

LINGUAGGI MASTER RIASSUNTO :

INTERPRETE

Un interprete per un generico linguaggio L, scritto in Lo, è un programma che riceve un programma scritto in L e un input per questo programma, lo esegue sull'input fornito utilizzando subentrate le primitive messe a disposizione dalla macchina astratta di Lo, e ritorna l'output risultante. Un interprete tipico è organizzato in più

fasi e componenti, ciascuna responsabile di un pezzo preciso dell'esecuzione.

I più rilevanti sono : ho sbagliato, questa parte riguarda compilatore.

1) Lexer (analisi lessicale) : trasforma la sequenza di caratteri in token (parola-chiave, identificatori, numeri, simboli o operatori).

2) Parser (analisi sintattica) : usa i token estratti per costruire un albero di antassi astratta (AST), che rappresenta la struttura generica del programma;

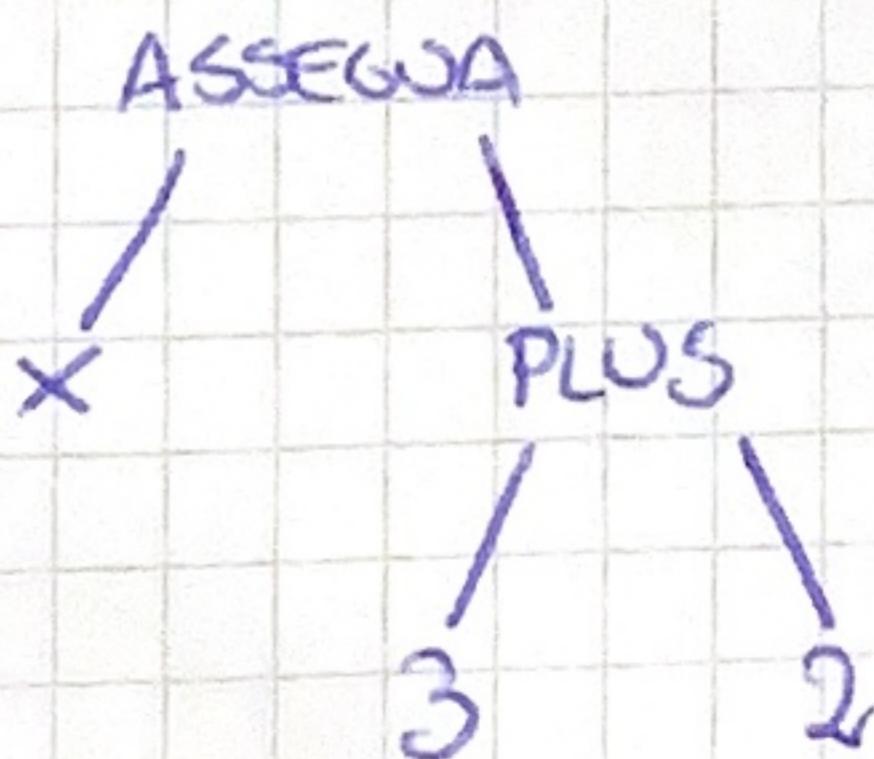
3) Evaluator (calcolazione semantica) : "cammina" sull'AST ed esegue semanticamente le istruzioni, aggiornando uno stato di opportuno mapping tra variabili e valori;

N.B:

AST (Abstract Syntax Tree) non è altro che una versione semplificata e strutturata del codice sorgente. Esempio:

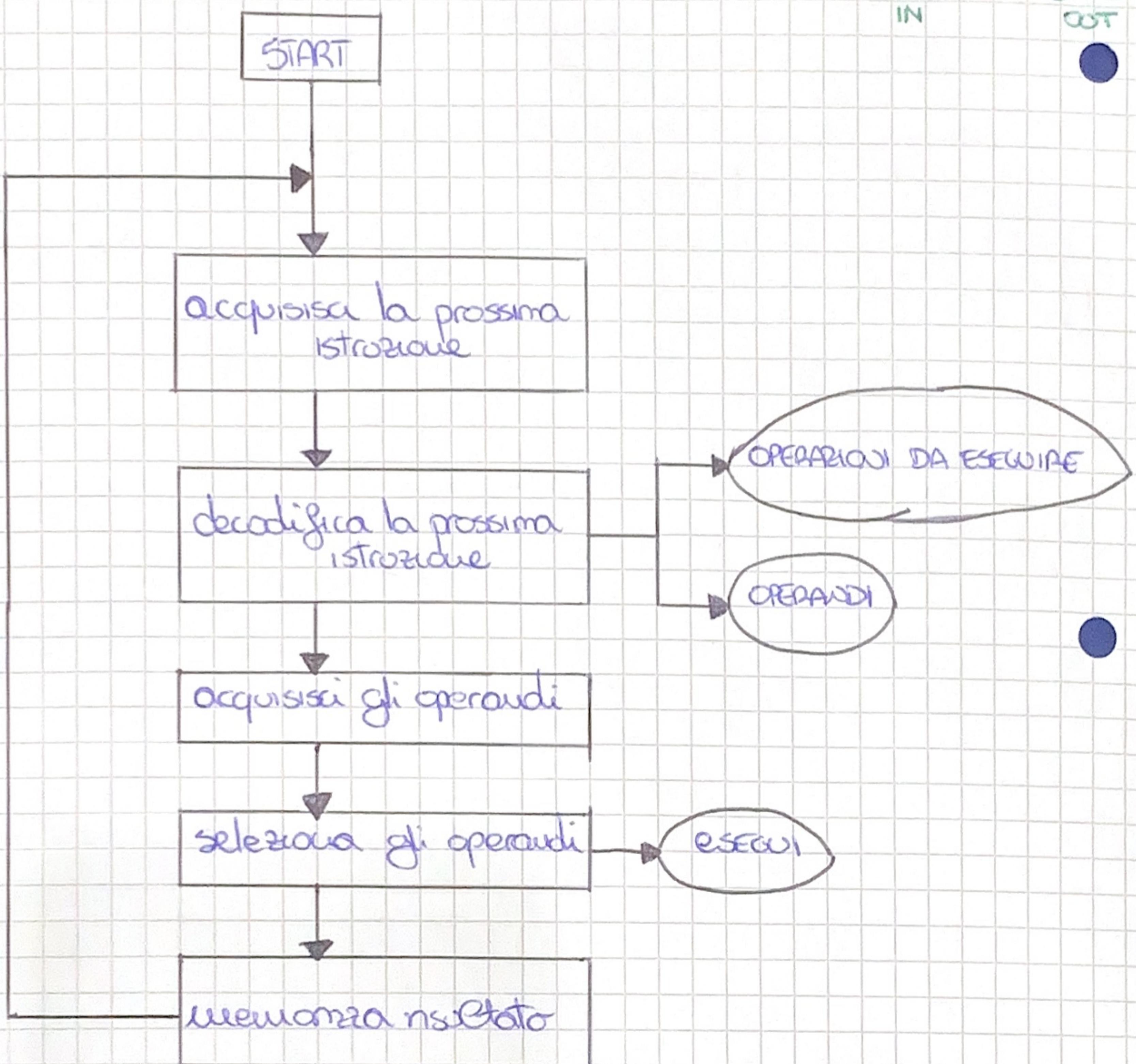
SRC : AST :

$x = 3 + 2$



Flow-chart interprete:

$$I^{LOL} = (\text{Prog}^L \times \underbrace{\text{Data}}_{\substack{\text{IN} \\ \text{OUT}}}^L) \rightarrow \text{Data}^L$$



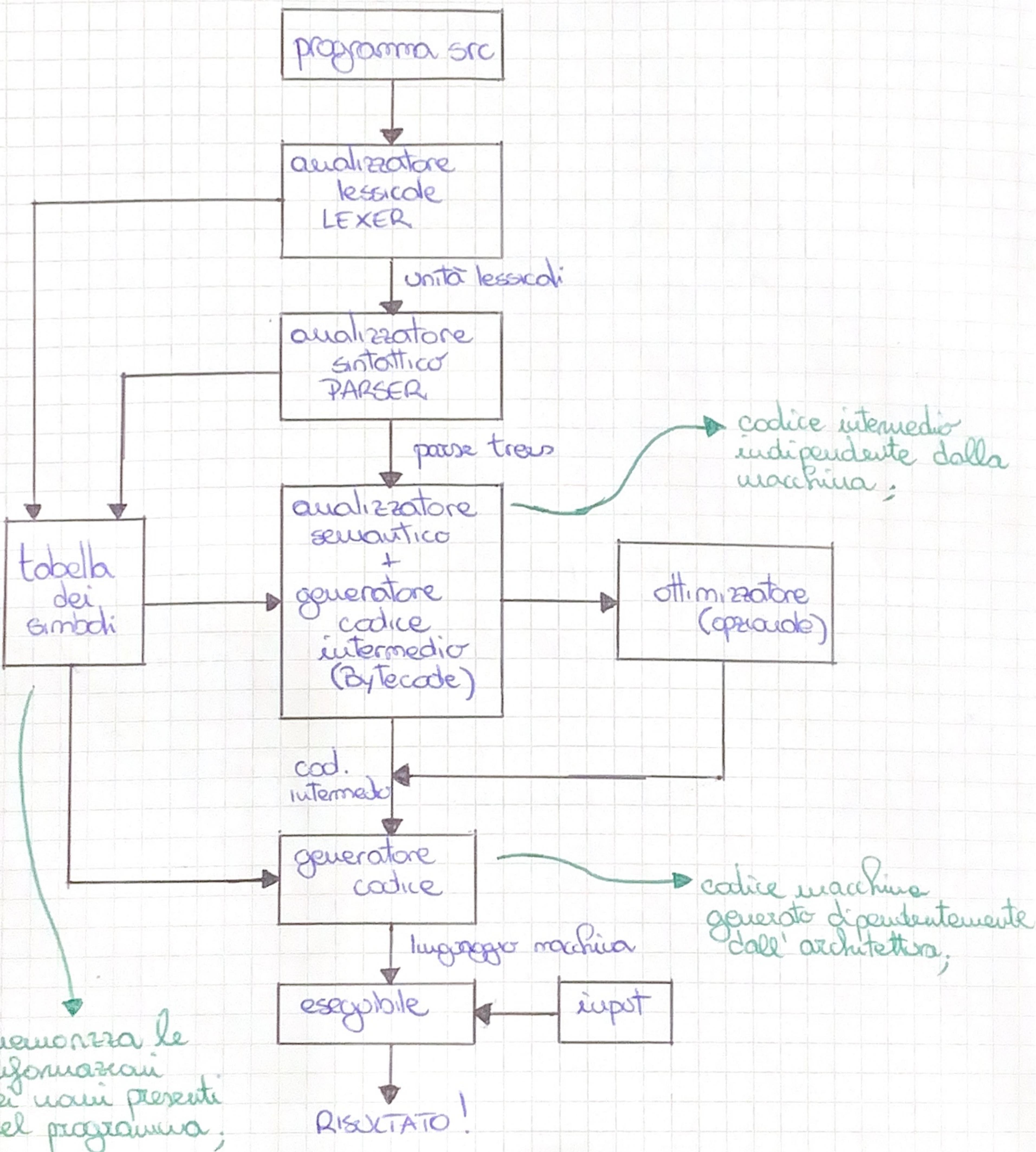
Campilatore

Un compilatore è un programma C^{LL} che traduce, preservando la semantica e la funzionalità, un programma scritto in un linguaggio sorgente L, in un programma scritto in linguaggio target L₀, quindi esegibile direttamente alla macchina astratta per L₀.

$$C^{LL}: \text{Prog}^L \rightarrow \text{Prog}^{L_0}$$

Quindi l'interprete esegue direttamente, mentre il compilatore genera un altro programma e delega la sua esecuzione.

Flow-chart compilatore :



ESEMPIO SUL' INDUZIONE MATEMATICA

Vedremo un esempio guidato per dimostrare che $\forall u \in \mathbb{N}$, $u+u^2$ è pari.

PASSO 1) Scrivere chiaramente la proprietà P da dimostrare :

$$P(n): u+u^2 \text{ pari}$$

oppure in forma algebrica

$$\exists k \in \mathbb{N} : u+u^2 = 2k$$

PASSO 2) CASO BASE: spesso $u=0$ || $u=1$, dimostriamo P per casi elementari;

$$P(0) : 0+0^2 = 0 \text{ pari } \checkmark$$

$$P(1) : 1+1^2 = 2 \text{ pari } \checkmark$$

PASSO 3) formula la giusta ipotesi induttiva su ' u ' :

$$hp : u+u^2 = 2k \text{ per qualche } k.$$

Quindi stiamo assumendo per vero $P(n)$.

PASSO 4) Dimostriamo induttivamente su ' $n+1$ ', quando l'hp posta:

$$P(n+1) = (n+1)+(n+1)^2$$

L'obiettivo è quindi arrivare alla forma dell'hp (2k in questo caso).

$$(n+1)+(n+1)^2 = (n+1)+u^2+1+2u = u+1+u^2+1+2u$$

$$\xrightarrow{\substack{\text{riordinato} \\ \text{soltanto per} \\ \text{evidenziare } (nu^2)}} (n+n^2) + 2n + 2 = \underset{\substack{\downarrow \\ hp}}{2k} + 2u + 2 = \underset{\substack{\downarrow \\ \text{razzo}}}{2(k+u+1)}$$

dove è chiaramente pari in quanto multiplo di 2 come nell'hp.

Scoping:

Sostanzialmente lo scoping è ciò che determina l'ambito di visibilità di una variabile. Il sistema deve capire a che variabile faccio riferimento tramite identificatore e, soprattutto, on' attenta valutazione del "luogo" in cui viene richiamata. Esistono due strategie fondamentali di scope, ed entrambe rispondono alla domanda: "a quale valore della var. 'x' faccio riferimento in questo punto del codice?". Abbiamo quindi:

1) scoping statico (detto anche 'lessicale'): guarda dove una var. è definita nel codice, a tempo di compilazione;

2) scoping dinamico: guarda 'chi ha chiamato chi' avendo l'ambiente di esecuzione, a tempo di esecuzione;

Chiaramente, ha senso parlare di scope quando i riferimenti alle var. di un determinato programma sono var locali e var globali, ossia quando la dichiarazione di una var. avviene al di fuori della modifica attuale, in uno dei blocchi esterni (globale è il caso più generico).

ESEMPIO:

Dato il seguente codice:

```
{  
    int x = 1;  
    void A()  
    {  
        int x = 2;  
        B();  
    }  
    void B()  
    {  
        print x;  
    }  
}
```

→ se il linguaggio usa scoping statico,
B var vede x=2 definita in A perché
B è scritta fuori da A, quindi stampa 1;
→ se il linguaggio usa invece scoping dinamico
B viene chiamato all'interno di A, quindi
ne eredita l'ambiente, e stampa 2;

Più formalmente:

STATICO: valuta tramite l'analisi sintattica del programma (senza eseguirlo);

DINAMICO: valuta basandosi sull'ordine effettivo delle chiamate a run-time
(durante l'esecuzione);

Scope statico e link statico:

Il link statico è un pontatore all'attivazione della funzione che contiene staticamente quella in esecuzione, ossia un collegamento che "nsale" i contesti lessicali che la contengono per trovare la dichiarazione corretta.

Il calcolo di un link statico avviene tramite la formula:

$$k = Sd(\text{chiamante}) - Sd(\text{chiamato}) + 1;$$

dove 'Sd' sta per static depth, 'Sd(chiamante)' è il livello di annidamento della funzione che fa la chiamata, 'Sd(chiamato)' il livello d'annidamento della fn. che stai chiamando, 'k' invece è quante volte devi seguire il link statico partendo dal chiamante per trovare il record di attivazione. Se $k=0$, il link statico punta direttamente all'inizio del chiamante, altrimenti $k \neq 0$ richiede n salti.

Esempio:

```
Void A() {  
    Void B() {  
        C();  
    }  
    Void C() {  
        // usa x  
    }  
    B();  
}
```

// Sd=1

// Sd=2

// chiamata

// Sd=2

'C' è definita con $Sd=2$, ma chiamata all'interno di B, anch'essa con $Sd=2$. Quindi:

$$- Sd(\text{chiamante}) = 2 \text{ (B)};$$

$$- Sd(\text{chiamato}) = 2 \text{ (C)};$$

$$- \underline{k = 2 - 2 + 1 = 1};$$

Il link statico di C deve uscire di un livello partendo dal chiamante (B), quindi il link statico di C punta all'ambiente di A.

Scoping dinamico:

Lo scoping dinamico invece guarda la sequenza di chiamate a funzione che hanno portato all'esecuzione del codice attuale. In altre parole, il valore di una variabile viene cercato nell'ambiente locale della fn. corrente. Se mai venisse trovata lì, si risalirebbe la pila di chiamate (sequenza di fn. invocate per arrivare a quel punto) fino a trovare una definizione della variabile. L'ambiente in cui una variabile viene risolta, è quindi la "storia" dell'esecuzione, e non la struttura testuale.

Esempio:

```
j var x=10;  
d   Void A {  
o     print x;  
g   }  
o Void B {  
d     var x=20;  
g     A();  
g   }  
B();
```

→ con lo scoping dinamico, B stampa di x in A() non ha definizione, quindi, nsalendo al chiamante B, troiamo la definizione x=20, pertanto viene stampato questo valore;

→ se invece avessimo usato scoping statico, A che stampa si troverebbe allo stesso livello di profondità di B, quindi il suo ambiente sarebbe quello globale, dove x=10;

All'interno di questo approccio dinamico ci sono due modi principali in cui le tracce dell'esecuzione vengono seguite, e sono il shallow e il deep binding.

NB BINDING = associazione nome - valore, dove valore può essere una classe di menu, un valore effettivo, una funzione o la definizione di una struttura.

Shallow binding:

Utilizza una tabella globale di var. e valori. Quando una fn. definisce una var. già presente, il valore globale è temporaneamente aggiornato. La risoluzione cerca il valore corrente nella tabella globale. Semplifica da implementare ma con potenziali effetti collaterali non locali.

Deep binding:

Associa alla var l'ambiente in cui è stata definita. La risoluzione di var. non locali avviene in questo ambiente "catturato" al momento della definizione, ma al momento della chiamata. Comportamento più prevedibile, simile allo scoping statico in certi casi, ma implementazione più complessa.

In sintesi, shallow guarda al val. attuale nella tabella globale, deep invece al val. nell'ambiente di definizione della funzione.

Esempio:

```
var x=10;  
Void A() {  
    print x;  
}  
Void B() {  
    var x=20;  
    A();  
}  
B();
```

→ con scope dinamico di tipo shallow,
l'output è $x=20$ dato che quando viene chiamata
 $A()$ in $B()$, quest'ultima ha aggiornato temp. B.
tabella delle variabili rispetto alla definizione globale
 $x=10$. Una volta terminata $B()$, il valore precedente
di x viene ripristinato;

→ con scope dinamico di tipo deep, $A()$ e $B()$
hanno ambiente locale di stesso livello, dove $x=10$.
Quando viene chiamata $B()$, x viene posta a 20,
ma nella nuova x locale a B , non influenzando
mai la tabella e l'ambiente di A . Quando $A()$
infatti, provando a stampare, non trova nel suo
ambiente locale una definizione per x , cerca nell'
ambiente associato alla definizione di A , quello
globale in questo caso, dove $x=10$.

Ricorsione e ricorsione in coda:

NB nell'esame viene spesso chiesto di scrivere / analizzare funzioni ricorsive, per poi trasformarle in fn. ricorsive in coda.

Una funzione è ricorsiva quando chiama se stessa per risolvere un problema. Oggi chiamata ricorsiva deve avvicinarsi a un caso base (condizione di terminazione). Se manca il caso base o non viene raggiunto, la ricorsione continua all'infinito. Oggi chiamata a fn. ricorsiva crea un nuovo record di attivazione per tenere traccia delle variabili locali e a del punto in cui tornare dopo la chiamata. Le chiamate quindi si "impilano", e il calcolo avviene a "retrocor", mentre le chiamate ricorsive tornano. Questo può talvolta causare stack overflow per ricorsioni profonde.

ESEMPIO:

```
int fattoriale ( int n ) {  
    if ( n == 0 )  
        return 1;  
    else  
        return n * fattoriale ( n - 1 );  
}
```

Ricorsione in coda (tail recursion):

Questa è una forma speciale di ricorsione, in cui la chiamata ricorsiva è l'ultima operazione eseguita dalla funzione. Non ci sono operazioni in sospeso dopo la chiamata ricorsiva. L'importanza della ricorsione in coda è che può essere ottimizzata dai compilatori. Invece di allocare un nuovo frame di stack per ogni chiamata ricorsiva, il compilatore può riutilizzare il frame corrente, trasformando la ricorsione in un ciclo, equivalente al for e al while in termini di efficienza in memoria. Il calcolo avviene "in avanti", accumulando il risultato nei parametri, mentre la chiamata ricorsiva è l'ultima azione. Gli unici svantaggi sono che non

tutti i linguaggi compilatori la supportano, e a volte può essere meno intuitiva della ricorsione standard. Al contrario tuttavia, possiamo "identificare" verificando che la chiamata ricorsiva sia effettivamente l'ultima operazione, e non vi siano operazioni in sospeso dopo.

ESEMPIO:

```
int fattoriale (int n, int accumulatore) {  
    if (n == 0)  
        return accumulatore;  
    else  
        return fattoriale (n-1, accumulatore * n);  
}  
  
int main () {  
    // richiesta n  
    int result = fattoriale (n, 1);  
    printf ("%d", result);
```

Passaggio di parametri:

Il passaggio di parametri è un meccanismo fondamentale nei linguaggi di programmazione. Determina come i valori e le variabili stesse, vengano forniti a una funzione quando questa viene chiamata. Esistono principalmente 2 modi nel passaggio di parametri:

- passaggio per valore;
- passaggio per riferimento;

Passaggio per valore:

Nel passaggio per valore, la fnz. riceve una copia del valore dell'argomento, e qualsiasi modifica apportata non si riflette sull'originale. In termini tecnici, viene allocata una nuova area di memoria per contenere questa copia, assicurando che la funzione non possa accidentalmente alterare i dati originali. Questo garantisce protezione e isolamento tra chiamante e chiamato.

Esempio:

```
void modifica(int x) {  
    x = x + 10;  
    printf("%d", x);  
}
```

```
int main() {  
    int y = 10;  
    modifica(y);  
    printf("%d", y);  
    return 0;  
}
```

In questo esempio, y rimane inizialmente 10, mentre modifica lo modifica a 20.

Passaggio per riferimento:

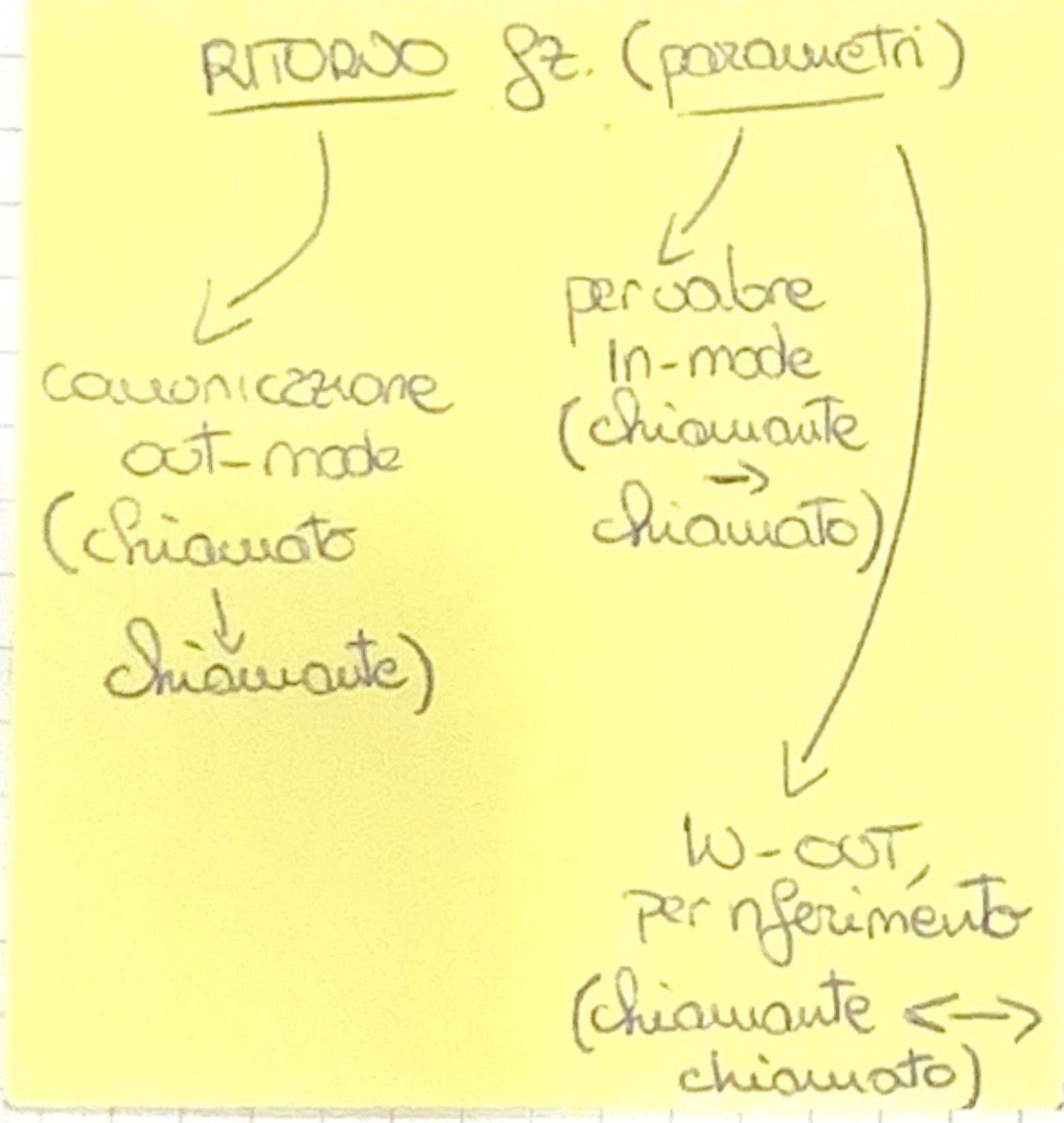
La fnz. riceve una copia del valore, ma l'indirizzo di memoria della variabile originale. Qualsiasi cambiamento del parametro all'interno della funzione si riflette direttamente sulla variabile originale nel codice chiamante. Richiede quindi una gestione attenta per evitare effetti indesiderati; ma non necessita di memoria aggiuntiva per contenere la "copia".

Esempio:

```
void modifica (int *x) {  
    x = x + 10;  
    printf ("%d", *x);  
}
```

Qui 'x' è un riferimento a 'y',
per tanto entrambe le printf stamperebbero 20.

```
int main() {  
    int y = 10;  
    modifica(y);  
    printf ("%d", y);  
    return 0;  
}
```



Semantica:

Nel contesto dei linguaggi di programmazione, la semantica si occupa di definire il significato dei programmi, cioè cosa succede quando un programma viene eseguito. In particolare dobbiamo distinguere semantica statica da semantica dinamica. La prima si concentra sul significato degli aspetti di un programma che possono essere determinati prima dell'esecuzione. Ciò include il controllo dei tipi, il controllo dei flussi, e altre proprietà verificate durante la compilazione.

Semantica dinamica:

La semantica dinamica specifica come lo stato di un sistema computazionale cambia nel tempo durante l'esecuzione. I concetti chiave per la semantica dinamica sono lo stato, la configurazione, la transizione e la derivazione.

STATO → Lo stato di un programma in un dato momento è l'insieme di tutte le informazioni rilevanti che descrivono il sistema. Questo include:

- ① **MEMORIA**: una funzione (o tabella) che mappa le locazioni di memoria ai loro valori;
- ② **AMBIENTE**: una funzione (o tabella) che mappa i nomi delle variabili alle loro locazioni di memoria, o direttamente ai valori in taluni casi;
- ③ **CONTESTO DI CONTROLLO**: informazioni su quale istruzione è in esecuzione, quali funzioni sono state chiamate, ecc. ecc;

CONFIGURAZIONE → una configurazione è una coppia (o una tupla) che descrive lo stato del programma in un dato momento. Per esempio una terza $\langle c, \sigma \rangle$ dove 'c' è il comando da eseguire, ' σ ' è l'ambiente, mentre ' σ ' è la memoria.

TRANSIZIONE → è un passo di computazione che cambia lo stato del programma. La semantica dinamica definisce come le configurazioni cambiano da un passo all'altro.

Il modo più comune per specificare la semantica dinamica è attraverso la semantica operazionale. Essa descrive l'esecuzione di un programma in termini di come le istruzioni singoli manipolano lo stato del programma. Ci sono 2 approcci principali:

- **SEMANTICA A PICCOLI PASSI**: descrive l'esecuzione come una sequenza di singoli passi di computazione. Ogni passo specifica come una singola istruzione cambia lo stato;
- **SEMANTICA A GRANDI PASSI**: descrive l'esecuzione di un programma nella sua interezza, specificando la transizione tra lo stato iniziale e lo stato finale;

Struttura generale della soluzione a quest'esercizio:

- ① **DEFINIRE LO STATO INIZIALE**: identifica l'ambiente iniziale ' \emptyset ' e la memoria (σ) forniti nell'esercizio. L'ambiente mappa i nomi delle variabili alle loro locazioni di memoria (o ai valori), e la memoria mappa le locazioni di memoria ai valori;
- ② **APPLICARE LE REGOLE DI SEMANTICA DINAMICA**: inizia con la configurazione iniziale, che è una tupla composta dal comando da eseguire, dall'ambiente e dalla memoria. Applica le regole di semantica operazionale appropriate al comando, mostrando come lo stato (ambiente e memoria) cambia ad ogni passo. Ogni passo deve essere giustificato indicando quale regola di semantica è stata applicata;

③ MOSTRARE LA DERIVAZIONE PASSO-PASSO: scrivi la sequenza di transizioni,
dove ogni transizione mostra come la configurazione
cambia da uno stato all'altro. La derivazione continua
finché non si raggiunge uno stato finale (se esiste);

→ ④ DESCRIVERE IL RISULTATO FINALE: indica lo stato finale del programma
(ambiente e memoria) dopo l'esecuzione del comando.