

Linguaggi

VR443470

marzo 2023

Indice

1 Introduzione	4
1.1 Nascita dei linguaggi	4
1.2 Definizioni	5
1.2.1 Algoritmo	5
1.2.2 Dati	5
1.2.3 Sintassi e semantica	5
1.2.4 Linguaggio matematico e logico	6
1.2.5 Linguaggio di programmazione	6
1.2.6 Programma	6
1.3 Aspetti di progettazione	7
1.3.1 Leggibilità	7
1.3.2 Scrivibilità	7
1.3.3 Affidabilità e costo	8
1.4 Classificazione dei linguaggi	9
1.4.1 Metodo di computazione	9
1.4.2 Per caratteristiche	9
1.5 Implementazione dei linguaggi	10
1.5.1 Macchina astratta	11
1.5.2 Realizzazione di una macchina astratta a vari livelli	12
1.5.3 Realizzazione software e livelli di astrazione	13
1.6 Realizzazione di una macchina a livello software/firmware	14
1.6.1 Soluzione interpretativa: interprete	15
1.6.2 Soluzione interpretativa: operazioni e struttura	16
1.6.3 Soluzione interpretativa: pro e contro	18
1.6.4 Soluzione compilativa: compilatore	19
1.6.5 Soluzione compilativa: struttura	21
1.6.6 Soluzione compilativa: pro e contro	22
1.6.7 Soluzione reale: ibrido	23
1.7 Sintesi	24
2 Descrivere i linguaggi	25
2.1 Sintassi	26
2.1.1 Definizione e notazione	26
2.1.2 Descrivere la sintassi	27
2.2 Grammatiche <i>context-free</i>	28
2.3 Notazione BNF	29
2.4 Descrivere un semplice linguaggio imperativo	30
2.5 Analisi semantica	31
2.6 Semantica dinamica	32
2.7 Induzione matematica e strutturale	33
2.8 Un significato, tante rappresentazioni	35
2.8.1 Semantica denotazionale	36
2.8.2 Semantica assiomatica	38
2.8.3 Semantica operazionale	40
2.8.4 Composizionalità	41
2.8.5 Equivalenza	41
2.9 Sintassi	42
2.9.1 Stato (ambiente e memoria)	42

2.9.2	Categorie sintattiche	42
2.9.3	Espressioni	43
2.9.4	Dichiarazioni	43
2.9.5	Comandi	44
2.10	Semantica operazionale: sistemi di transizione	44
2.11	Esempio di un linguaggio reale (PL_0)	45
3	Espressioni	46
3.1	Introduzioni	46
3.2	Descrivere le espressioni	46
3.3	Caratteristiche	47
3.3.1	Notazione	47
3.3.2	Notazione post-fissa	48
3.3.3	Notazione pre-fissa	51
3.3.4	Notazione in-fissa	54
3.4	Valutazione delle espressioni	55
3.5	Semantica delle espressioni	56
3.6	Regole di transizione	57
3.7	Valutazione ed equivalenza	60
4	Dichiarazioni	61
4.1	Identificatori	61
4.2	Bindings	62
4.2.1	Legami	62
4.2.2	Scope	63
4.2.3	Bindings nei linguaggi di programmazione	64
4.2.4	Tipi di bindings nei linguaggi di programmazione	65
4.2.5	Tempi	65
4.3	Semantica: Identificatori, ambienti e dichiarazioni	65
4.3.1	Termini chiusi e ground	65
4.3.2	Ambienti nei linguaggi imperativi	65
4.3.3	Espressioni con identificatori	65
4.3.4	Identificatori liberi	65
4.4	Nuove regole	65
4.5	Tipo	65
4.5.1	Binding di tipo (<i>Type binding</i>)	65
4.6	Ambiente statico e semantica statica	65
4.6.1	Semantica statica delle espressioni	65
4.7	Compatibilità di ambienti	65
4.8	Espressioni con identificatori costanti - Regole	65

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.

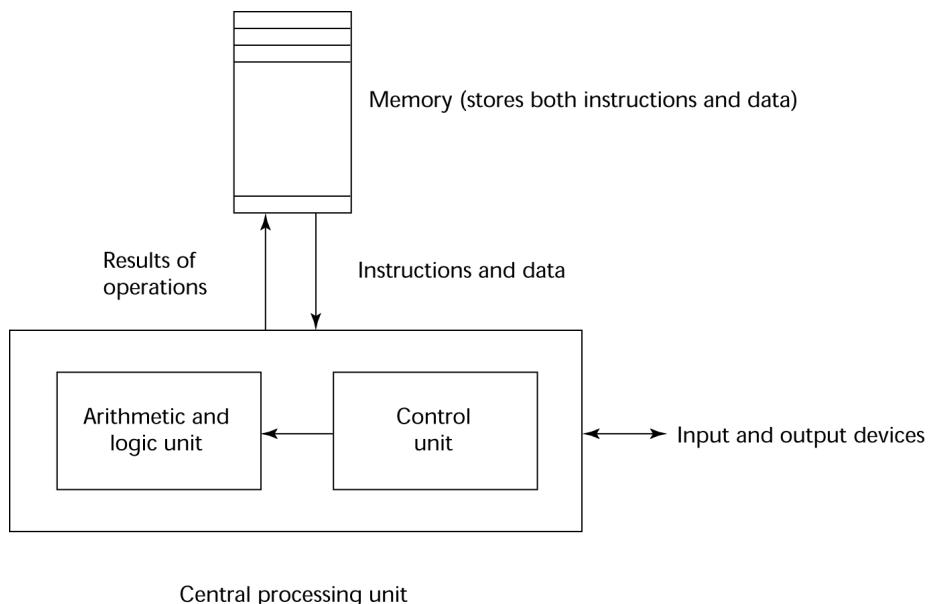


Figura 1: Architettura di Von Neumann.

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

Definizione 1

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 scomponete 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

Definizione 2

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

Definizione 3

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

Definizione 4

La **scrivibilità** (*writability*) è la facilità di utilizzo di un linguaggio per creare programmi, la facilità di analisi e la verifica dei programmi.

I fattori che influenzano la leggibilità sono gli stessi della scrivibilità.

In breve i fattori 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

Definizione 5

Per **affidabilità** (*reliability*) si intende la conformità alle sue specifiche.

Definizione 6

Per **costo** si intende 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 eccezioni**. 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 soluzione 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.

Definizione 7

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.

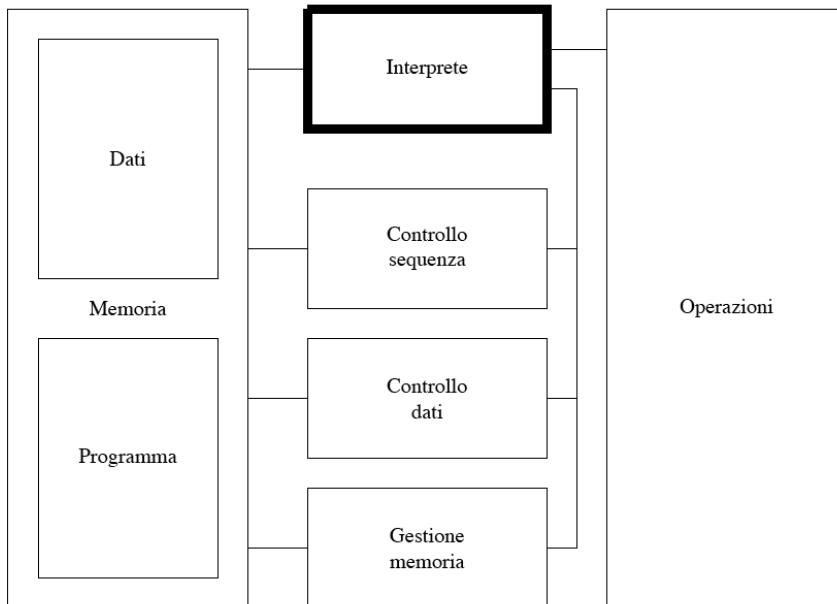


Figura 3: Rappresentazione di una macchina astratta.

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.

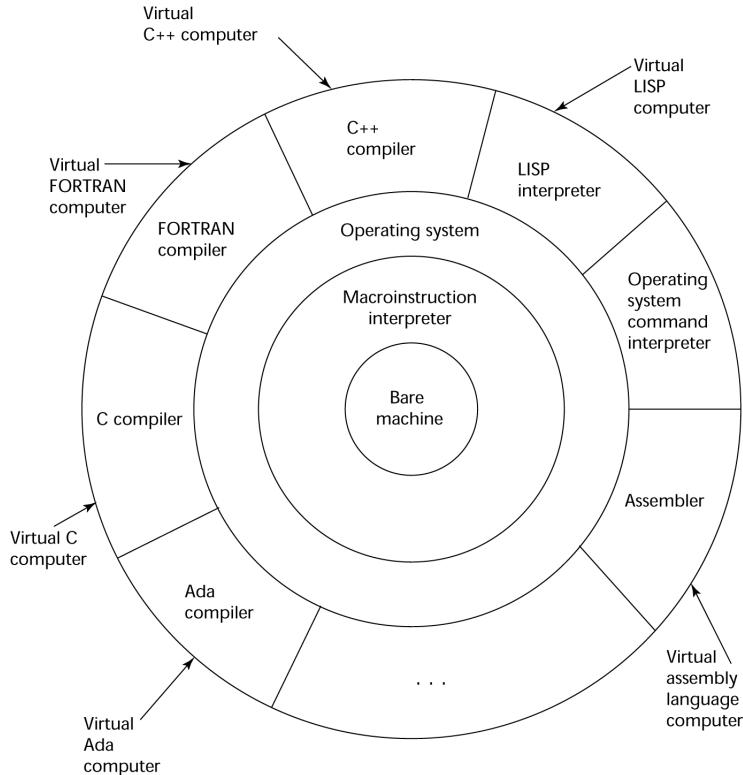
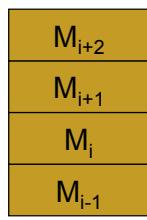


Figura 4: Livelli di astrazione utilizzati dai linguaggi ad alto livello.

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 $L_{M_{i-1}}$)
 - per fornire servizi a M_{i+1} (interpretare $L_{M_{i+1}}$)
 - 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 $\text{int}^{L_0, L}$ 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.

Definizione 8

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: $P^L \in \text{Prog}^L$ e $\text{in}, \text{out} \in D$;
- $\llbracket P^L \rrbracket : D \longrightarrow D$ 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 $\llbracket P^L \rrbracket (\text{in}) = \text{out}$.

Definizione 9

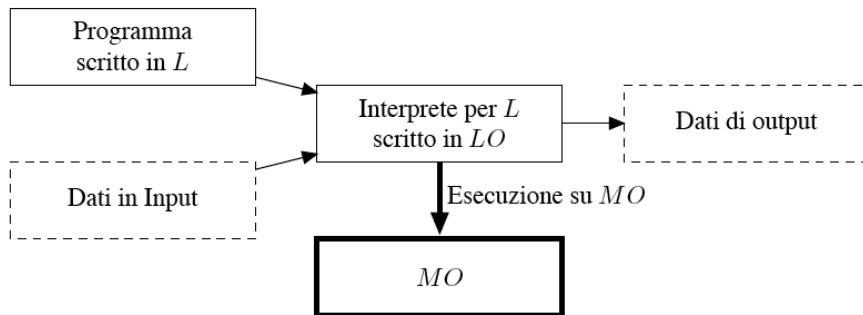
Un **interprete formalmente** è esprimibile nel seguente modo.
Si consideri un interprete da L a L_0 : dato $P^L \in \text{Prog}^L$ e $\text{in} \in D$, un interprete int^{L, L_0} per L su L_0 è un programma tale che:

$$\llbracket \text{int}^{L, L_0} \rrbracket : (\text{Prog}^L) \longrightarrow D$$

e dunque:

$$\llbracket \text{int}^{L, L_0} \rrbracket (P^L, \text{in}) = \llbracket P^L \rrbracket (\text{in})$$

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);
- Riga 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);
- Riga 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).

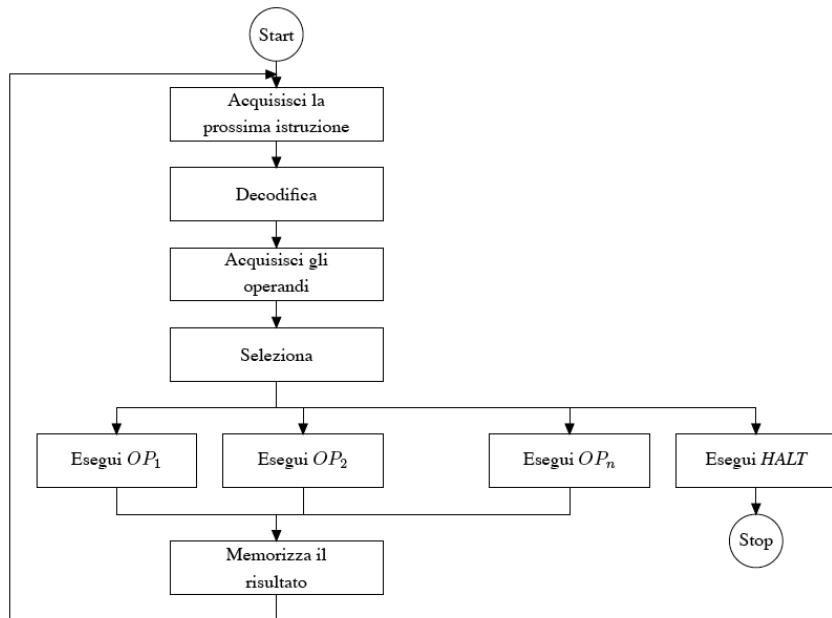


Figura 5: Diagramma a blocchi della struttura di un interprete.

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.

Definizione 10

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: $P^L \in \text{Prog}^L$ e $in, out \in D$;
- $\llbracket P^L \rrbracket : D \longrightarrow D$ rappresenta la semantica di P^L .

Definizione 11

Un **compilatore formalmente** è esprimibile nel seguente modo.
Dato $P^L \in \text{Prog}^L$, un **compilatore** comp^{L, L_0} da L a L_0 è un programma tale che:

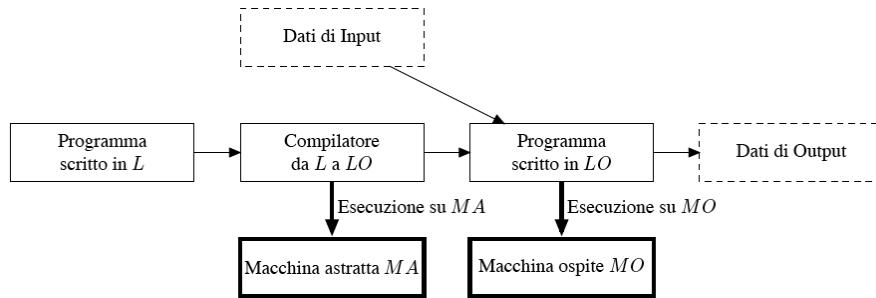
$$\llbracket \text{comp}^{L, L_0} \rrbracket : \text{Progr}^L \longrightarrow \text{Progr}^{L_0}$$

e dunque:

$$\llbracket \text{comp}^{L, L_0} \rrbracket (P^L) = P^{L_0} \text{ tale che } \forall in \in D. \llbracket P^{L_0} \rrbracket (in) = \llbracket P^L \rrbracket (in)$$

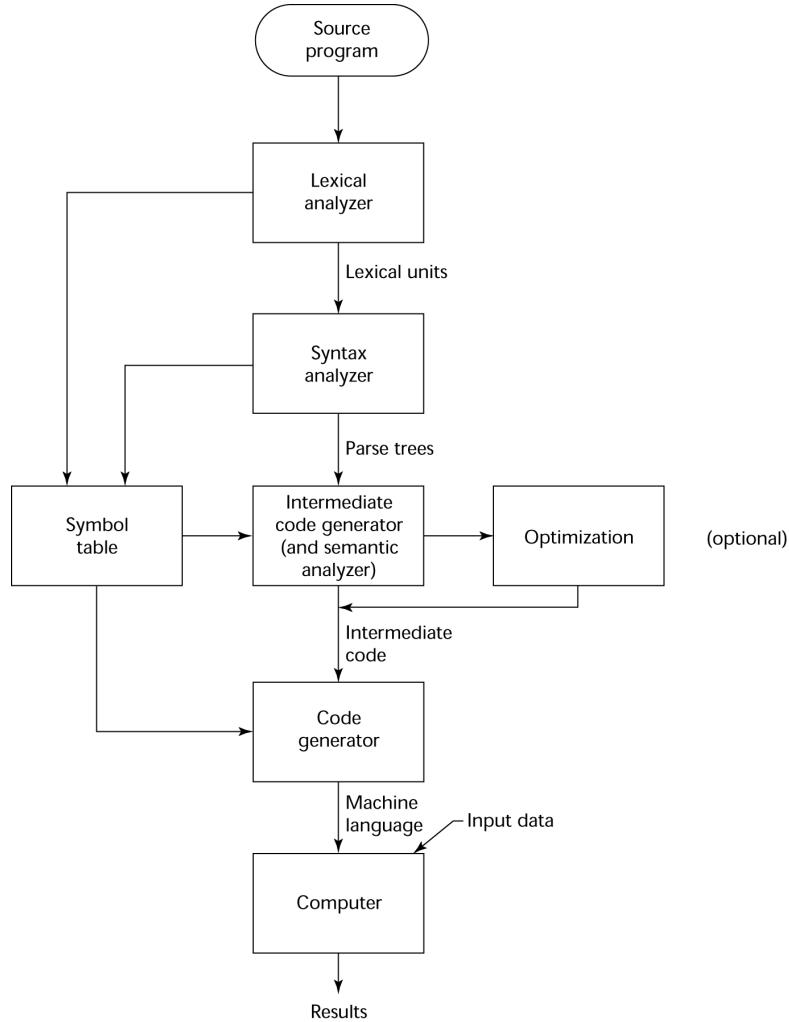
Ovvero che l'esecuzione della compilazione del linguaggio L a L_0 con input il programma scritto in L , l'output sia uguale al programma scritto nel linguaggio L_0 ; tale che per ogni input appartenente all'insieme dei dati, l'esecuzione del programma scritto in L_0 con input in , sia uguale all'esecuzione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

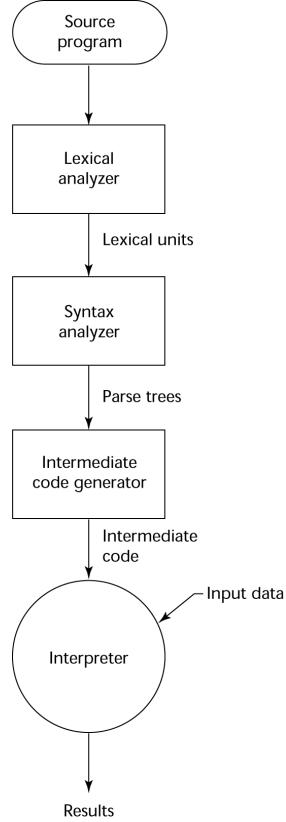
- **Analisi sintattica** (*Syntax analyzer*), crea una rappresentazione ad albero della sintassi del programma. Ogni foglia è un *token* e le foglie lette da sinistra verso destra costituiscono frasi ben formate del linguaggio. Inoltre, l'albero costituisce la struttura logica del programma e dunque 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 presenti nel programma, come gli identificatori, le chiamate di procedura, ecc.
 - **Analisi semantica** (*Semantic analyzer*), consente di rilevare errori semanticici, 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.

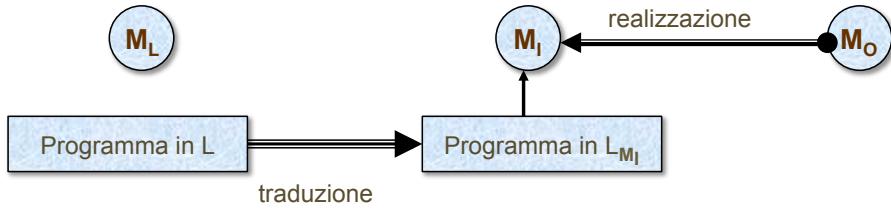


Figura 6: Soluzione ibrida con le *system call*.

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.

Definizione 1

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.

Definizione 2

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 $A \rightarrow \alpha$, 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 $\langle \rangle$, e per le produzioni si utilizza il simbolo $::=$ al posto della freccia.

$$\langle A \rangle ::= \alpha \langle B \rangle \gamma \mid \alpha$$

$$\langle B \rangle ::= \varepsilon \mid \beta_1 \mid \beta_2$$

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:

$$\langle A \rangle ::= \alpha [\beta_1, \beta_2] \gamma \mid \alpha \gamma \mid \gamma$$

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.

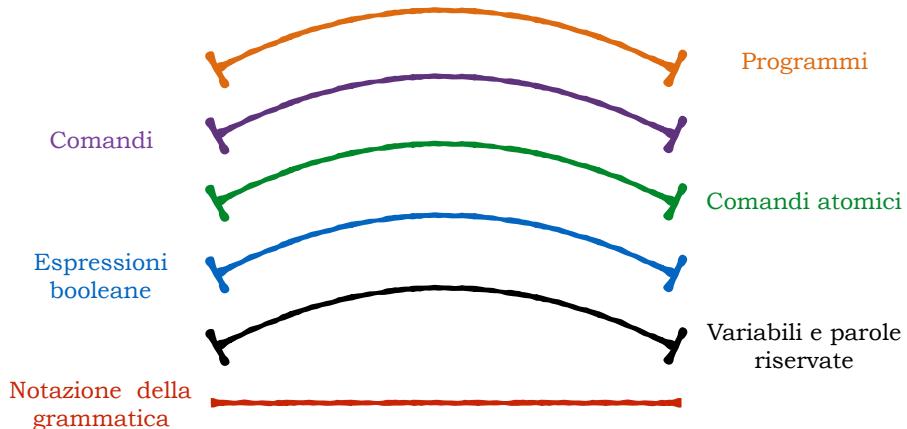


Figura 7: Linguaggio descritto a più livelli di astrazione.

Un esempio di descrizione a più livelli:

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

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.

Definizione 3

La **semantica** attribuisce un significato ad ogni frase sintatticamente corretta.

Definizione 4

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.

Definizione 5

Questa forma di definizione si chiama **induzione** e più formalmente è: data un insieme A ed una relazione binaria $\subset \subseteq A \times A$ 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**.

Definizione 6

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 una dimostrazione d'**esempio** di induzione.

Dimostrazione induttiva. Dimostrare che:

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

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

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

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

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

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 denazionale**: 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

Definizione 7

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:

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

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:

```

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

La semantica denotazionale consente questa rappresentazione:

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

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

2.8.2 Semantica assiomatica

Definizione 8

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

L'**obiettivo** 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 $\{P\} \text{ statement } \{Q\}$. Questa semanticà si calcola mediante un sistema di prova: si ha una tripla da verificare e si cerca di dimostrarla applicando le regole.

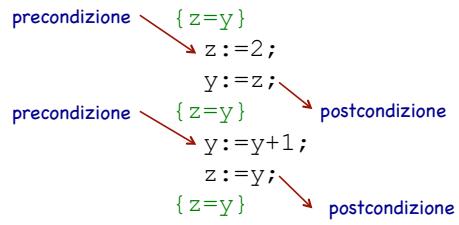
Il programma P è formato nel seguente modo:

```

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

```

$$\begin{array}{c}
 \frac{\{P\} S \{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\} S \{Q'\}} \\[10pt]
 \frac{\{P1\} S1 \{P2\}, \{P2\} S2 \{P3\}}{\{P1\} S1; S2 \{P3\}} \\[10pt]
 \frac{\{B \text{ and } P\} S1 \{Q\}, \{(not B) \text{ and } P\} S2 \{Q\}}{\{P\} \text{ if } B \text{ then } S1 \text{ else } S2 \{Q\}} \\[10pt]
 \frac{}{\{I\} \text{ while } B \text{ do } S \{I \text{ and } (not B)\}}
 \end{array}$$



2.8.3 Semantica operazionale

Definizione 9

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 $\sigma : \text{Var} \rightarrow \text{Val}$ 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 separati 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;
```

$$\begin{aligned}
& \langle z := 2; \quad y := z; \quad y := y + 1; \quad z := y, [z = \perp, y = \perp] \rangle \\
& \langle y := z; \quad y := y + 1; \quad z := y, [z = 2, y = \perp] \rangle \\
& \langle y := y + 1; \quad z := y, [z = 2, y = 2] \rangle \\
& \langle z := y, [z = 2, y = 3] \rangle \\
& \langle \varepsilon, [z = 3, y = 3] \rangle
\end{aligned}$$

2.8.4 Composizionalità

Definizione 10

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

Definizione 11

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.

Definizione 12

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.

Definizione 13

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

Definizione 14

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.

¹In 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.

Definizione 15

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'obiettivo 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.

Definizione 16

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.

Definizione 17

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

2.10 Semantiche operazionali: 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**.

Definizione 18

Formalmente: un **sistema di transizione** è una struttura (Γ, \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**.

2.11 Esempio di un linguaggio reale (PL_0)

Un esempio di linguaggio reale è PL_0 strutturato nel seguente modo:

```

<program>   P → B.
<block>      B → D S
<declar>     D → [const I=N{,I=N};]
              | [var I{,I};]
              | [procedure I;B;]
<stmts>      S → ε | I := E
              | call I
              | if C then S
              | while C do S
              | begin S{;S} end
<Cond>       C → odd E | E > E
              | E >= E | E = E
              | E # E | E < E
              | E <= E
<Expr>       E → I | N | E bop E
              | ( E )
  
```

- I programmi P sono blocchi.
- I blocchi B sono una dichiarazione D seguita da un comando S , entrambi liberi dal contesto.
- Le dichiarazioni D sono dichiarazioni di valori costanti con nome (const), dichiarazioni di variabili (var) e dichiarazioni di procedure.
- I nomi sono gli identificatori I , considerati parte del vocabolario, e N sono i numeri naturali, anch'essi considerati parte del vocabolario.
- Non esiste il concetto di tipo poiché l'unico dato ammesso è intero.
- I comandi sono:
 - Comando vuoto;
 - Assegnamento di un valore ad un identificatore;
 - Chiamata ad una procedura mediante il suo nome;
 - Comando condizionale che esegue un comando al verificarsi di una condizione booleana;
 - Ciclo che ripete un comando finché una data condizione è vera;
 - Composizione sequenziale di comandi.
- Le condizioni C sono espressioni che hanno al loro interno valori booleani.
- Le espressioni E sono identificatori, valori naturali, operazioni tra espressioni e espressioni tra parentesi.

3 Espressioni

3.1 Introduzioni

Per manipolare i valori è necessario rappresentarli e per questo motivo vengono utilizzate le espressioni. Quest'ultime, devono essere valutate per dare un valore e con valore non viene intesa la locazione di memoria: i valori fanno parte dello specifico linguaggio di programmazione, mentre la locazione è parte della struttura dell'architettura. Quindi, la locazione di memoria modella il modo in cui il linguaggio di programmazione si relazione alla macchina sottostante.

3.2 Descrivere le espressioni

Per associare significati e valori alle **espressioni**, esse **devono essere valutate**.

Il **valore ottenuto mediante la valutazione** è chiamato **valore esprimibile**, poiché è un **valore che può essere espresso attraverso la sintassi del linguaggio di programmazione**, e in particolare attraverso la sintassi delle espressioni.

Formalmente con **valore esprimibile** si intende l'unione tra i tipi booleani e interi:

$$Eval = \text{bool} \cup \text{int} \quad \text{con metavariabile } ev$$

Per metavariabile si intende qualsiasi valore esprimibile.

Per **esempio**, un linguaggio come PL_0 (paragrafo 2.11), si hanno espressioni che rappresentano interi e identificatori, e indirettamente anche booleani in quanto le condizioni corrispondono a valori booleani (e.g. $3 = 3$ è *true*, mentre $3 \# 3$ è *false*).

Dal punto di vista **sintattico**, i **costituenti elementari delle espressioni** sono **letterali** (costanti e identificatori), i quali sono **composti mediante operatori**.

Per concludere, un'espressione è un oggetto sintattico composto da operatori, operandi (che sono espressioni alla fine), parentesi e funzioni/procedure.

3.3 Caratteristiche

Ecco qua di seguito gli **aspetti che caratterizzano le espressioni**:

- **Notazione** che specifica in che modo gli operandi e gli operatori vengono rappresentati. Inoltre, indica su quali operandi un operatore opera.
 - Paragrafo 3.3.1 sulla notazione
 - Paragrafo 3.3.2 sulla notazione post-fissa
 - Paragrafo 3.3.3 sulla notazione pre-fissa
 - Paragrafo 3.3.4 sulla notazione in-fissa
 - **Regole di precedenza** tra operatori
 - **Regole di associatività** degli operatori
 - **Ordine di valutazione degli operandi**
 - **Presenza di *side-effects***, ovvero qualunque effetto che non consiste puramente nella rappresentazione di valori;
 - **Overloading degli operatori**, ovvero simboli di operatori che hanno significati diversi a seconda del tipo degli operatori a cui sono applicati.
 - **Espressioni con tipi misti**
-

3.3.1 Notazione

A seconda di **come viene rappresentata l'espressione**, varia anche il modo in cui viene determinata la **semantica** e di conseguenza varia la sua valutazione.

Esistono diversi modi per denotare le espressioni:

- Notazione **in-fissa**, per esempio $a + b$
- Notazione **pre-fissa** (polacca), per esempio $+ab$
- Notazione **post-fissa** (polacca inversa), per esempio $ab+$

La notazione in-fissa è quella più utilizzata nelle espressioni aritmetiche poiché più intuitiva, ma sono necessarie altre regole per eliminare l'ambiguità. Nelle altre notazioni (pre e post fissa), l'**arietà dell'operatore**, ovvero il **numero di operandi a cui si applica**, è spesso sufficiente ad evitare ambiguità nella valutazione.

3.3.2 Notazione post-fissa

La notazione **post-fissa** è più semplice della **in-fissa**, dal punto di vista della **computazione**, poiché:

- Non sono necessarie regole di precedenza
- Non sono necessarie regole di associatività
- Non sono necessarie parentesi

Viene anche concessa una rappresentazione uniforme per qualsiasi arietà di operatori².

Vengono presentati due **esempi**:

- La notazione $ab + cd + *$ viene valutata da sinistra a destra usando una pila LIFO (*Last In, First Out*).
- La notazione $532*+$ viene valutata nel seguente modo: durante la lettura, i valori vengono messi sulla pila; nel momento in cui viene trovato un operatore, come $*$, sapendo che la sua arietà è 2, è noto che l'operatore viene applicato ai primi due valori in cima alla pila. Infine, il risultato dopo ogni operazione viene caricato in cima alla pila;

L'**algoritmo di valutazione** si articola in pochi passaggi. **Graficamente** è possibile leggerlo nella prossima pagina, mentre in forma scritta qui di seguito:

1. Lettura di un simbolo dall'espressione.
 - (a) Se il simbolo letto è un operatore:
 - i. Applicazione dell'operatore a n operandi immediatamente precedente sulla pila, il numero di operandi dipende dall'arietà dell'operatore.
 - ii. Memorizzazione del risultato in R .
 - iii. Eliminazione dell'operatore e degli operandi dalla pila.
 - iv. Memorizzazione del valore R sulla pila.
 - (b) Se il simbolo letto non è un operatore, allora è un operando e viene inserito direttamente in pila.
2. Se ci sono altri simboli da leggere rinizia dal punto 1, altrimenti termina e il risultato sarà in cima alla pila.

²Arietà: numero di operandi a cui si applica.

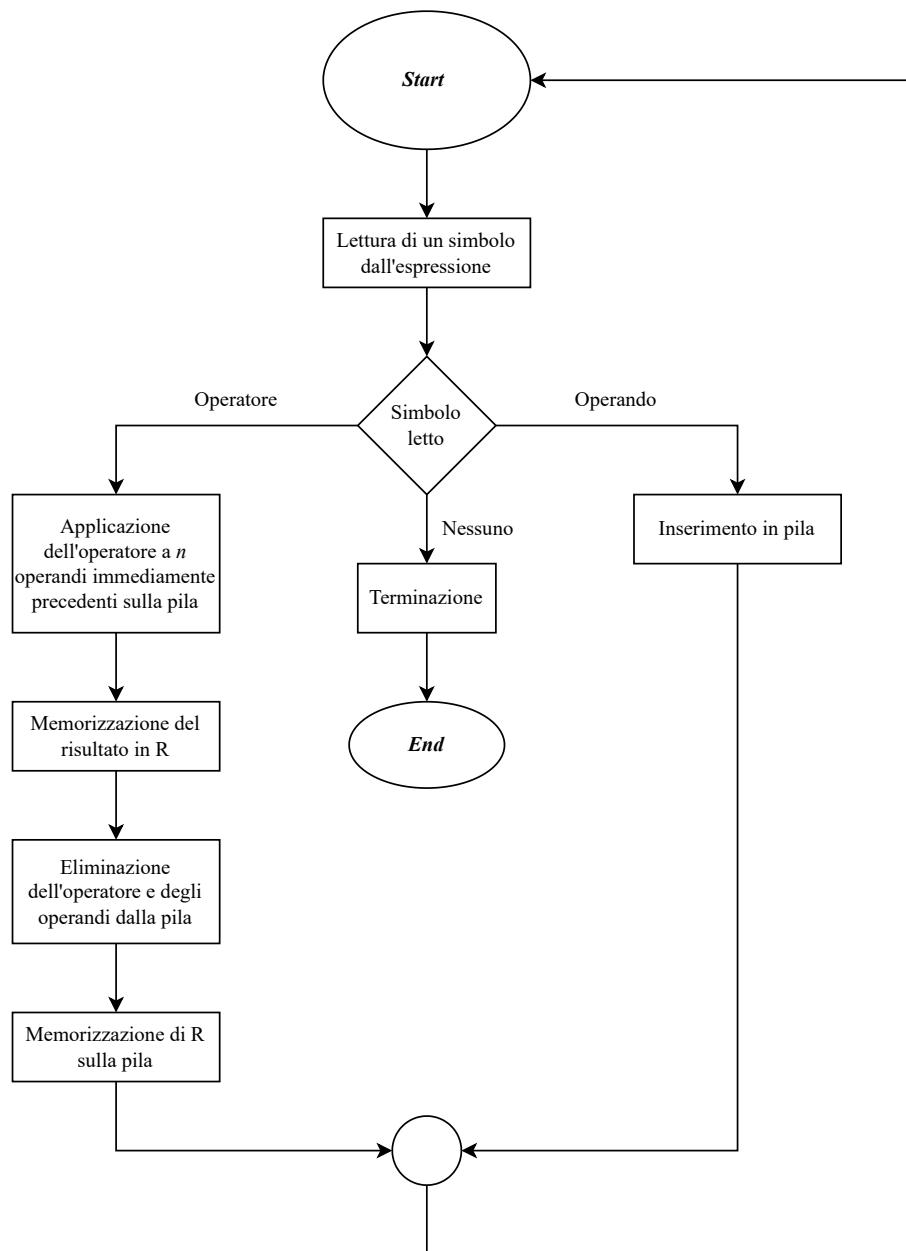
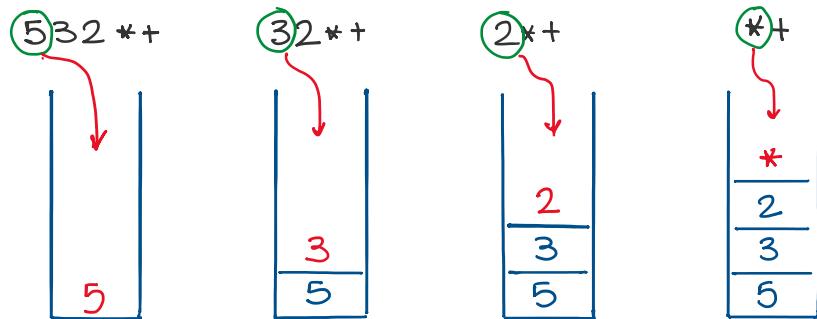


Figura 8: Algoritmo di valutazione nella notazione post-fissa.



OPERANDO IN CIMA ARIETA' = 2



Figura 9: Esempio di applicazione dell'algoritmo di valutazione della notazione post-fissa.

3.3.3 Notazione pre-fissa

La notazione **pre-fissa** è più semplice della **in-fissa**, dal punto di vista della **computazione**, poiché:

- Non sono necessarie regole di precedenza
- Non sono necessarie regole di associatività
- Non sono necessarie parentesi

Viene anche concessa una rappresentazione uniforme per qualsiasi arietà di operatori³.

A differenza della notazione post-fissa, l'algoritmo di valutazione è più complicato poiché è necessario contare gli operandi che vengono letti.

Per **esempio**, la notazione $* + ab + cd$ viene valutata da sinistra a destra usando una pila LIFO (*Last In, First Out*). A differenza delle altre notazioni, in questo caso ogni volta che viene letto un operatore, è necessario contare gli operandi da leggere prima di eseguire il calcolo.

L'**algoritmo di valutazione** si articola in pochi passaggi. **Graficamente** è possibile leggerlo nella prossima pagina, mentre in forma scritta qui di seguito:

1. Lettura di un simbolo dall'espressione e inserimento in pila.
 - (a) Se il simbolo letto è un operatore, inizializza il contatore C con il numero di argomenti dell'operatore ($C_{op} = n$) e torna al punto 1;
 - (b) Se il simbolo letto è un operando, decrementa il contatore C di uno ($C_{op} = C_{op} - 1$) (op è l'ultimo operatore inserito):
 - i. Se $C_{op} \neq 0$, torna al punto 1;
 - ii. Se $C_{op} = 0$ vengono eseguite le seguenti operazioni:
 - A. Applicazione dell'ultimo operatore agli operandi in cima alla pila;
memorizzazione del risultato in R ;
eliminazione dell'operatore e degli operandi processati dalla pila;
memorizzazione del risultato R sulla pila;
 - B. Se non vi sono più simboli di operatore sulla pila, si salta al punto 2 (la fine);
 - C. Inizializzazione del nuovo contatore: $C_{op} = n - m$ dove n è l'arietà del nuovo operatore nella pila, il quale si trova in cima, e m è il numero di operandi presenti nella pila sopra tale operatore;
 - D. Ritorna al punto i.
2. Se la sequenza da leggere non è vuota torna all'1.

³Arietà: numero di operandi a cui si applica.

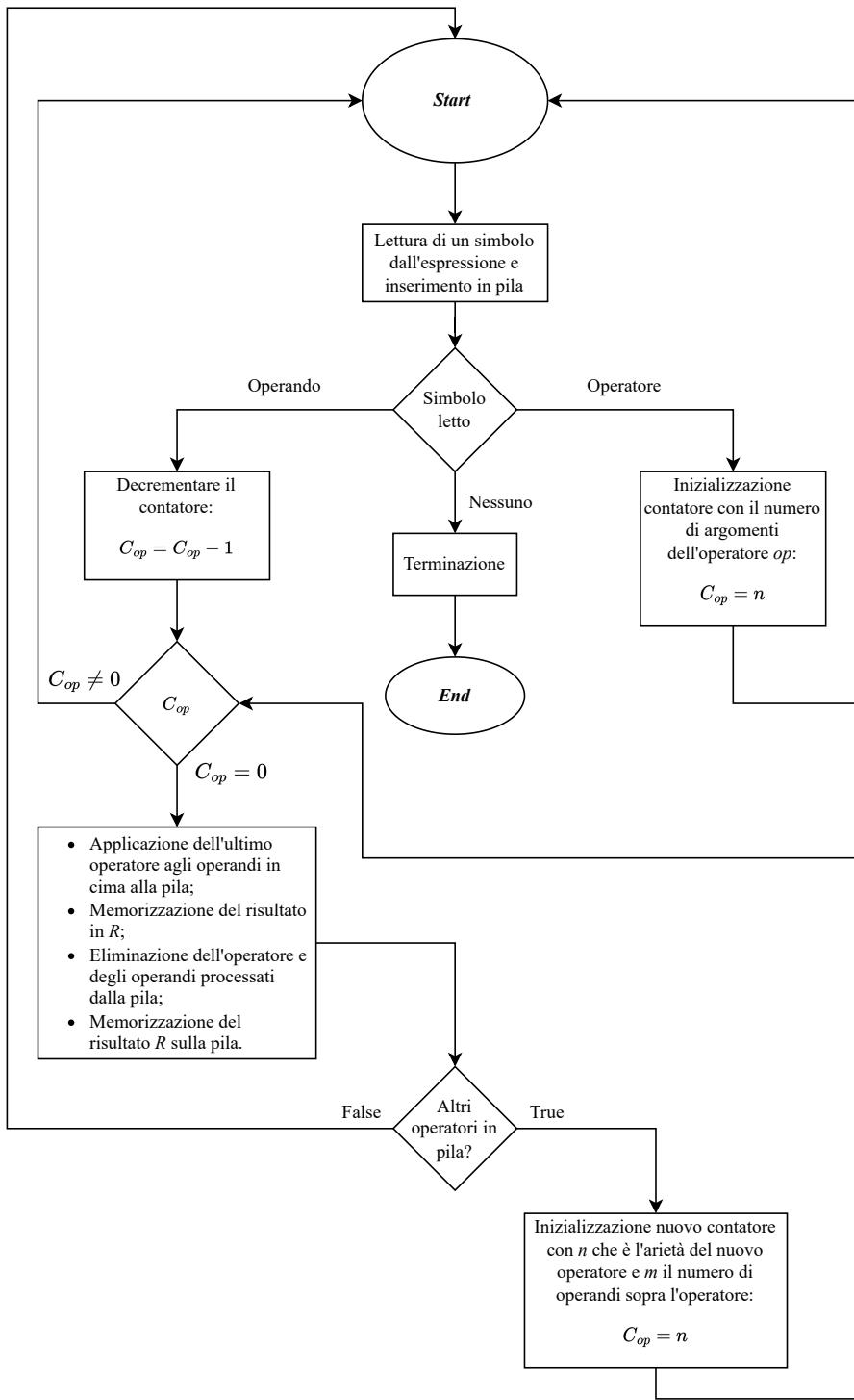
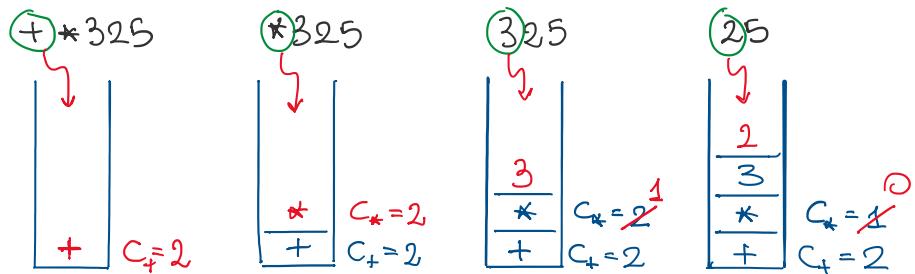


Figura 10: Algoritmo di valutazione nella notazione post-fissa.



$C = \emptyset$ DELL' OPERATORE IN CIMA \rightarrow ESEGNO

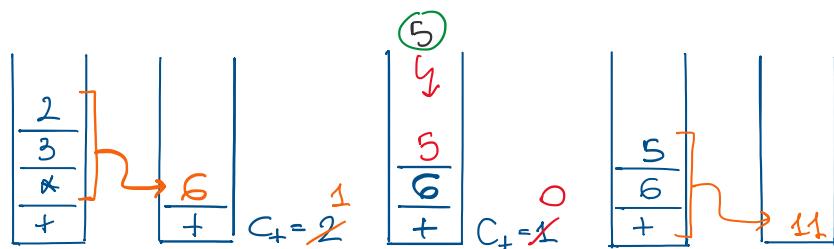


Figura 11: Esempio di applicazione dell'algoritmo di valutazione della notazione pre-fissa.

3.3.4 Notazione in-fissa

La notazione **in-fissa** è la **notazione più utilizzata in matematica** e nei linguaggi di programmazione come zucchero sintattico. Inoltre, è quella che consente una lettura più intuitiva ma che richiede maggiori specifiche.

Per **esempio**, considerando $15 - 5 - 3$, a seconda dell'operazione che viene eseguita per prima, cambia il risultato.

L'algoritmo di valutazione della notazione in-fissa non è così semplice come i precedenti. Infatti, a causa di alcune regole, chiamate di **precedenza** e **associatività**, e alla presenza di alcune regole, non è possibile valutare l'espressione con una semplice scansione da sinistra a destra o viceversa.

Definizione 1

Con **regole di precedenza** si intende: **le regole di precedenza di un operatore per la valutazione di un'espressione, definisce l'ordine in cui operatori "adiacenti" a diversi livelli di precedenza vengono valutati.**

Anche se ogni linguaggio differisce per le priorità assegnate agli operatori, l'importante è che vengano stabilite a priori. Per **esempio**, un tipico livello di precedenza è dato da: (1) parentesi, (2) operatori unari, (3) elevazione a potenza, (4) moltiplicazione e divisione, (5) somma e sottrazione.

Definizione 2

Con **regole di associatività** si intende: **le regole di associatività per la valutazione delle espressioni definiscono l'ordine con cui operatori allo stesso livello di precedenza vengono valutati.**

Per **esempio**, delle tipiche regole di associatività sono: da sinistra verso destra; a volte operatori unitari associano da destra; in alcuni linguaggi tutti gli operatori hanno lo stesso livello di precedenza e associano da destra; le regole di precedenza e associatività possono essere sovrascritte usando le parentesi.

Un altro **esempio** $a + b - c + d$ restituisce valori diversi a seconda che si associa da destra o da sinistra.

3.4 Valutazione delle espressioni

La rappresentazione interna delle espressioni consiste in una **rappresentazione ad albero** (*abstract syntax tree*), dove i **nodi interni** sono **operatori** mentre le foglie sono valori (operandi elementari). Ogni nodo interno ha come figli gli alberi che rappresentano le espressioni a cui l'operatore si applica.

In altre parole, l'albero **fornisce la struttura dell'espressione**, ovvero:

- **Aspetto positivo:** elimina eventuali ambiguità legate a precedenze e associatività;
- **Aspetto negativo:** non dà informazioni riguardanti l'ordine di valutazione dell'espressione.

A causa delle eventuali ambiguità sulle espressioni, nasce il **problema dell'ordine di valutazione**, il quale è prettamente informatico. Infatti, la presenza di alcune caratteristiche influenzano il risultato:

- **Operandi non definiti.** Se esistono degli **operandi non definiti**, può accadere che l'espressione venga valutata solo con un preciso ordine. Per esempio, con l'espressione $(a == 0?b : b/a)$, è chiaro che b/a può essere una divisione per 0, quindi non definita. Tuttavia è necessario che venga valutata solo per divisioni diverse da zero, quindi un linguaggio di programmazione può essere:
 - **Valutazione lazy** (o corto circuito), ovvero gli operandi vengono valutati solo quando è necessario, quindi quando l'espressione è sempre definita.
 - **Valutazione eager**, ovvero gli operandi vengono valutati sempre.

Un ottimo **esempio** per comprendere la distinzione: [link](#)

- **Effetti collaterali** (*side effects*). Sono **effetti non legati all'oggetto di cui vi è la manipolazione**, per esempio: la valutazione di un'espressione provoca un cambiamento della memoria; una funzione modifica i propri parametri o modifica variabili non locali. Un **esempio** più tangibile è $((a + f(b)) * (c + f(b)))$ con la funzione $f(b)$ definita come $b++$ e l'espressione ritorna valori diversi a seconda dell'ordine di valutazione.
- **Aritmetica finita.** Nelle macchine l'**aritmetica è limitata dall'architettura, quindi i numeri non sono infiniti ed esiste un massimo numero rappresentabile**. I **problema di valutazione** avviene quando le espressioni coinvolgono tale numero.
Per **esempio** se viene eseguita una somma prima di una differenza, è possibile cambiare il risultato se la somma calcola un numero maggiore del massimo numero rappresentabile.

3.5 Semantica delle espressioni

La **semantica delle espressioni** di un linguaggio imperativo *IMP* cerca di descrivere la semantica senza interessarsi della grammatica, la quale include tutte le regole di associatività e precedenza. Quindi, vengono ignorante anche le parentesi, nonostante tutte queste regole rendono un'espressione chiara e non ambigua.

L'aspetto d'interesse è la **struttura induttiva** e non la struttura sintattica. Per questo motivo si introducono alcune definizione teoriche:

Definizione 3

Le **espressioni** \mathcal{E} è un **insieme di (alberi di valutazione) di espressioni valutate ad intero o booleano**. Gli elementi solitamente vengono rappresentati con la lettera e .

Definizione 4

I **numeri** \mathcal{N} è un **insieme di numeri reali nella macchina**. Gli elementi solitamente vengono rappresentati con le lettere m , n o p .

Definizione 5

I **booleani** \mathcal{B} è un **insieme di valori booleani**. Composto solo da **true** e **false**. Gli elementi solitamente vengono rappresentati con la lettera t .

Questi insiemi vengono usati per definire il **sistema di transizione**:

$$\Gamma = \mathcal{E}, \quad T = \mathcal{N} \cup \mathcal{B}, \quad \text{op} \in \{+, -, *, =\} \quad \text{bop} \in \{=, or\}$$

- Γ rappresenta l'insieme delle configurazioni;
- \mathcal{E} rappresenta l'insieme delle espressioni da valutare;
- T rappresenta l'insieme delle configurazioni terminali;
- op rappresenta l'insieme delle operazioni ammesse;
- bop rappresenta una metavariabile.

Quindi, viene definito l'insieme delle configurazioni come l'insieme delle espressioni da valutare ($\Gamma = \mathcal{E}$); l'insieme delle configurazioni terminali sono un tutti i valori numerici e booleani ($T = \mathcal{N} \cup \mathcal{B}$).

3.6 Regole di transizione

Si espongono qua di seguito le **regole di transizione**. Si tenga conto che con **bop** in grassetto si identifica il simbolo sintattico di un'operazione e con bop normale si identifica il simbolo dell'operazione nella macchina sottostante.

Si introducono le prime tre **regole necessarie per valutare espressioni aritmetiche**:

- È un **assioma e valuta l'espressione sintattica contenente un operatore aritmetico nel valore che esso rappresenta**. Per esempio, l'espressione $3 + 5$ viene valutata letteralmente come il valore 3, l'operazione di somma + e il valore 5.

$$\mathcal{E}_1: \begin{array}{c} m \text{ } \mathbf{op} \text{ } n \rightarrow p \\ \text{se } m \text{ op } n = p, \\ m, n, p \in \mathcal{N} \end{array}$$

- È una **regola induttiva e indica che se gli operandi non sono valori primitivi allora è necessario valutarli**.

$$\mathcal{E}_2: \frac{e \rightarrow e'}{e \text{ } \mathbf{op} \text{ } e_0 \rightarrow e' \text{ } \mathbf{op} \text{ } e_0}$$

- La regola stabilisce che **nel momento in cui l'operando a sinistra è un valore allora è possibile iniziare a valutare l'operando a destra**. Ovviamente se anche l'operatore a destra è un valore allora si ricade nella prima regola ed è possibile restituire il valore finale.

$$\mathcal{E}_3: \frac{e \rightarrow e'}{m \text{ } \mathbf{op} \text{ } e \rightarrow m \text{ } \mathbf{op} \text{ } e'}$$

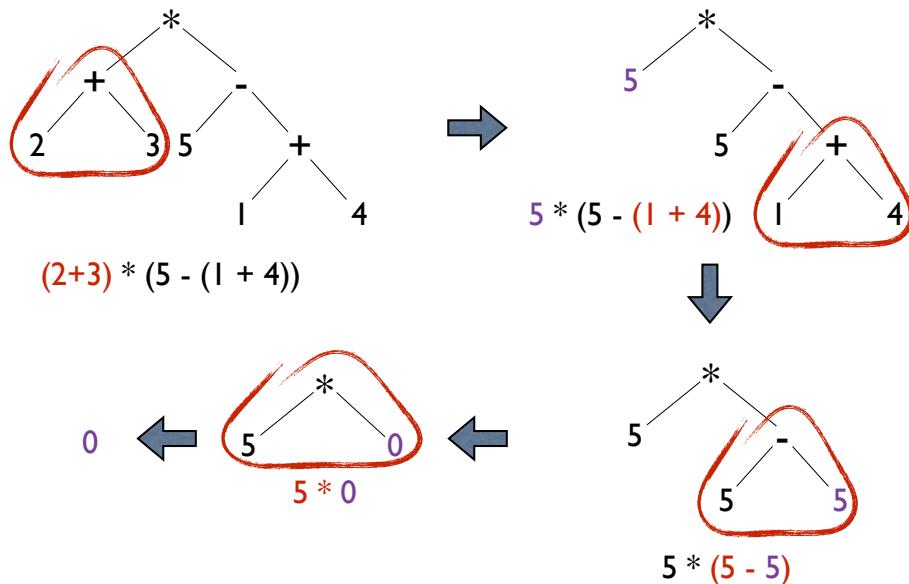
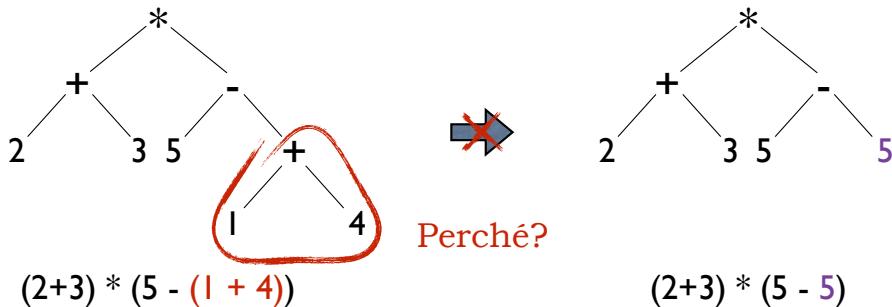


Figura 12: Esempio di applicazione delle regole aritmetiche.

Attenzione! Non è possibile eseguire una derivazione di questo tipo:



Questo perché le regole impongono la valutazione da sinistra verso destra e nella figura viene eseguita una valutazione da destra verso sinistra. Nonostante sia possibile implementare una nuova regola, non avrebbe senso poiché aumenterebbe la possibilità di errore nell'implementazione.

Si continua con le regole necessarie per valutare le espressioni booleane (ricordando che **bop** è il simbolo sintattico di un'operazione e bop è il simbolo dell'operazione nella macchina sottostante):

- È un assioma che **valuta l'espressione sintattica contenente un operatore, nel valore che esso rappresenta**. Ovviamenete con valori booleani, i valori coinvolti devono essere tali.

$$\mathcal{E}_4: k_1 \text{ bop } k_2 \rightarrow t \quad \text{se } k_1 \text{ bop } k_2 = t, \\ k_1, k_2 \in \mathcal{B} \cup \mathcal{N}, \quad t \in \mathcal{B}, \quad \text{bop} \in \{=, \text{or}\}$$

- Questa regola è una modifica alla precedente poiché **ammette che l'espressione contenga operatori binari booleani**. L'ordine di valutazione è da sinistra verso destra.

$$\mathcal{E}_{3'}: \frac{e \rightarrow e'}{e \text{ bop } e_0 \rightarrow e' \text{ bop } e_0}$$

- Questa regola stabilisce che nel momento in cui l'**operando a sinistra è un valore booleano allora è possibile iniziare a valutare l'operando a destra**. Ovviamemente, se l'operatore a destra è un valore booleano, allora si ricade nell'assioma \mathcal{E}_4 ed è possibile restituire il valore finale.

$$\mathcal{E}_5: \frac{e \rightarrow e'}{t \text{ op } e \rightarrow t \text{ op } e'}$$

- Nel caso booleano è necessario aggiungere la **regola per l'operatore unario**. Quindi si crea l'assioma che **restituisce il valore corrispondente all'applicazione dell'operatore sul valore rappresentato**.

$$\mathcal{E}_6: \text{not } t_1 \rightarrow t \quad \text{se } \text{not } t_1 = t, \quad t_1 \in \mathcal{B}$$

- Sempre nel caso booleano, si deve creare la **regola di valutazione**, analoga alle precedenti.

$$\mathcal{E}_7: \frac{e \rightarrow e'}{\text{not } e \rightarrow \text{not } e'}$$

3.7 Valutazione ed equivalenza

Definizione 6

La **valutazione delle espressioni** è una funzione $\text{Eval} : \mathcal{E} \rightarrow \mathbb{N} \cup \mathbb{B}$ che descrive il comportamento dinamico delle espressioni restituendo il valore in cui esse sono valute:

$$\text{Eval}(e) = k \iff e \xrightarrow{*} k$$

Definizione 7

L'**equivalenza di espressioni** è una **relazione** $\equiv \subseteq \mathcal{E} \times \mathcal{E}$ definita come segue:

$$e_0 \equiv e_1 \iff \text{Eval}(e_0) = \text{Eval}(e_1)$$

4 Dichiarazioni

4.1 Identificatori

Gli **identificatori** sono sequenze di caratteri, che hanno lo **scopo di denotare** (identificatore appunto) qualcos'altro.

In altre parole, gli **identificatori** sono nomi usati per riferire altri **elementi del linguaggio**, come le procedure, o della macchina sottostante, come le colle di memoria, durante la **computazione**, senza necessariamente conoscerne il valore a priori.

Gli **identificatori** sono detti **variabili** quando identificano celle di memoria.

I **nomi** sono fondamentali per gli identificatori perché sono **facili da ricordare** e consentono di effettuare un processo di **astrazione**. Inoltre, è importante sottolineare il fatto che un identificatore ed un oggetto denotato non sono la stessa cosa:

- Un identificatore può denotare più elementi;
- Un elemento può essere denotato da più identificatori diversi (*aliasing*)

Un **esempio** è il seguente codice:

```
1 const pi = 3.14;
2 int x;
3 void f() {...};
```

Gli oggetti denotati sono una costante, una variabile e una procedura:

- 3.14 è una costante;
- x è una variabile;
- {...} è una procedura

Mentre pi, x, f sono nomi.

In definitiva, gli identificatori non possono essere stringhe di caratteri qualunque, ma devono seguire delle regole così che il *parser* del linguaggio li possa riconoscere come tali.

Definizione 1

Gli **identificatori** sono una sequenza di caratteri usata per rappresentare o denotare un altro oggetto.

4.2 Bindings

4.2.1 Legami

Definizione 2

Il ***binding occurrences*** o in italiano la **creazione binding** è il momento in cui viene definito il nome (identificatore) e il suo significato (denotazione).

In **matematica** questo accade ogni volta che viene posto un problema enunciando (e.g.) “sia $x \dots$ ”. Viene creato un *binding* tra il significato dichiarato e il nome che verrà poi utilizzato nella specifica del problema. In **logica** i *binding* sono creati dai quantificatori (universali o esistenziali).

In **programmazione** i *binding* sono creati dalle dichiarazioni. Successivamente, il nome può essere correttamente utilizzato per riferirsi/rappresentare il suo significato.

Definizione 3

L’***applied occurrences*** o in italiano l’**applicazione binding** è il momento in cui viene utilizzato il nome (identificatore) per accedere al suo significato (denotazione).

In programmazione, come in matematica, non è sempre necessario creare i *binding* prima del loro utilizzo, dipende dal linguaggio di programmazione.

Un **esempio** di creazione e applicazione:

$$\sum_{i=1}^n \sum_{j=1}^m a_{ij} \cdots b_{jk}$$

↓

$$\underbrace{\sum_{i=1}^n \sum_{j=1}^m}_{\text{Binding occurrences } i,j} \overbrace{a_{ij} \cdots b_{jk}}^{\text{Applied occurrences } i,j}$$

La i e j sotto il simbolo di sommatoria sulla sinistra indicano la creazione del *binding* e la i, j a pedice dei simboli a e b sulla destra indicano l’applicazione del *binding*.

4.2.2 Scope

Esistono anche le occorrenze libere che sono simili a quelle applicate con l'unica differenza che riguarda il raggio di azione (*scope*) di un'occorrenza di definizione.

Definizione 4

Le *free occurrences* o in italiano le **occorrenze libere** (non legate) è il momento in cui viene utilizzato un nome (identificatore) il cui significato (denotazione) non è stato definito.

Definizione 5

Lo *scope* di un *binding* o in italiano il **raggio d'azione di una definizione** è la definizione di spazio in cui è possibile utilizzare un nome per rappresentare il significato associato da un *binding*.

Un **esempio** di creazione e applicazione:

$$\sum_{i=1}^n \sum_{j=1}^m a_{ij} \cdots b_{jk}$$

↓

$$\underbrace{\sum_{i=1}^n \overbrace{\sum_{j=1}^m a_{ij} \cdots b_{jk}}^{\text{Scope di } j}}$$

Scope di *i*

La *n*, la *m* e la *k* sopra il simbolo di sommatoria sulla sinistra e al pedice della *b*, indicano le occorrenze libere, cioè non legate. Invece, lo *scope* di *j* si concentra sulla sommatoria più innestata, mentre lo *scope* di *i* sulla sommatoria più esterna.

4.2.3 Bindings nei linguaggi di programmazione

Si riportano le definizioni già date ma con una veste diversa (non so il motivo di questa scelta, ma sulle slide è così...).

Definizione 6

Definizione - *Binding occurrence*: un identificatore in posizione di definizione quando si *(ri)definisce il significato* dell'identificatore.

Definizione 7

Uso - *Applied occurrence*: un identificatore in posizione di uso quando si *riferisce/denota il significato* definito da una definizione.

Definizione 8

Libera - *Free occurrence*: un identificatore in posizione libera se il suo uso non è nel *raggio di azione (scope)* di una definizione.

4.2.4 Tipi di bindings nei linguaggi di programmazione

4.2.5 Tempi

4.3 Semantica: Identificatori, ambienti e dichiarazioni

4.3.1 Termini chiusi e ground

4.3.2 Ambienti nei linguaggi imperativi

4.3.3 Espressioni con identificatori

4.3.4 Identificatori liberi

4.4 Nuove regole

4.5 Tipo

4.5.1 Binding di tipo (*Type binding*)

4.6 Ambiente statico e semantica statica

4.6.1 Semantica statica delle espressioni

4.7 Compatibilità di ambienti

4.8 Espressioni con identificatori costanti - Regole