\![z := 2]\!])\,[z=\bot, y=\bot] \\
&= (E[\![z := y]\!]\,(E[\![y := y+1]\!]\,(E[\![y := z]\!]\,(E[\![z := 2]\!])\,[z=\bot, y=\bot]))) \\
&= (E[\![z := y]\!]\,(E[\![y := y+1]\!]\,(E[\![y := z]\!]\,[z=2, y=\bot]))) \\
&= (E[\![z := y]\!]\,(E[\![y := y+1]\!]\,[z=2, y=2])) \\
&= (E[\![z := y]\!]\,[z=2, y=3]) \\
&= [z=3, y=3]
\end{aligned}
$$

Ad ogni passo viene consumata una valutazione, la quale modifica in memoria i valori tra parentesi quadre.

2.8.2 Semantica assiomatica

La **semantica assiomatica** è un modello matematico dei programmi basato sulla logica formale, ovvero il calcolo dei predicati.

L'**obbiettivo** della semantica assiomatica è la verifica formale di programmi. Per farle, vengono utilizzati assiomi e regole di inferenza. Le **espressioni logiche** utilizzate sono chiamate **asserzioni**:

- **Precondizione**: asserzione prima di un comando che dichiara le relazioni e i vincoli validi prima dell'esecuzione del comando;
- **Postcondizione**: asserzione che segue il comando che descrive cosa vale dopo l'esecuzione;
- ***Weakest precondition***: una precondizione meno restrittiva che garantisce la post-condizione.

Quindi, la semantica assiomatica consente di **dimostrare proprietà parziali di correttezza**: quando lo stato iniziale rispetta la precondizione e il programma termina, allora lo stato finale soddisfa la postcondizione.

Dato che la semantica assiomatica è un sistema logico, deve **godere di due proprietà**:

- **Correttezza** (*soundness*), ogni proprietà derivabile nel sistema vale per il programma;
- **Completezza**, ogni proprietà che vale per il programma è derivabile nel sistema di regole.

Per **esempio**, si definisce la notazione $\{P\}$ *statement* $\{Q\}$. Questa semantica si calcola mediante un sistema di prova: si ha una tripla da verificare e si cerca di dimostrarla applicando le regole.

Il programma P è formato nel seguente modo:

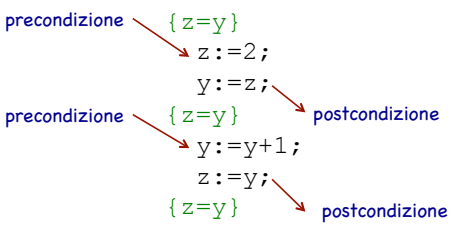
```
P:
z := 2;
y := z;
y := y+1;
z := y;
```

$$\frac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1; S2\ \{P3\}}$$

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S1\ \texttt{else}\ S2\ \{Q\}}$$

$$\frac{(I \text{ and } B)\ S\ \{I\}}{\{I\}\ \text{while } B \text{ do } S\ \{I \text{ and } (\text{not } B)\}}$$



2.8.3 Semantica operazionale

La **semantica operazionale** è un modello matematico dei programmi basato sui sistemi di transizione.

La semantica operazionale descrive il significato del programma **eseguendo i suoi comandi su una macchina**, simulata o reale. La **trasformazione dello stato definisce il significato del comando**. Quindi, lo stato è la rappresentazione astratta della macchina.

Inoltre, la semantica operazionale è necessaria per definire una macchina astratta.

Il modello matematico utilizzato è quello dei sistemi di transizione. Per **esempio**, si consideri la funzione memoria $\sigma : \text{Var} \to \text{Val}$ che associa valori alle variabili. Con **stato** si identifica una coppia nel programma ancora da eseguire:

$$\text{Stato} = \langle P, \sigma \rangle$$

Ad ogni passo, viene eseguita un'operazione e la configurazione o stato cambia. La chiusura transitiva descrive l'esecuzione completa.
Inoltre, questa semantica esegue i comandi separai da ";" sequenzialmente e nell'ordine in cui compaiono da sinistra a destra.

Infine, nonostante sia una delle forme più concrete di semantica, viene eseguita comunque un'astrazione di come il programma viene eseguito, quindi essa è indipendente dall'architettura.

Per **esempio**, il programma P è formato nel seguente modo:

```
1    P:
2    z := 2;
3    y := z;
4    y := y+1;
5    z := y;
```

$$\begin{gathered}\langle z \coloneqq 2; \quad y \coloneqq z; \quad y \coloneqq y+1; \quad z \coloneqq y, [z = \bot, y = \bot]\rangle \\ \langle y \coloneqq z; \quad y \coloneqq y+1; \quad z \coloneqq y, [z = 2, y = \bot]\rangle \\ \langle y \coloneqq y+1; \quad z \coloneqq y, [z = 2, y = 2]\rangle \\ \langle z \coloneqq y, [z = 2, y = 3]\rangle \\ \langle \varepsilon, [z = 3, y = 3]\rangle\end{gathered}$$

2.8.4 Composizionalità

La **composizionalità** è la proprietà per cui il significato di ogni programma deve essere in funzione del significato dei costituenti immediati.

La composizionalità è una **proprietà della semantica** necessaria per caratterizzare i comportamenti e significati di sistemi che possono avere infiniti elementi.

L'**importanza** della composizionalità è dovuta alla necessità di analizzare le proprietà di un programma. Infatti, per farlo è necessario capire che cosa fa e dunque capire la semantica in ogni sua forma. L'analisi diventa molto più semplice grazie alla **modularità**, la quale è **garantita dalla composizionalità**. Quindi, anche nei software di grandi dimensioni, è possibile analizzare il codice separatamente nei suoi moduli, per poi ricomporre il risultato dell'analisi componendo i risultati ottenuti sui singoli moduli.

2.8.5 Equivalenza

L'**equivalenza** è vera quando due programmi hanno la stessa semantica.

Per capire se due programmi sono equivalenti, viene osservata la relazione tra input e output. Se la relazione è la stessa, indipendentemente da come l'algoritmo la calcola, allora sono equivalenti. Quindi, solo caratterizzando la funzionalità I/O è possibile determinare questa caratteristica.

Infine, l'equivalenza è necessaria in varie fasi di analisi:

- **Correttezza**, per dimostrare che il programma scritto calcola esattamente la funzione attesa;
- **Equivalenza di programmi**, per dimostrare che due programmi calcolano la stessa funzione;
- **Efficienza**, Dati due programmi che calcolano la stessa funzione, si vuole dimostrare quale lo fa in modo più efficiente.

2.9 Sintassi

2.9.1 Stato (ambiente e memoria)

Per descrivere il significato dei programmi, è necessario introdurre due entità fondamentali: ambiente e memoria.

L'**ambiente** (***environment***) **è un insieme di legami (*bindings*) tra identificatori e denotazioni**. Inoltre, l'ambiente specifica quali nomi sono usati e per quali oggetti, solitamente possono essere legati ad un tipo, ad un valore, ad una locazione. Quindi, i *binding*[1] sono associati ai nomi, i quali possono essere variabili, costanti, procedure, e altro. I nomi sono solo un'entità separata dall'oggetto che denotano, infatti esso potrebbe essere usato in contesti diversi per rappresentare valori differenti.

La **memoria** (***store***) **è un insieme di effetti sugli identificatori (causati da assegnamenti)**. Essa è una mappa che solitamente rappresenta la storia, cioè l'evoluzione, delle variazioni dei valori associati agli identificatori. In altre parole, fornisce un *binding* tra locazione e valore.
Si ricorda, che la memoria è fortemente legato all'approccio imperativo, infatti i linguaggi funzionali pure non ne hanno bisogno dato che non gestiscono variabili che cambiano durante l'esecuzione del programma.

2.9.2 Categorie sintattiche

Le **categorie sintattiche** **sono la classificazione dei costrutti in funzione del loro significato atteso, ovvero della classe di effetti che ha la loro esecuzione causa.**
Quindi, esse sono classi che rappresentano un diverso tipo di significato, effetto, esecuzione di un programma.

Le categorie sintattiche si distinguono in: **espressioni**, **comandi** e **dichiarazioni**. Formalmente, le categorie sono i simboli non terminali della grammatica classificati in funzione di cosa modificano dello stato e come lo modificano.

[1] In breve sarebbero le API (*Application Programming Interface*): link fonte.

2.9.3 Espressioni

Le **espressioni** **nascono dalla necessità di avere una categoria sintattica che consenta di rappresentare e denotare i valori**. Esse possono essere espresse con dei vincoli, per esempio il tipo.

Nonostante le espressioni denotino i valori, esse **non sono locazioni di memoria**. Quest'ultime sono legati all'architettura (struttura) di una macchina, mentre le espressioni fanno riferimento allo specifico linguaggio di programmazione.

Due espressioni sono considerate **equivalenti** **se vengono valutate nello stesso valore in tutti gli stati di computazione. Anche eventuali *side-effect* devono essere gli stessi**.

Infatti, due espressioni possono essere diverse ma essere valutate nello stesso valore. Per **esempio**, l'espressione logica `not(a and b)` è logicamente (semanticamente) equivalente ad `(not a) or (not b)` ma sono sintatticamente diverse.

2.9.4 Dichiarazioni

Le **dichiarazioni** **sono la categoria sintattica che consente la creazione o la modifica dei legami associati agli identificatori, ovvero gli ambienti**.

Esse nascono con l'obbiettivo di utilizzare i valori. Per farlo, è stato necessario creare, utilizzare e modificare **legami** tra nomi e valori.

Inoltre, gli le modifiche agli ambienti sono trasformazioni **reversibili**, ovvero che le trasformazioni hanno valenza esclusivamente all'interno del raggio d'azione (*scope*) attuale dell'identificatore. In altre parole, i linguaggi di programmazione delimitano la validità di un ambiente. Più precisamente con **reversibili si intende** che una volta terminata la validità di un ambiente, le modifiche verranno annullate.

Due dichiarazioni sono **equivalenti** **se producono lo stesso ambiente e la stessa memoria in caso di *side-effect* in tutti gli stati di computazione**. A causa del *side-effect* è necessario che l'equivalenza richieda che due dichiarazioni generino le stesse identiche modifiche a tutto lo stato di computazione.

2.9.5 Comandi

I **comandi** **sono richieste di modifica dello stato di computazione, ed in particolare della memoria**.

Le trasformazioni sono **irreversibili**, ovvero sono definite nell'esecuzione del programma. Quindi, per annullarle è necessario eseguire altri comandi e altre trasformazioni che consentono di tornare alla stessa memoria iniziale. Quindi, i comandi non sono altro che funzioni di trasformazione e devono essere **eseguiti** per poter attuare la corrispondente trasformazione (irreversibile) della memoria.

Due comandi sono **equivalenti** **se per ogni stato (memoria) in input producono lo stesso stato (memoria) in output**.

2.10 Semantica operazionale: sistemi di transizione

I **sistemi di transizione** **sono strumenti astratti mirati a specificare senza ambiguità e senza dipendenza dalla macchina, cosa fa un linguaggio**. Le sue caratteristiche principali sono:

- Matematicamente precisi;
- Molto concisi;
- Metodo di specifica generale che consente l'astrazione;
- Specificano cosa viene calcolato per induzione sulla struttura sintattica del linguaggio.

Inoltre, sfruttando la definizione induttiva del linguaggio, consentono di definire il significato mediante induzione sull'*abstract syntax tree*. Quindi, viene **usata l'induzione strutturale matematica per ragion sui programmi**.

Formalmente: **un sistema di transizione è una struttura** $(\Gamma, \rightarrow)$**, dove** Γ **è un insieme di elementi** γ **chiamati configurazioni e la relazione binaria** $\rightarrow \subseteq \Gamma \times \Gamma$ **è chiamata relazione di transizione. Se** $\Gamma_T \subseteq \Gamma$ **è un insieme di configurazioni terminali, il sistema è detto terminale.**