

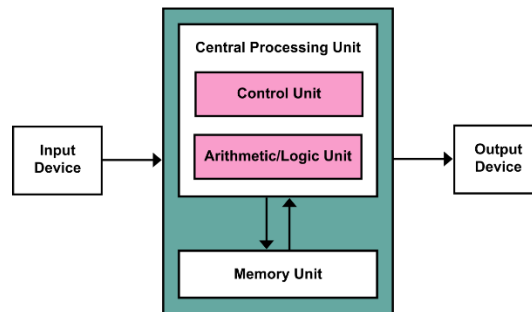
*Linguaggi*

**Fabiola F.**



# Indice

1 - Introduzione.....	4
2 – Descrivere i linguaggi.....	11
2 – Sintassi e categorie sintattiche .....	21
2- Semantica operativaale .....	23
2 – PL0 .....	25
--- end intro .....	28
Concetti necessari .....	27
3 – Espressioni .....	33
4 – Dichiarazioni.....	40
5 – Comandi (contesto) .....	49
5 - Comandi.....	53
6 - Procedure.....	63
6 – Procedure e parametri.....	73
7 - Paradigmi di programmazione .....	77



# 1 - Introduzione

Di fatto la macchina HW è in grado di interpretare solo il linguaggio binario, ovvero un linguaggio costituito esclusivamente da stringhe di bit. In linguaggio binario, dato e programma sono scritti allo stesso modo. Questo porta a un'enorme difficoltà:

- No distinzione fra dato e programma, che sarebbe necessaria dato che li tratto diversamente
- Manipolazione difficile.

→ Nasce dunque la necessità di avere un linguaggio.

## Perché studiarli?

1. Migliorare la capacità di sfruttare il linguaggio per scrivere algoritmi, poiché conoscendone il funzionamento si possono scrivere algoritmi più efficienti o più affini a come era stato pensato il linguaggio.
2. Scegliere il linguaggio in funzione dell'applicazione
3. Velocizzare l'apprendimento di nuovi linguaggi.

Linguaggi di programmazione: sono linguaggi formali le cui frasi sono programmi, ovvero forme concrete di algoritmi eseguibili o interpretabili da una macchina. Servono a rappresentare in modo finito oggetti potenzialmente infiniti; in particolare, vogliamo rappresentare dati e algoritmi.

Quindi, il PL deve poter rappresentare:

1. Oggetti finiti
2. Descrivere passi di calcolo
3. Eseguire, ipoteticamente, infiniti passi di calcolo.

**Programma:** frase in un PL. Rappresentazione finita di un insieme, potenzialmente infinito di passi primitivi di computazione.

**Algoritmo:** concetto più astratto: è una sequenza di task che si concretizza in un programma.

Ciascun algoritmo può essere concretizzato in tanti programmi diversi.

Per scrivere un programma è necessario stabilire:

- **Grammatiche context-free:** regole che permettono di costruire frasi grammaticalmente corrette
- **Categorie sintattiche:** elementi base – ovvero elementi terminali della grammatica – di cui abbiamo bisogno per definire i programmi in un linguaggio di programmazione

Gli algoritmi sono, di fatto, funzioni matematiche. Tuttavia, i linguaggi logico-matematici non sono necessariamente la soluzione più adatta a rappresentarli:

**Linguaggio matematico:** notazione rigorosa per rappresentare funzioni ma non sempre oggetti infiniti e computazioni, ovvero passi di calcolo.

- ✗ Non permette sempre di rappresentare oggetti infiniti in modo finito → Ad esempio, in insiemistica, la matematica permette di descrivere gli insiemi solo per enumerazione, quindi non arriva ad insiemi infiniti.
- ✗ Non descrive i passaggi di calcolo
- ✗ Non rappresenta tutti i problemi calcolabili.

**Linguaggio logico (logica proposizionale):** Regole e assiomi rendono possibile specificare il processo di computazione (ancora in modo implicito).

- ✓ Possiamo rappresentare in modo finito oggetti infiniti, per esempio con il modus ponens
- ✓ Fornisce direttamente un metodo di calcolo di ciò che si vuole rappresentare.
- ✗ Non è in grado di rappresentare in modo finito dimostrazioni infinite, ovvero dimostrazioni che richiedono

✓ Sì computazioni infinite (es. sommatoria)

un numero infinito di passi di esecuzione.  
→ Teorema di incompletezza di Godel

## Aspetti di progettazione

Le caratteristiche di un PL sono:

- **Leggibilità:** *Sintassi chiara, nessuna ambiguità, facilità di lettura e comprensione dei programmi.*
  - › + Meglio pochi modi per scrivere la stessa cosa
  - › + Ortogonalità: il significato degli operatori non deve dipendere dal contesto
  - › + Tipi di dati diversi (ad esempio, usare gli int come bool porta a confusione)
  - › + Sintassi semplice
  - › - Overloading degli operatori
- **Scrivibilità:** *Facilità di utilizzo di un linguaggio per creare programmi, facilità di analisi e verifica dei programmi.*
  - › + Ortogonalità, semplicità e sintesi: altrimenti il programmatore potrebbe non conoscerli tutti e non usarli adeguatamente
  - › + Supporto per l'astrazione: ad esempio, gli oggetti sono un'astrazione che mi permette di nascondere i dettagli implementativi
  - › + Espressività: rappresentare in modo coerente le operazioni. Ad esempio, mettere a disposizione un insieme di modi relativamente conveniente per specificare operazioni.
  - › - Molti costrutti: altrimenti il programmatore potrebbe non conoscerli tutti e non usarli adeguatamente
- **Affidabilità:** conformità alle specifiche. È legata alla safety: quanto riesco ad evitare gli errori?
  - › Type checking statico che controlla errori di tipo
  - › Gestione delle eccezioni e per permettere la continuazione dell'esecuzione e l'attuazione di eventuali misure correttive
  - › Non permettere l'aliasing: con l'aliasing ho più identificatori che puntano a una sola posizione, e rischio di non riuscire a modificarne solo uno...
- **Costo**
  - › Costo di addestramento
  - › Costo di compilazione ed esecuzione
  - › Costo di mantenimento

## Classificazione dei linguaggi

Abbiamo diverse tecniche di classificazione.

- **Basso livello:** hanno caratteristiche specifiche dipendenti dall'architettura.  
→ Binario, assembly...
- **Alto livello:** permettono una programmazione strutturata, in cui dati ed istruzioni hanno rappresentazioni diverse.

Linguaggi imperativi:	Linguaggi funzionali:	Linguaggi logici:
Astrazione di Von Neumann. Il concetto di variabile è l'astrazione logica della cella, mentre	Sono i più vicini alla matematica, ovvero descrivono i passi di calcolo come funzioni matematiche e	Usano logica, ovvero pattern matching e unificazione/sostituzione

l'assegnamento è l'operazione primitiva di modifica della cella di memoria, e quindi dello stato della macchina	quindi compongono e applicano funzioni. Qui una variabile è un placeholder per futuri valori.	come passo di calcolo primitivo. Ragionano per dimostrazione.
---	---	---

## Implementare linguaggi

Implementare un linguaggio significa renderlo comprensibile alla macchina da programmare. L'implementazione dipende da:

- Architettura → Von Neumann → Dati e programma sono in memoria, Memoria separata da CPU
- Scelte di progettazione sui programmi

Quando si opera su un linguaggio di programmazione a basso livello usiamo le primitive messe a disposizione della macchina. Quando usiamo un linguaggio ad alto livello, lavoriamo su una macchina che interpreta le istruzioni del linguaggio di programmazione, ed ignoriamo l'esistenza della macchina fisica. Implementare un linguaggio, quindi, significa realizzare la macchina astratta che interpreta il linguaggio.

### Macchina astratta

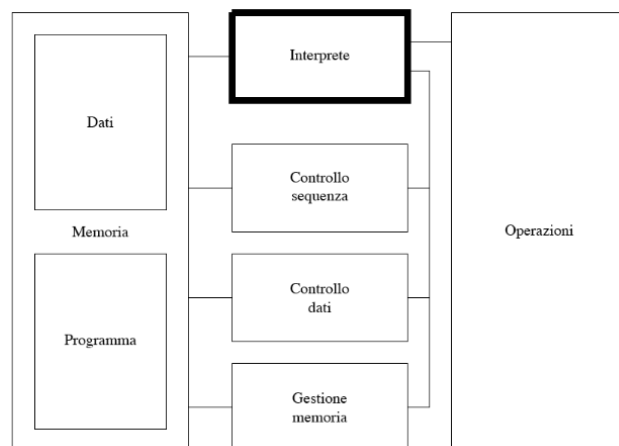
Dato un linguaggio  $L$ , la macchina astratta  $M_L$  per  $L$  è un insieme di **strutture dati** ed **algoritmi** che permettono di memorizzare ed eseguire i programmi scritti in  $L$ .

La collezione di strutture dati ed algoritmi serve per:

- Acquisire la prossima istruzione
- Gestire le chiamate ed i ritorni dai sottoprogrammi
- Acquisire gli operandi e memorizzare i risultati delle operazioni
- Mantenere le associazioni fra nomi e valori denotati
- Gestire dinamicamente la memoria

Il linguaggio  $L$  riconosciuto (interpretato) dalla macchina astratta viene anche detto linguaggio macchina. Formalmente è l'insieme di tutte le stringhe interpretabili da  $M_L$ .

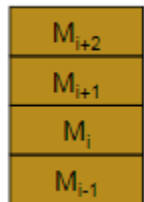
Questa macchina astratta  $M_L$  può essere realizzata:



HW	FW	SW
<p>Molto vicina a una macchina fisica, e concettualmente semplice. <b>Si realizza tramite dispositivi fisici</b>, e il linguaggio macchina è il linguaggio fisico/binario. Tipicamente usata per sistemi dedicati. → L'esecuzione è veloce, ma i linguaggi ad alto livello sono molto lontani dalle funzionalità a basso livello; inoltre è difficile da modificare.</p>	<p>Consiste nell'emulazione diretta della macchina fisica; è realizzata come <b>microprogramma scritto a basso livello</b>. Consiste di microistruzioni che specificano <b>semplici operazioni di trasferimenti dati tra registri</b>, da e per la memoria principale ed eventualmente attraverso circuiti logici che realizzano operazioni aritmetiche. I microprogrammi risiedono in speciali aree di memoria, in sola lettura.</p> <p>→ <b>Veloce e più flessibile della HW.</b></p>	<p>Simulazione software: le strutture e i dati sono tutti scritti nel linguaggio sottostante. Avendo la macchina astratta per <math>L'</math>, possiamo realizzare quella per <math>L</math> mediante opportuni programmi scritti in <math>L'</math> simulando le funzionalità di <math>M_L</math> per <math>L</math>. Si arriva ad avere gerarchie di macchine astratte.</p> <p>→ <b>Riduce la velocità ma aumenta la flessibilità.</b></p>

## Livelli di astrazione

In una macchina reale si parla di **gerarchia di macchine**: per permettere l'uso di linguaggi di programmazione ad alto livello, controllando la complessità, è stata data una struttura suddivisa a livelli di astrazione cooperanti ma sequenziali e indipendenti: ciascun livello è definito da un linguaggio  $L$  che è l'insieme delle istruzioni che il livello mette a disposizione per i livelli superiori.



- $M_i$ :
- 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  $M_{i-1}$

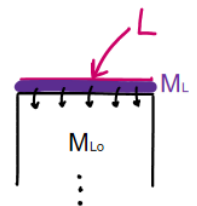
Stando al livello  $i$  può non essere noto  
(e in genere non serve sapere...) quale sia il livello 0 (hw)

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

La macchina fisica è controllata via macchine astratte, ognuna delle quali si preoccupa solo di quella immediatamente sottostante.

## Realizzare la macchina astratta

Abbiamo un linguaggio  $L$  da implementare e una macchina astratta  $M_{L0}$  macchina ospite (in linguaggio macchina  $L_0$ ).  $M_{L0}$  è il livello su cui vogliamo implementare  $L$ , e che mette le proprie funzionalità a disposizione di  $M_L$ .



Una macchina astratta può essere implementata in più modi: dato un linguaggio  $L$ , esistono infinite macchine astratte per  $L$ .

Per farlo, abbiamo uno spettro di scelte fra due estremi:

**Traduzione implicita**  
(interpretazione)



**Traduzione esplicita**  
(compilazione)

*Notazione:*

$Prog^L$  = insieme di programmi scritti in  $L$

$D$  = Insieme di dati  $\rightarrow$  input e output

$P^L \in Prog^L, in, out \in D$

$\llbracket P^L \rrbracket: D \rightarrow D$  semantica di  $P^L$  tale che  $\llbracket P^L \rrbracket(in) = out$  se  $P^L$  eseguito a partire da  $in$  restituisce  $out$

## Traduzione implicita: interprete

Prende ogni istruzione singolarmente, la traduce e la esegue.

Un interprete è un programma  $I^{L_0L}$  che esegue, sulla macchina astratta  $L_0$ , programmi in  $P^L$ .

$$Prog^L \times Dato(input) \rightarrow Dato(output)$$

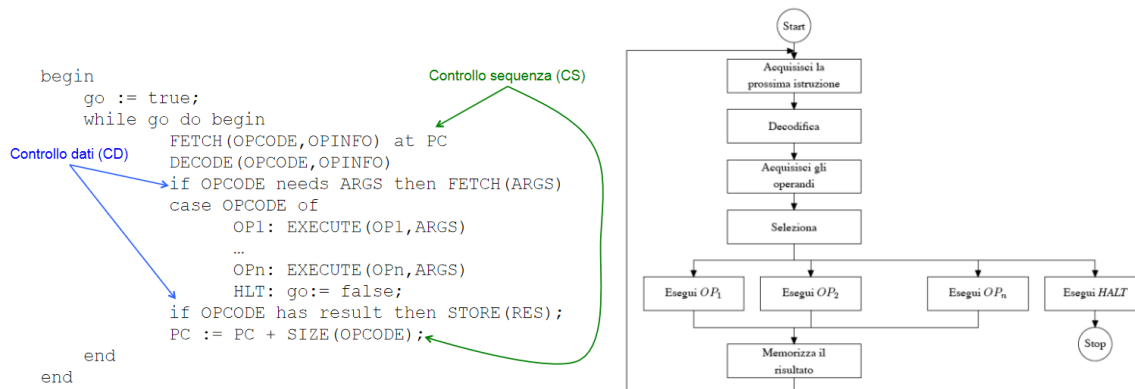
$$I^{L_0L}(P^L, input) = P^L(input)$$

In altre parole, 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 sottostante.

## Operazioni

Di fatto, un interprete è un ciclo che svolge operazioni per la simulazione delle istruzioni del linguaggio. Tali operazioni sono:

- **Elaborazione dei dati primitivi**
  - › I dati primitivi sono rappresentabili in modo diretto nella memoria. Le operazioni aritmetiche per elaborare questi dati sono direttamente implementate nella struttura della macchina (operazioni primitive)
- **Controllo di sequenza delle esecuzioni**
  - › È la gestione del flusso di sequenza, non sempre sequenziale. Nell'interprete ci sono strutture adeguate – tipo registro con la prossima istruzione o PC . che vengono manipolate da istruzioni specifiche
- **Controllo dei dati**
  - › Serve a controllare operandi e dati, recuperandoli anche mediante strutture ausiliarie. Riguardano le modalità di indirizzamento della emmoria e l'ordine con cui recuperare gli operandi.
- **Controllo della memoria**
  - › Sia il programma che i dati vanno memorizzati nella macchina, quindi è necessario gestire i processi di allocazione per dati e programmi. In una macchina astratta vicino all'HW questo processo è semplice, poiché i dati possono rimanere sempre nelle stesse locazioni; al contrario nelle macchine astratte SW esistono costrutti di allocazione e deallocazione che richiedono strutture dati opportune (pole) e operazioni dinamiche.



1. START
2. Acquisisci la prima istruzione
3. Decodifica per estrarre operazione e operandi
4. Acquisisci operandi dalla memoria
5. Esegui l'operazione
6. Memorizza l'eventuale risultato
7. Ripeti fino al raggiungimento dell'operazione di halt.

Ogni operazione è tradotta ed eseguita prima della successiva! → Costo computazionale potenzialmente alto: ad esempio in caso di cicli traduco il contenuto ogni volta.



Interpreti puri: implementazioni vecchie di linguaggi logici funzionali come LISP o PROLOG, linguaggi di scripting come JS, PHP

## Traduzione esplicita: compilatore

Un compilatore è un programma  $comp^{L \rightarrow L_0}$  che traduce (preservandone semantica e funzionalità) programmi scritti in  $L$  in programmi scritti in  $L_0$ , e quindi in eseguibili direttamente sulla macchina astratta per  $L_0$ .

### Compilatore da $L$ a $L_0$ , formalmente

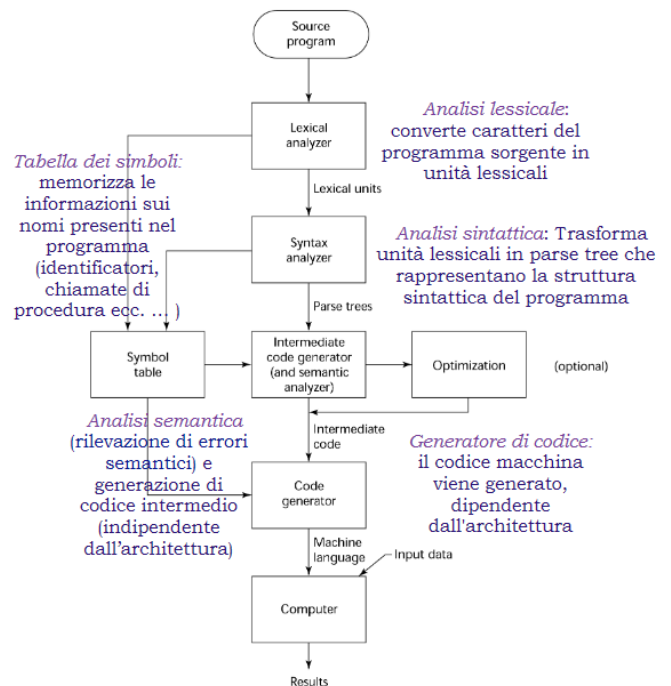
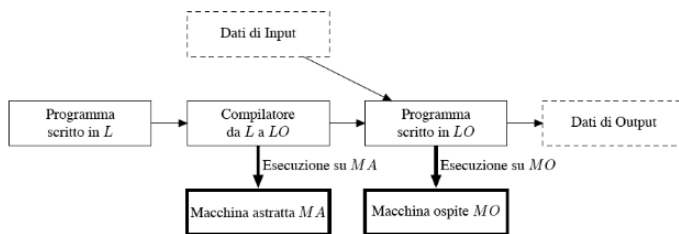
Compilatore  $C^{L \rightarrow L_0}$

$Prog^L \rightarrow Prog^{L_0} = C^{L \rightarrow L_0}$

$C^{L \rightarrow L_0}(P^L) = P^{L_0}$  tale per cui  $\forall input, P^L(input) = P^{L_0}(input)$

Dato  $P^L \in Prog^L$ ,

un compilatore  $C^{L \rightarrow L_0}$  da  $L$  a  $L_0$  è un programma tale che  $\llbracket C^{L \rightarrow L_0} \rrbracket$  da  $L$  a  $L_0$  è un programma tale che  $\llbracket C^{L \rightarrow L_0} \rrbracket : Prog^L \rightarrow Prog^{L_0}$  e  $\llbracket C^{L \rightarrow L_0} \rrbracket (P^L) = P^{L_0}$  tale che  $\forall input \llbracket P^{L_0} \rrbracket(in) = \llbracket P^L \rrbracket(in)$



Dobbiamo avere la certezza che il programma compilato faccia esattamente ciò che faceva il programma originale. Per questo un compilatore è uno strumento molto più complesso di un interprete, e si passa per varie fasi:

- **Analisi lessicale** (Scanner)
  - › Spezza un programma nei componenti sintattici primitivi, chiamati tokens. I tokens formano linguaggi regolari
- **Analisi sintattica** (parser)
  - › Crea una rappresentazione ad albero della sintassi del programma, dove ogni foglia è un token e le foglie lette da sx a dx costituiscono frasi ben formate del linguaggio. Tale albero costituisce la struttura logica del linguaggio, quindi quando non si riesce a costruire l'albero significa che quella frase è illegale e la compilazione si blocca. Le frasi di token formano linguaggi CF.

Usato in alcuni linguaggi imperativi, come C, C++

### Interprete vs compilatore

Interprete	Compilatore
<ul style="list-style-type: none"> <li>• <b>Esecuzione più lenta</b></li> <li>• <b>Maggiore portabilità e versatilità</b></li> </ul>	<ul style="list-style-type: none"> <li>• <b>Esecuzione veloce</b> (si può ottimizzare il codice in base al contesto)</li> <li>• <b>Progettazione complessa</b> (dipende dalla distanza <math>L - L_0</math>): devo anche poter dimostrare che <math>M(L) = M'(L')</math></li> </ul>

- **Debugging e interazione facilitati**

- **Debugging complesso**  
(difficile ricondurre un errore runtime alla sua causa in L0)

### Soluzioni ibride

Quelli visti sono i due estremi, ma chiaramente esistono anche soluzioni ibride. Sono un compromesso tra compilatore e interprete, dove un linguaggio ad alto livello viene compilato in un linguaggio più a basso livello, e quest'ultimo viene interpretato. Alcuni esempi sono:

- Java (Java Bytecode)
- Compilatori per linguaggi funzionali logici, tipo LISP, PROLOG, L
- Alcune vecchie implementazioni di Pascal, con Pcode

## Specializzatore

Mentre compilatore e interprete hanno come obiettivo l'esecuzione, lo specializzatore si accontenta di eseguire parzialmente.

### Formalmente

È un programma che prende in input un programma scritto in un linguaggio e restituisce un programma specializzato.

$$SPEC_L : Prog_{L'} \times Dato \rightarrow Prog_{L'}$$

$$SPEC(P, d) = P(d)$$

Quello che fa è valutare il programma su una parte dell'input, ottenendo un programma specializzato rispetto a tale input e per questo più efficiente → la parte di quell'input è "già svolta"!

$$\forall in, d \in D : \llbracket SPEC^L \rrbracket(P^L, d) = Q^L \text{ tale che } \llbracket P^L \rrbracket(d, in) = \llbracket Q^L \rrbracket(in)$$

## Proiezioni di Futamura

La proiezione di Futamura dimostra che si può esprimere un compilatore come combo di interprete e specializzatore. Il compilatore genera un nuovo programma target, che si comporta come il sorgente.

$$\begin{cases} \llbracket source \rrbracket(d) = \llbracket target \rrbracket, \text{ oppure } comp(source) = target \\ \llbracket int \rrbracket(P, d) = \llbracket P \rrbracket(d) \text{interprete} \end{cases} \rightarrow target = \llbracket spec \rrbracket(int, source)$$

## 2 – Descrivere i linguaggi

Linguaggio di programmazione: è un formalismo artificiale mediante il quale posso descrivere algoritmi.

→ Come il linguaggio naturale, ma semplificato e non ambiguo.

Context free: un termine ha lo stesso significato indipendentemente da dove si trova.

Vogliamo descrivere:

- **Sintassi /grammatica**
  - › regole di formazione: quando una frase è corretta = *relazione fra segni*
- **Semantica**
  - › Attribuzione di significato: cosa significa una frase corretta = *relazione fra segni e significato*
- **Pragmatica**
  - › In qual modo frasi corrette e sensate sono usate = *relazioni tra segni, significato e utente*
- **Implementazione**
  - › Esecuzione di una frase corretta nel rispetto della semantica.

Semantica, pragmatica e implementazione sono elementi da studiare separatamente, ma legati dalla sintassi in funzione della quale sono tutti definiti.

### Sintassi

È l'insieme delle regole che permettono di costruire frasi corrette.

Ogni frase è costituita da componenti che rappresentano le categorie sintattiche – nel linguaggio naturale sarebbero soggetto, verbo, oggetto...

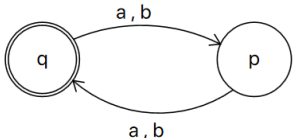
Individuato l'alfabeto, a livello lessicale si individuano le sequenze di simboli legali, ovvero quelle che costituiscono le parole del linguaggio. La fase che determina la corretta struttura delle frasi, ovvero che *verifica se una frase rispetta la grammatica*, è detta **parsing**.

Le regole sintattiche specificano quali stringhe sono legali nel linguaggio.

- **Parola**: stringa in un alfabeto
- **Frase**: sequenza (ben formata) di parole
- **Linguaggio**: insieme di frasi
- **Lessema**: parola chiave del linguaggio, che ha già un significato specifico. Corrisponde a un terminale, ed è l'unità minima sintattica. La loro sintattica è solitamente separata dalla definizione del linguaggio, ed è descritta come stringhe di linguaggi regolari.
  - › Esempi: in "index = 2" → "index", "=", "2"
- **Token**: categoria di lessemi. Per i linguaggi di programmazione sono categorie sintattiche.
  - › Ad esempio, "index = 2" è un token (assegnamento/comando)
- **Programma**: sequenza / composizione sequenziale di frasi ben formate

### Descrivere la sintassi

Per descrivere la sintassi abbiamo bisogno di strumenti formali, e in particolare vogliamo poter verificare l'appartenenza o meno al linguaggio per via automatica. Vi sono due strade:

Riconoscitori	Generatori
<p>Riconoscono se la stringa fa parte del linguaggio leggendo in input la stringa. Automi a stati finiti → macchine con stati e transizioni; se leggendo arrivano a uno stato finale, la sequenza è accettata.</p> <p><math>L = \{\sigma \in \Sigma^* \mid  \sigma  \text{ pari}\}</math>      <math>\Sigma = \{a, b\}</math></p>  <p><math>S := \varepsilon \mid aaS \mid abS \mid baS \mid bbs</math></p>	<p>Generano le stringhe di un linguaggio. Presa una stringa, cerca di generarla e se ci riesce allora essa sta nel linguaggio.</p> <p><math>L = \{\sigma \in \Sigma^* \mid \sigma \text{ palindromo}\}</math></p> <p>Def: • <math>\varepsilon</math>, <math>a</math> e <math>b</math> sono palindromi          • se <math>\sigma</math> è palindromo allora lo sono anche <math>a\sigma a</math> e <math>b\sigma b</math></p> <p><math>S := \varepsilon \mid a \mid b \mid aSa \mid bSb</math></p>

! Alcune cose possono essere riconosciute solo da generatori, come ad esempio le stringhe palindrome.

La sintassi è composta da:

- **Vocabolario** → alfabeto, ovvero l'insieme dei caratteri ammessi
- **Regole di composizione** → ci dicono come andremo a combinare i caratteri per costruire parole di senso compiuto. Le regole sono date attraverso uno strumento formale, come le **grammatiche context free**.

### Grammatiche context-free

Formalmente, le grammatiche context-free (CFG) sono una quadrupla

$$G = \langle V, T, P, S \rangle$$

Con:

- $V$  insieme finito di variabili, detti anche **simboli non terminali**. Hanno un significato variabile che deve essere ancora generato. Vanno sostituiti fino ad arrivare a una sequenza di simboli terminali.  
→ **categorie sintattiche**, aka elementi della frase
- $T$  è un insieme finito di **simboli terminali**, che non possono essere riscritti.  
→ **vocabolario**, ovvero collezione di lessemi  
  - >  $T \cap V = \emptyset$
- $P$  è un insieme finito di **produzioni**, ovvero regole.  
Ciascuna produzione ha forma  $A \rightarrow \alpha$ , con  
  - >  $A \in V$  variabile
  - >  $\alpha$  sequenza di simboli con  $\alpha \in (V \cup T)^*$
- $S \in V$  è una variabile speciale detta **simbolo iniziale**.

Usare le grammatiche CFG ha svantaggi e vantaggi:

- Vantaggi: dato un programma abbiamo strumenti efficaci di parsing; la sua complessità dipende dalla grammatica e dal grado di non determinismo delle produzioni
- Svantaggi: non catturano i vincoli contestuali; ad esempio il poter usare una variabile solo se questa è stata dichiarata/definita. Avremo bisogno di altri formalismi per descrivere questi vincoli.

## Notazione BNF

Per i linguaggi di programmazione le grammatiche CFG sono tipicamente descritte con la **forma BNF (Backus normal form)**. La BNF è un metalinguaggio usato per descrivere i linguaggi di programmazione. Dal punto di vista del significato è identico alla forma “classica” delle grammatiche.

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>I non terminali sono identificati racchiudendoli tra parentesi angolate <math>\langle \rangle</math></li> <li>Le produzioni si indicano col simbolo <math>::=</math> al posto della freccia</li> </ul> | $\langle A \rangle ::= \alpha \langle B \rangle \gamma \mid \alpha$ $\langle B \rangle ::= \varepsilon \mid \beta_1 \mid \beta_2$ |
|---|---|

Per semplificare la rappresentazione si usa anche EBNF, che aggiunge altre opzioni:

- |  |   |
|--|---|
| <ul style="list-style-type: none"> <li><math>[]</math> indica 0 o 1 occorrenze del contenuto</li> <li><math>\{\}</math> indica 0 o più occorrenze del contenuto</li> <li><math>'</math> indica più opzioni in or.</li> </ul> | $\langle A \rangle ::= \alpha [\beta_1, \beta_2] \gamma \mid \alpha \gamma \mid \alpha$ |
|--|---|

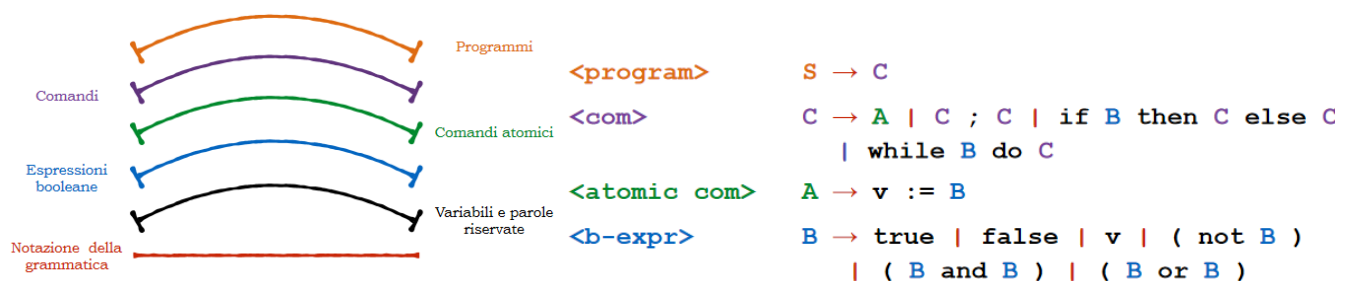
## Descrizione di un semplice linguaggio imperativo

### Grammatica

Un linguaggio può anche essere descritto in modo informale, definendo per esempio le seguenti caratteristiche:

- No dichiarazioni, solo comandi
- Solo variabili e espressioni booleane
- Assegnamento e composizione sequenziale
- Comando condizionale
- Comando iterativo

Questo può essere sufficiente per un umano, ma non basta per programmatori e interpreti. Possiamo dunque individuare dei **livelli di astrazione**.



→ Le frasi di questo linguaggio sono solo comandi; no dichiarazioni e no preamboli. Un programma è un comando.

→ Ciascuna produzione si basa sui livelli sottostanti (credo; la mastroeni scrive arabo)

→ Nelle espressioni abbiamo parentesi ovunque per evitare ambiguità, ovvero l'esistenza di più parse-tree per ciascuna operazione.

## Analisi "semantica"

Abbiamo già osservato che esistono alcuni **limiti** nei linguaggi CF, dati dal fatto che non colgono il contesto, come ad esempio:

- Numero di parametri attuali di una chiamata di procedura: bisogna controllare se è uguale al numero di parametri formali della dichiarazione
- Assegnazioni del tipo  $I:=R+3$ ; potrebbe essere illegale nel caso in cui il linguaggio fosse fortemente tipato e R o I non siano state dichiarate in precedenza
- I tipi della variabile e dell'espressione di un assegnamento devono essere compatibili.

Ergo, stringhe di per sé corrette potrebbero essere illegali in certi contesti. Si tratta di un **vincolo sintattico**, ma lo strumento sintattico scelto – aka le grammatiche CF – non permette di descriverlo. Alcune soluzioni possibili sono:

- Usare strumenti più potenti → grammatiche contestuali (**CSG**), riscrittura di un non-terminale solo in un determinato contesto
  - **! Non esistono algoritmi lineari per il riconoscimento di stringhe generate**, quindi non ci sono algoritmi efficienti di parsing. Quindi si preferisce specificare separatamente dei vincoli contestuali. → **sintassi (CFG) + semantica statica (vincoli semantici)**
- Controlli *ad hoc* → *semantica statica*

## Semantica (dinamica)

È il significato del programma, ovvero la funzione che calcola; Attribuisce un significato ad ogni frase sintatticamente corretta.

Significato: entità autonoma che esiste indipendentemente dai segni usati per descriverlo.

La semantica deve dare un significato ad ogni frase ben formata della mia grammatica. Il significato è un'entità autonoma, ovvero esiste indipendentemente dai simboli che usiamo per descriverlo.

Dobbiamo specificare gli effetti della sintassi sulla rappresentazione astratta della macchina, chiamata **stato**. Quindi, per ogni costrutto, va descritto il significato della sua esecuzione come trasformazione di stato. L'astrazione della macchina nel concetto di stato permette di dare al costrutto un significato puro, indipendente dalla macchina.

Essa deve garantire:

- **Esattezza**: descrizione precisa e non ambigua di cosa ci si debba aspettare da ciascun costrutto sintatticamente corretto, per sapere a priori quello che succederà durante l'esecuzione. (*ma è legale?*)
- **Flessibilità**: non deve anticipare scelte che possono essere demandate all'implementazione.

Non esiste un singolo formalismo universalmente accettato per dare significato ai programmi.

In generale, il significato di un programma è la composizione del significato delle categorie sintattiche e quindi delle parole che lo compongono, arrivando ad usare il significato di un vocabolario finito e noto.

→ Per dare semantica dobbiamo sempre dare significato agli elementi complessi in funzione del significato degli elementi più semplici.

Esiste quindi un legame forte fra sintassi e semantica:

- Sintassi → metodo finitario per rappresentare un insieme finito di programmi, la cui sola cosa analizzabile è la struttura
- Semantica → metodo finitario per dare significato a tutti gli elementi dell'insieme infinito dei programmi.

## Induzione matematica e induzione strutturale

L'induzione lavora su un insieme infinito di numerabili naturali.

*Dato un insieme  $A$  ed una relazione binaria  $< \subseteq A \times A$  ben fondata :*

*Se  $A = \text{Nat}$  si ha induzione matematica;*

*Se  $A = L(G)$  linguaggio generato da una grammatica  $G$  si ha induzione strutturale.*

L'induzione strutturale è leggermente diversa ma usa lo stesso principio. Serve a dimostrare le proprietà dei linguaggi generati da una sintattica; dimostro proprietà che devono valere su tutti gli elementi generati da  $S$  simbolo iniziale.

### Principio di induzione

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
2. **Ipotesi induttiva:** si dimostra la proprietà per tutti gli elementi composti, assumendo che la proprietà sia verificata da tutti i loro componenti immediati

### ESEMPIO

Dimostrare che  $\sum_{i=1}^m i = \frac{m(m+1)}{2} \quad m \geq 1$

**BASE:**  $m=1 \quad \sum_{i=1}^1 i = 1 = \frac{1(1+1)}{2} \quad \checkmark$

#### PASSO INDUTTIVO:

Supponiamo vero per  $m$   
(ipotesi induttiva  $\sum_{i=1}^m i = \frac{m(m+1)}{2}$ )

Dimostriamo per  $m+1$ :  $\sum_{i=1}^{m+1} i = \frac{(m+1)(m+2)}{2}$

$$\begin{aligned} \sum_{i=1}^{m+1} i &= \sum_{i=1}^m i + (m+1) = \frac{m(m+1)}{2} + m+1 = \\ &\quad \text{per definizione di } \Sigma \quad \text{per ipotesi induttiva} \\ &= \frac{m(m+1) + 2m+2}{2} = \frac{m^2 + m + 2m + 2}{2} = \\ &= \frac{(m+1)m + (m+1)2}{2} = \frac{(m+1)(m+2)}{2} \quad \checkmark \end{aligned}$$

## Tipi di semantica

Funzione  $\rightarrow$  Algoritmo  $\rightarrow$  Programma.

L'algoritmo è semplicemente una funzione intermedia che si frappone fra funzione e programma.

- **Comportamento I/O**  $\rightarrow$  implementatore
  - › Descrive una funzione, ma come trasformazione di stato della macchina: fissati gli stati di input quali sono gli stati di output dopo l'esecuzione del costrutto?
- **Funzione descritta dall'algoritmo**  $\rightarrow$  progettista
  - › Descrive il significato di un programma caratterizzando la funzionalità: il progettista deve progettare i costrutti del linguaggio per permettere l'implementazione di certe funzionalità
- **Proprietà invarianti**  $\rightarrow$  sviluppatore
  - › Non interessa l'esatta trasformazione di stato, ma interessa come l'uso e la combinazione dei costrutti permetta di preservare invariants o garantire proprietà desiderate

Di fatto queste diverse rappresentazioni sono equivalenti: sono solo punti di vista differenti. Questi punti di vista hanno dato origine a tipi di semantica diversi, che modellano i significati con strumenti matematici specifici (funzioni, transizioni di stato, proprietà). Possiamo raccogliere le tipologie di strumenti – e quindi di semantiche – in tre macro classi:

- **Semantica denotazionale**  $\rightarrow$  descrive **funzionalità**
  - › Studia gli effetti dell'esecuzione, cerca proprietà del programma studiando proprietà della funzione calcolata
- **Semantica assiomatica**  $\rightarrow$  descrive **proprietà**
  - › Serve per fare deduzioni logiche a partire da assiomi dati su parti del programma  $\rightarrow$  dimostrazioni di correttezza
- **Semantica operativa**  $\rightarrow$  descrive **trasformazioni di stato**
  - › Si preoccupa di COME i risultati finali vengono prodotti, permettendo l'implementazione di un interprete.

L'equivalenza delle semantiche è nota come **full abstraction**.

**Esempio:** Linguaggio L contiene solo assegnamento `:=`.  
Ogni forma di semantica descrive l'effetto/significato dell'esecuzione del programma sulla memoria.

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

?? wut



## Semantica denotazionale

**Semantica denotazionale:** modello matematico dei programmi, **basato sulla ricorsione**.  
È la semantica più astratta con cui descrivere i programmi.

Consiste nel definire un oggetto matematico per ogni entità del linguaggio, e poi nel definire una funzione che mappa istanze delle entità del linguaggio nei corrispondenti oggetti matematici.

Il linguaggio dei costrutti è definito solamente in funzione del valore delle variabili del programma, ovvero in funzione della rappresentazione dello stato.

Formalmente, il modello è quello delle **funzioni matematiche ricorsive**; ovvero un programma corrisponde a una funzione

$$E: Prog \rightarrow ( (Var \rightarrow Val) \rightarrow (Var \rightarrow Val) )$$

L'equivalenza tra programmi si dimostra come equivalenza fra funzioni.

Quindi la semantica denotazionale è un oggetto puramente matematico, che descrive gli effetti manipolando oggetti matematici  $\rightarrow$  si astrae dall'esecuzione.

Descrive gli effetti di una sequenza di comandi separati da ; attraverso la composizione dei singoli comandi, da sinistra a destra. L'effetto di ogni programma è la funzione che dato uno stato produce un nuovo stato.

È usata per dimostrare la correttezza dei programmi e fornisce un modo formale per ragionarvi, tanto che è usata anche in sistemi di generazione di compilatori. Tuttavia, a causa della sua complessità, ha poca utilità per gli utilizzatori dei linguaggi.

$E[P]\sigma$  è la **funzione di valutazione**, che valuta il programma  $P$  sulla memoria  $\sigma$  restituendo  $\sigma'$ .

$$\begin{aligned} E[P][z=\perp, y=\perp] = & (E[z:=y] \circ E[y:=y+1] \circ E[y:=z] \circ E[z:=2])[z=\perp, y=\perp] = \\ & (E[z:=y](E[y:=y+1](E[y:=z](E[z:=2][z=\perp, y=\perp]))) ) \\ & \quad \quad \quad \underbrace{\hspace{10em}}_{[z=2, y=\perp]} \\ & \quad \quad \quad \underbrace{\hspace{10em}}_{[z=2, y=2]} \\ & \quad \quad \underbrace{\hspace{10em}}_{[z=2, y=3]} \\ & \underbrace{\hspace{10em}}_{[z=3, y=3]} \end{aligned}$$

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

## Semantica assiomatica

**Semantica assiomatica:** modello matematico dei programmi, **basato sulla logica formale**  
 $\rightarrow$  calcolo dei predicati

Nasce con l'obiettivo di fare verifica formale dei programmi. Consiste in assiomi e regole di inferenza fornite per ciascun costrutto del linguaggio. Le espressioni logiche della semantica si dice **asserzione**.

- **Precondizione:** asserzione prima di un comando. Dichiara relazioni e vincoli validi prima dell'esecuzione del comando
  - > **Weakest precondition:** precondizione meno restrittiva che garantisce la post condizione.
- **Postcondizione:** asserzione che segue il comando. Descrive cosa vale dopo l'esecuzione.

Attraverso queste asserzioni, la semantica assiomatica permette di dimostrare proprietà parziali di correttezza: se lo stato iniziale rispetta la preconditione e il programma termina, allora lo stato finale soddisfa la postcondizione.

Questo tipo di semantica considera solo alcuni aspetti dell'esecuzione, ovvero quelli descritti da pre e post condizioni. Questo significa che, date esse, esistono infiniti programmi che le soddisfano e che hanno potenziali comportamenti diversi.

La semantica assiomatica deve godere di:

- Correttezza: ogni proprietà derivabile nel sistema vale per il programma
- Completezza: ogni proprietà che vale per il programma è derivabile nel sistema di regole.

Notazione:  $\{P\} \text{ statement } \{Q\}$ . Questa semantica si calcola mediante un sistema di prova: abbiamo una tripla da verificare e cerchiamo di dimostrarla applicando le regole.

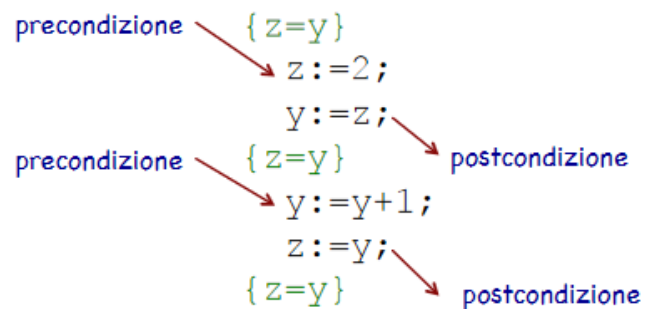
$$\frac{\{P\} S \{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\} S \{Q'\}}$$

$$\frac{\{P_1\} S_1 \{P_2\}, \{P_2\} S_2 \{P_3\}}{\{P_1\} S_1; S_2 \{P_3\}}$$

$$\frac{\{B \text{ and } P\} S_1 \{Q\}, \{\text{not } B \text{ and } P\} S_2 \{Q\}}{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \{Q\}}$$

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

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



## Semantica operativa

**Semantica operativa:** modello matematico dei programmi, **basato sui sistemi di transizione**

La semantica operativa descrive il significato dei programmi eseguendo i suoi comandi su macchina (memoria, registri...) e definisce il significato del comando.

In un linguaggio ad alto livello è necessaria per definire una macchina astratta, che esegue le sue componenti induttivamente sulla struttura del programma.

Descrive il significato del programma come trasformazioni di stato, a livello di astrazione variabile e sempre indipendente dall'architettura.

Lo stato è la rappresentazione astratta della macchina, e quindi la sua formalizzazione è ciò che decide il livello di astrazione a cui guardare il comportamento run-time dei costrutti.

La semantica operativa si occupa di come vengono calcolati i risultati finali attraverso i sistemi di transizione.

Data una funzione memoria  $\sigma: Var \rightarrow Val$  che associa valori alle variabili, lo stato è una coppia che consiste nel programma ancora da eseguire e nello stato in cui si deve eseguire il programma  $stato = \langle P, \sigma \rangle$ .

Ad ogni passo si esegue un'operazione e si cambia stato: applicando una relazione fra configurazioni.

Relazione di transizione:  $\rightarrow \subseteq \langle P, \sigma \rangle \times \langle P, \sigma \rangle$

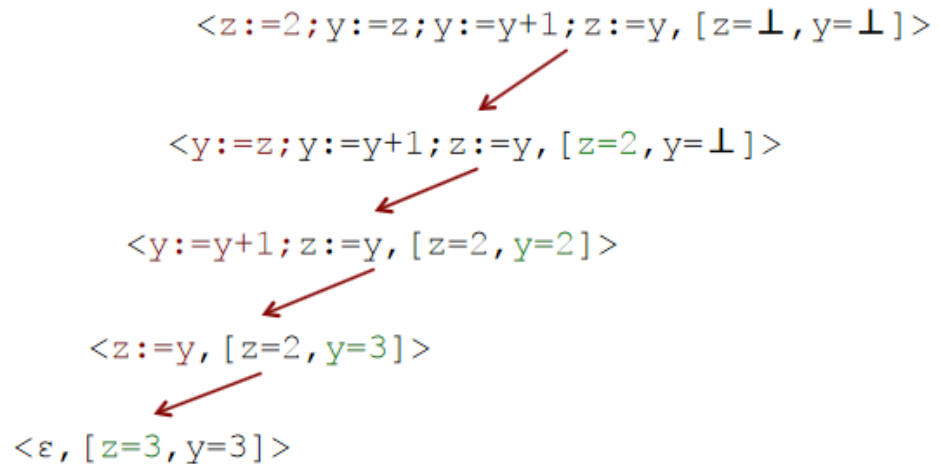
La chiusura transitiva descrive l'esecuzione completa.

Tale semantica opera eseguendo i comandi separati da ; sequenzialmente, da sinistra a destra.

Sebbene sia una delle forme più concrete di semantica, essa comunque esegue un'astrazione ed è quindi indipendente dall'architettura.

Esempio iniziale:

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



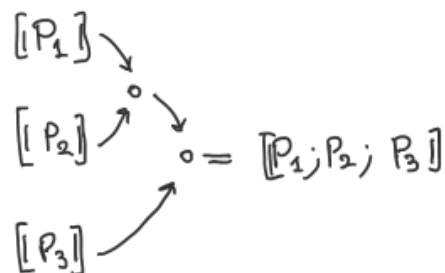
## Composizionalità

**Composizionalità:** Il significato di ogni programma deve essere funzione del significato dei costituenti immediati.

È una proprietà della semantica necessaria a caratterizzare comportamenti dei sistemi che possono avere infiniti elementi.

La semantica denotazionale e assiomatica rispettano immediatamente quest principio, mentre per l'operazionale è meno immediato.

La composizionalità diventa essenziale per garantire la modularità all'analisi dei programmi, il che permette di ragionare modularmente anche su software più grandi: infatti, se la semantica è composizionale, allora è possibile analizzare il software separatamente nei suoi moduli per poi ricomporre il risultato dell'analisi.



## Composizionalità

Due programmi sono equivalenti quando hanno la stessa semantica.

La semantica serve anche a confrontare programmi: i programmi potranno essere confrontati quando calcolano la stessa funzione; significa guardare il programma come una scatola nera, e vedere solo la relazione di input/output. Se essa è la stessa, allora indipendentemente da come essa è calcolata, la funzionalità è la stessa.

Solo quando sappiamo che due programmi calcolano la stessa funzione possiamo confrontare sull'efficienza.

L'equivalenza è necessaria in più fasi di analisi:

- Correttezza: dimostrare che il programma scritto calcola esattamente la funzione attesa
- Equivalenza di programmi: dimostrare che due programmi calcolano la stessa funzione
- Efficienza: dati due programmi che calcolano la stessa funzione, dimostrare quale lo fa in modo più efficiente.

## 2 – Sintassi e categorie sintattiche

Ci poniamo il problema di capire quali classi sintattiche deve avere un linguaggio per essere un linguaggio di programmazione. Per fare ciò classifichiamo i costrutti – che corrispondono ai non terminali della grammatica – in categorie sintattiche base.

### Stato

Lo stato è dato da ambiente e memoria

**Ambiente (environment):** insieme di legami = bindings tra identificatori e denotazioni.

L'ambiente specifica quali nomi sono usati e per quali oggetti; solitamente sono legati a un tipo, valore o locazione. Quindi specifica i nomi e i bindin ad essi associati.

Il nome, quindi, è un'entità separata dall'oggetto che denota (infatti lo stesso nome può essere usato in contesti/scope diversi e con oggetti diversi).  
→ I binding hanno un tempo di vita

**Memoria (store):** insieme degli effetti sugli identificatori (causati da assegnamenti).

La memoria è una mappa che solitamente rappresenta la storia delle variazioni dei valori associati agli identificatori – solitamente tenendo traccia solo dell'effetto dell'ultimo aggiornamento eseguito. Fornisce un binding fra locazione e valore.

I linguaggi puri / non imperativi non ne hanno bisogno, non dovendo gestire variabili che cambiano durante l'esecuzione.

### Categorie sintattiche

**Categorie sintattiche:** classificazione dei costrutti in funzione del loro significato atteso, ovvero la classe di effetti che la loro esecuzione causa.

Sono tre:

- **Generazioni di valori** → espressioni
- **Modifica/creazione di legami** → Comandi
- **Trasformazione di stato** (=memoria) → Dichiarazioni

Le categorie sintattiche necessarie per parlare di linguaggi di programmazione si distinguono in funzione di cosa denotano/producono/ottengono rispetto ad uno stato di computazione. **Le tre classi sono ben disgiunte.**

Dal punto di vista formale, le categorie sono i **simboli non terminali della grammatica** classificati in funzione di cosa e come modificano dello stato.

### Espressione

→ Generazione dei valori

- Denotano VALORI
- Devono essere **VALUTATE** per restituire un valore
- Equivalenza: due espressioni sono equivalenti se vengono valutate nello stesso valore in tutti gli stati di computazione; anche eventuali side effects devono essere gli stessi.
  - › Due espressioni possono essere DIVERSE ma essere valutate nello stesso VALORE (in tutti gli stati). Ad esempio,  $\text{not}(a \text{ and } b)$  è logicamente (semanticamente) equivalente a  $(\text{not } a) \text{ or } (\text{not } b)$  (De Morgan) ma sono sintatticamente diverse. Quindi usiamo oggetti sintattici ma ragioniamo sempre sul loro significato, che è ciò che a noi interessa
- In forma pura non modificano lo stato di computazione degli ambienti

## Dichiarazione

→ Manipolazione dei legami

- Denotano richieste di MODIFICA o CREAZIONE di LEGAMI fra nomi e valori
- Sono trasformazioni REVERSIBILI, in quanto le trasformazioni valgono solamente nel raggio d'azione corrente: posso delimitare la validità di un ambiente, quindi tutte le modifiche dell'ambiente sono reversibili nel senso che quando termina la sua validità anche le modifiche si annullano automaticamente.
- Vengono **ELABORATE**
- Equivalenza: due dichiarazioni sono equivalenti se producono lo stesso ambiente (e la stessa memoria in caso di side-effects) in tutti gli stati di computazione.
  - › Sicuramente devono generare le stesse richieste, ma soprattutto dato che possono avere molti side-effects l'equivalenza deve richiedere che si generino le stesse modifiche a tutto lo stato di computazione.
- In forma pura non producono valori e non modificano la memoria

## Comandi

→ Manipolazione dei valori

- Denotano richieste di MODIFICA della MEMORIA
- Sono trasformazioni IRREVERSIBILI poiché l'unico modo di annullarle è eseguire altri comandi.
- Devono essere **ESEGUITI** per poter attuare la corrispondente trasformazione della memoria.
- Equivalenza: due comandi sono equivalenti se per ogni stato (memoria) in input producono lo stesso stato (memoria) in output
  - › Devono rappresentare la stessa funzione di trasformazione.
  - › L'equivalenza è una proprietà universale sui possibili input
- In forma pura non producono valori e non modificano gli ambienti

Queste tre categorie non sempre sono presenti in linguaggi imperativi: ad esempio nei linguaggi funzionali abbiamo solo espressioni e dichiarazioni, poiché la computazione si basa sul concetto di applicazione delle funzioni.

## 2- Semantica operativa

Abbiamo bisogno di un framework per descrivere la semantica operativa partendo dalla grammatica e dai parse tree. Ogni costrutto, in particolare le espressioni, è dato assunto il suo abstract syntax tree – ovvero un albero senza ambiguità.

Il significato operativo consiste nel tradurre i passi computazionali in forma indipendente dall'architettura/macchina, ma che modelli l'evoluzione dello stato.

### Sistemi di transizione

Dato che la semantica va espressa in funzione della sintassi, si decide che per descrivere la semantica si userà la manipolazione di simboli.

A noi interessa sapere *cosa* fa un costrutto, non *come*; escludiamo dunque:

<ul style="list-style-type: none"><li>• Esattamente la manipolazione del linguaggio sorgente verso il codice eseguibile come avviene nel compilatore,</li><li>• Descrizione degli algoritmi corrispondenti a ogni costrutto</li></ul>	<ul style="list-style-type: none"><li>&gt; Descrivono il COME non il COSA</li><li>&gt; Non coinciso né astratto</li><li>&gt; Dipendono dall'architettura</li></ul>
<ul style="list-style-type: none"><li>• Linguaggio naturale per descrivere il comportamento</li></ul>	<ul style="list-style-type: none"><li>&gt; Problemi di ambiguità, verboso e poco accurato</li></ul>

Scegliamo quindi i **sistemi di transizione**.

**Sistema di transizione:** un sistema di transizione è una struttura  $(\Gamma, \rightarrow)$  dove  $\Gamma$  è un insieme di elementi  $\gamma$  chiamati configurazione 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.

Essi sono:

- Matematicamente precisi
- Molto concisi
- Metodo di specifica generale che permette astrazione
- Espressi mediante una collezione di regole date induttivamente in funzione della sintassi: specificano cosa viene calcolato per infusione sulla struttura sintattica del linguaggio.
- Molto generale e funziona per qualunque architettura
- Astrae i dettagli a basso livello

Le configurazioni dipendono dal tipo di linguaggio. Quando due configurazioni sono nella relazione di transizione, diciamo che la prima **si può muovere** nella seconda. Il sistema di transizione genera un multigrafo diretto infinito, dove le configurazioni sono nodi, mentre le transizioni sono archi diretti.

L'esecuzione di un programma, come la sintassi, si può rappresentare con un grafo dove i nodi sono le configurazioni.

*Notazione:*

$$\gamma \rightarrow \gamma' \text{ per } \langle \gamma, \gamma' \rangle \in \rightarrow$$

$$\gamma_0 \rightarrow^* \gamma_n \Leftrightarrow \exists \gamma_1 \dots \gamma_n \cdot \gamma_i \rightarrow \gamma_{i+1}, i \in [0, n]$$

Una derivazione è una sequenza di transizioni.

Il sistema di transizioni definisce come le derivazioni avvengono, ovvero come si possono raggiungere le configurazioni finali. Quindi il sistema di transizione è il framework di base, e il modo in cui viene definito dipende da quello che si vuole descrivere.

Queste regole funzionano bene perché:

- Livello di dettaglio variabile
- Definisce la semantica in modo induttivo
- Dicono cosa si può e cosa non si può inferire: quello che non si può si deriva semplicemente dall'impossibilità di derivarlo.

*Esempio:*

$$G = \langle V, T, P, S \rangle, \quad \Gamma = (V \cup T)^*$$

$$A \rightarrow \alpha \quad \Rightarrow \quad \frac{A \rightarrow \alpha}{\beta A \gamma \rightarrow \beta \alpha \gamma}$$



### Grammatica

```

<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 )

```

```

<Expr>     E → [+,-] T [A T]
<Add op>   A → + | -
<Terms>    T → F [M F]
            M → * | /
<Factor>   F → I | N | ( E )
<Numbers>  N → [+,-] d{d}
<Digit>    d → 0 | ... | 9
<Ident>    I → l{l,d}
<Letter>   l → A | ... | Z

```

- Linguaggio Pascal-like
- Singolo tipo di dato → INTEGER
- Strutture di controllo standard
- Astrazione del controllo, ovvero procedure senza parametri

### Sintassi

- I programmi P sono blocchi
- I blocchi B sono una dichiarazione D seguita da un comando S.
  - › D e S sono definiti in modo libero dal contesto, ovvero quello che possiamo generare da S non ha alcun vincolo con D → Non siamo obbligati ad usare identificatori precedentemente definiti! Quindi, dalla descrizione sintattica rimangono esclusi i vincoli sintattici / context sensitive. Tutto come previsto: lo vedremo nella semantica statica.
- Le dichiarazioni D sono dichiarazioni di valori costanti con nome, dichiarazioni di variabili e dichiarazione di procedure.
- I nomi sono gli identificatori I
- N sono i numeri naturali
- Le produzioni di D sono tutte opzionali: posso avere un programma che è un blocco senza dichiarazioni
- I comandi sono:
  - › Comando vuoto
  - › Assegnamento di un valore (espressione) ad un identificatore

- › Comando condizionale che esegue al verificarsi della condizione booleana C
  - › Ciclo al verificarsi della condizione booleana C
  - › Composizione sequenziale di comandi
- Le condizioni C sono espressioni booleane
  - › Odd è predicato unitario che verifica se l'espressione è 0
- Le espressioni E sono identificatori, valori naturali, operazioni tra espressioni e espressioni tra parentesi
- Sintetizziamo in **op** la metavariable degli operatori.
- ! Per la grammatica delle espressioni: risolviamo le ambiguità via parentesi
  - › No parentesi → ambiguo, tutte le parentesi → pesante e prone a errori.

Per il compilatore, la grammatica deve essere completa, quindi dobbiamo specificare tutti gli elementi. L'identificatore è una stringa che deve necessariamente iniziare con un carattere non numerico. Qui ci sono solo i caratteri maiuscoli, ma è molto facile estendere. È da notare che questo ultimo gruppo di grammatiche (N e I) non sono parte del parsing ma dell'analisi lessicale (scanning), essendo linguaggi regolari, per questo non sono tipicamente considerati parte della grammatica principale. Ricordiamo che lo scanning prende il programma e lo divide in parole/lessemi che descrivono entità del programma (parole, identificatori, costanti...) e poi lo si guarda come un singolo elemento. Questo è usato anche per raccogliere le informazioni sui tipi, la fase di scan quindi fa l'operazione di convertire una sequenza di caratteri in una sequenza di lessemi, ovvero di elementi corredati dal "tipo". Questo significa che, se ci mettiamo al livello della grammatica dei comandi, allora possiamo trattare semplicemente identificatori e numeri come insiemi noti  $N = \text{INTEGERS}$ ,  $I = \{\text{id, rate, ciao2, ...}\}$ . Questo è un esempio di come la potenza delle CFG permetta di modificare la grammatica, ottenendo nuove grammatiche equivalenti ma più adatte magari al processo di elaborazione per il quale vengono definite (compilazione). Quindi le CFG permettono di descrivere i linguaggi ad un

profondo livello di dettaglio che non è necessario conoscere per utilizzare il linguaggio. Solitamente l'utente deve conoscere la sintassi astratta del linguaggio. Quindi i punti di vista dell'implementatore e dell'utente sono diversi: è necessaria una grammatica abbastanza grossolana, magari con ambiguità, per l'utente che è interessato solo all'abstract syntax tree, dove serve sapere se scrivendo in un modo o in un altro ci sono differenze **\*\*semantiche\*\***, ma si necessita una grammatica abbastanza fine per evitare problemi di ambiguità ad esempio nell'implementazione di un parser.

## Concetti necessari

### Identificatori

Sequenza di caratteri usata per rappresentare o denotare un altro oggetto.

Sono sequenze di caratteri che hanno lo scopo di denotare *qualcos'altro*, e dunque sono nomi usati per riferire altri oggetti – sia elementi del linguaggio (come procedure) che della macchina (celle di memoria), senza conoscerne il valore a priori.

Quando identificano celle di memoria, gli identificatori sono anche detti variabili.

L'uso di identificatori è fondamentale perché:

- Più facile ricordare i nomi
- Contribuiscono ad astrarre i dati e il loro controllo → Non devo usare gli indirizzi assoluti

**! Oggetto denotato e identificatore sono due cose diverse: un identificatore può denotare più elementi e un elemento può essere denotato da più identificatori diversi ( *aliasing* )**

Non possono essere stringhe di caratteri qualsiasi; ci vogliono regole affinché siano riconoscibili dal parser. Quesiti da risolvere:

- Case sensitive o no
- Parole riservate
- Vincoli sulla lunghezza
- Caratteri speciali necessari (tipo PHP che vuole \$)

### Bindings – legami

Il concetto di binding viene dalla matematica, dove è espresso con fasi tipo “sia x...”, oppure in logica con i quantificatori universale ed esistenziale.

In programmazione, i binding sono creati dalle dichiarazioni. Prima di poter usare un nome per denotare un oggetto complesso, è necessario creare il legame: dichiaro il nome e definisco il suo significato.

Nei linguaggi di programmazione, questi concetti prendono una connotazione più specifica.

In generale, è il PL che definisce sintatticamente e semanticamente cosa è definizione e cosa è uso di un identificatore; inoltre ci sono PL in cui un programma completo non sono ammesse occorrenze libere di identificatori.

<b>Binding occurrences / creazione di binding</b> $\left( \sum_{i=1}^n \left( \sum_{j=1}^m a_{ij} \dots b_{jk} \right) \right)$	Definizione di un nome-identificatore e del suo significato-denotazione.	$\left( \sum_{i=1}^n \left( \sum_{j=1}^m a_{ij} \dots b_{(k)} \right) \right)$ <b>Occorrenza libera</b>
<b>Applied occurrences / occorrenze di applicazione</b> $\left( \sum_{i=1}^n \left( \sum_{j=1}^m a_{ij} \dots b_{jk} \right) \right)$	Si usa il nome-identificatore per accedere al significato-denotazione.	Uso un nome il cui significato non è stato definito / uso un identificatore la cui denotazione non è stata definita. In generale un nome non dichiarato è un nome che può rappresentare qualunque cosa in quel punto, e quindi una sua occorrenza può essere legata a qualunque oggetto. In realtà le occorrenze libere ed applicate sono identiche: cambia solo se sono nel raggio di azione (scope) di un'occorrenza di definizione.
<b>Scope di un binding / raggio di definizione di un binding</b>	Definisce lo spazio in cui si può usare un nome per rappresentare il significato associato da un binding	

## Tipi di bindings

A seconda dell'oggetto denotato, il tipo di binding è diverso:

→ <b>Binding statico</b> : occorre per la prima volta prima dell'esecuzione del programma e rimane invariato durante tutta l'esecuzione.	<i>Binding nome-valore</i>  Esempi: tipi delle variabili nei linguaggi fortemente tipati
→ <b>Binding dinamico</b> : occorre durante l'esecuzione e può variare. Esempi: locazioni e valori contenuti in esse.	<i>Binding Nome-Locazione</i> → legame non modificabile fissato dalla definizione <i>Binding Locazione valore</i> → legame modificabile dall'esecuzione

## Tempi di binding

La creazione del binding può avvenire in tempi diversi:

<i>Tempo di compilazione</i> (early binding)	<i>Tempo di esecuzione</i> (Late binding)
Per ogni nome tutto viene risolto a tempo di compilazione, quindi abbiamo allocazione statica con indirizzi assoluti. <ul style="list-style-type: none"><li>• Esecuzione veloce</li><li>• Non flessibile</li><li>• Uso dello stack: allocazione statica</li></ul>	Per ogni nome la memoria viene allocata dinamicamente durante l'esecuzione, e tutti i binding sono creati dinamicamente con indirizzi non assoluti. <ul style="list-style-type: none"><li>• Esecuzione lenta</li><li>• Flessibile</li><li>• Uso dello heap</li></ul>
Fortran, cobol, C	SNOBOL4, LISP

Diversi linguaggi stanno nel mezzo, allocando dinamicamente solo se serve → C

Altri esempi di tempi di bindings:

- **Tempo di progettazione** → operatori e simboli
- **Tempo di implementazione di linguaggi** → tipo floating point, con la propria rappresentazione
- **Tempo di compilazione** → binding delle variabili con il loro tipo per C o Java
- **Tempo di caricamento** → binding delle variabili statiche (C) alla cella di memoria
- **Tempo di esecuzione** → binding di variabile non statiche alla cella di memoria

La questione è pragmatica e semantica.

**!!! Un'espressione non ha significato se contiene nomi non legati a nulla.**

## Identificatori liberi

Data un'espressione o una frase scritta in un certo linguaggio, gli identificatori liberi sono quelli che non hanno alcuna dichiarazione che li coinvolge. Questo significa che la frase è parte di una frase più grande, dove gli identificatori liberi sono legati e trovano significato: quindi sono identificatori liberi in una espressione tutti quelli che hanno solo occorrenze libere e non di applicazione – ovvero occorrenze che non sono nel raggio di azione di una occorrenza di binding.

Dentro un'espressione, la presenza di un identificatore libero richiede l'accesso a un ambiente esterno che gli dia significato → non possiamo dare significato ad un'espressione con identificatori liberi.

## Ambienti in IMP

Nel nostro semplice linguaggio imperativo consideriamo solo interi e booleani, che saranno i nostri valori denotabili  $Dval = Int \cup Bool$

$\perp$  non rappresenta alcun valore, dunque va associato all'identificatore non definito.

**Definizione** (ambiente). Un ambiente dinamico è un elemento dello spazio di funzioni

$$Env = \cup_{V \subseteq Id} Env_V$$

dove  $Env_V : V \rightarrow DVal \cup \{\perp\}$  ha metavariable  $\rho$ .

Ovvero, un ambiente per un insieme di identificatori finito  $V (=Env_V)$  è una funzione che ad ogni identificatore associa un valore denotabile, oppure il valore indefinito  $\perp$ .

L'ambiente è l'unione disgiunta di tutte queste funzioni al variare di  $V$

## Tipi

Il tipo determina **l'insieme dei valori** (che condividono una certa proprietà strutturale) **che un identificatore può denotare**, e **l'insieme di operazioni** definite su quei valori. Può denotare anche altre informazioni (es. la precisione per i floating point).

Oltre a int e bool definiamo una nuova famiglia di tipi, che ci permetta di distinguere identificatori variabili e costanti.  $\tau_{loc}$ : identificatori variabili di tipo  $\tau$

### Elenco dei tipi:

- $\tau \in DTyp = \{int, bool, proc, \perp\}$
- $\tau_{loc} \in DTypLoc = \{intloc, boolloc\}$
- $Typ = DTyp \cup DTypLoc$

Per elaborare una dichiarazione dobbiamo costruire l'ambiente statico corrispondente alla dichiarazione. Dobbiamo quindi introdurre un nuovo tipo, che ci permette di identificare gli identificatori che si riferiscono a procedure.

Non sempre la distinzione fra cosa è un tipo e cosa no è così netta, e dipende dal linguaggio.

I tipi sono utili per motivi diversi:

- **Livello di progetto:** organizzano l'informazione  
→ Tipi diversi per concetti diversi; sono come commenti sull'uso inteso degli identificatori
- **Livello di programma:** identificano e prevengono errori  
→ Dato che sono controllabili automaticamente, fanno un controllo dimensionale.
- **Livello di implementazione:** permettono alcune ottimizzazioni  
→ Bool richiede meno bit di real, e pre-calcolo degli offset per accesso alle struct.

## Type binding

Il tipo è ora un nuovo oggetto denotabile, e quindi associabile agli identificatori. In generale non si associa il tipo all'identificatore per essere riferito come oggetto, ma per descriverne proprietà che permettono di descrivere vincoli semantici non descrivibili tramite la grammatica (ovvero i vincoli contestuali).

Quando avviene il legame?

<i>Legame statico :</i>	<i>Legame dinamico:</i>
Una volta creato rimane inalterato per tutta l'esecuzione.	il legame può cambiare durante l'esecuzione; si specifica tramite un comando di assegnamento.
Se il legame è statico, può essere specificato sia con una dichiarazione implicita che esplicita. Esplicita: quando esiste un comando del linguaggio che permette di dichiarare il tipo delle variabili Implicita: meccanismo di default che specifica il tipo delle variabili attraverso convenzioni di default Vantaggio: facilità di scrittura Svantaggio: scarsa affidabilità	<ul style="list-style-type: none"><li>• Vantaggio: flessibilità</li><li>• Svantaggi:<ul style="list-style-type: none"><li>○ Costo alto di implementazione</li><li>○ Difficile rilevazione dei valori di tipo</li></ul></li></ul> Può avvenire solo nei linguaggi interpretati.

I computer non hanno istruzioni i cui tipi degli operandi non sono noti, quindi un compilatore non può costruire un'intera istruzione macchina se i tipi non sono noti.

Un interprete puro per fare questa operazione però impiega 10 volte più tempo, perché deve anche risolvere il binding dei tipi (ma questo tempo viene nascosto dal tempo di compilazione ?????????).

Viceversa, raramente i linguaggi con binding statico sono interpretati, dato che la compilazione è più efficiente.

## Necessità di una semantica statica

Dobbiamo definire uno strumento formale per creare legami di tipo e verificare che i vincoli contestuali siano verificati. Lo facciamo attraverso la semantica statica, che permette di

- Essociare un tipo ad ogni identificatore, e in particolare a ogni espressione ben formata
- Verificare se dichiarazioni e comandi sono scritti correttamente

Il legame di tipo non cambia mai per lo stesso identificatore, dato che consideriamo linguaggi con tipaggio statico, mentre i legami con i valori cambiano durante l'esecuzione. Per questo ci serve uno strumento diverso.

### *Semantica dinamica:*

La semantica per la valutazione delle espressioni, l'elaborazione delle dichiarazioni e l'esecuzione dei comandi.

- Permette di creare e modificare i legami con gli oggetti denotati. Questi legami sono quelli che formano **l'ambiente dinamico**, il quale è creato e modificato durante l'elaborazione delle dichiarazioni e l'esecuzione dei comandi

### *Semantica statica:*

La semantica per la creazione dei legami di tipo e per la verifica della corretta forma degli elementi. Nella sua verifica della forma permette anche di verificare i vincoli contestuali

- I legami di tipo sono raccolti **nell'ambiente statico**, creato esclusivamente dalla semantica statica delle dichiarazioni e che non può essere modificato durante l'esecuzione.

## Ambiente statico e semantica statica

L'ambiente statico associa agli identificatori il tipo degli oggetti che denoteranno.

**Definizione** (ambiente statico). Un ambiente statico (o di tipi) è un elemento dello spazio di funzioni  $TEnv$  definito da

$$TEnv = \cup_{V \subseteq_f Id} TEnv_V$$

dove  $TEnv_V : V \rightarrow DTyp$  ha metavariable  $\Delta$  e  $DTyp$  è l'insieme dei tipi denotabili.

Usiamo la notazione  $\Delta \vdash_V$  quando vogliamo esplicitare il dominio  $V$  degli identificatori a cui l'ambiente dinamico associa il tipo di una derivazione.

Nel nostro linguaggio i  $DTyp$  sono, per ora, solo interi e booleani, più un tipo speciale che rappresenta il tipo di un identificatore non inizializzato.

$$\tau \in Dtyp = \{int, bool, \perp\}$$

A questo punto possiamo definire la semantica statica come un insieme di regole che permettono di associare un tipo ad ogni espressione corretta.

Le regole della semantica statica hanno la forma

$\mathcal{E}_{S_i} : \Delta \vdash_V e : \tau$

Dove:

- $\Delta$  indica l'ambiente statico nel quale valutare semanticamente l'espressione, ovvero l'insieme dei legami statici validi nel contesto di valutazione
- $V$  indica l'insieme degli identificatori per i quali l'ambiente definisce un'associazione. Se questi elementi non ci sono, allora valutiamo nell'ambiente vuoto; In tal caso, diciamo che  $e$  è di tipo  $\tau$  nell'ambiente  $\Delta$ .

## Compatibilità di ambienti

Possiamo aspettarci che i due ambienti definiti sullo stesso programma parlino in modo coerente degli stessi identificatori. Definiamo quindi la compatibilità fra ambienti:

**Definizione** (compatibilità di ambienti). Sia  $\rho : V$  un ambiente dinamico e  $\Delta : V$  un ambiente statico con  $V \subseteq_f Id$ . Gli ambienti  $\rho$  e  $\Delta$  sono compatibili (scritto  $\rho : \Delta$ ) se e soltanto se

$$\forall id \in V. (\Delta(id) = \tau \wedge \rho(id) \in \tau)$$

Ovvero, un ambiente statico e un ambiente dinamico sono compatibili se parlano in modo coerente degli stessi identificatori. Ad esempio, questo significa che se l'ambiente statico stabilisce che una certa espressione ha tipo  $\tau$ , allora la semantica dinamica deve arrivare ad associare all'identificatore un valore contenuto nel tipo  $\tau$ , ovvero nell'insieme dei valori con lo stesso tipo  $\tau$ .



Di conseguenza, nelle regole anziché guardare l'ambiente dinamico specificando l'insieme di identificatori, possiamo specificare semplicemente l'ambiente statico compatibile:  $\rho \vdash_{\Delta}$

L'insieme degli identificatori  $V$  rimane implicito dentro l'ambiente statico.

## Aggiornamento di ambienti

Consideriamo due ambienti  $\beta, \beta' \in Env$  dove  $\beta : V, \beta : V' (V, V' \subseteq Id)$ .

L'aggiornamento dell'ambiente  $\beta$  mediante l'aggiornamento  $\beta'$  è l'ambiente  $\beta' \in Env$ , denotato  $\beta[\beta']$  e definito come

In altre parole, l'ambiente che aggiorna ha la precedenza – tutto quello di cui questo ambiente non parla rimane inalterato.

$$\beta''(I) = \begin{cases} \beta'(I) & \text{se } I \in V' \\ \beta(I) & \text{otherwise} \end{cases}$$



## 3 – Espressioni

Le espressioni sono oggetti sintattici usati per ragionare sui valori, composto da operatori, operanti, parentesi e potenzialmente chiamate a funzioni/procedure.

Le espressioni devono essere VALUTATE, e il valore ottenuto dalla valutazione è detto **valore esprimibile** in quanto è il valore che può essere espresso attraverso la sintassi del PL.

In generale, in un linguaggio come PL0, abbiamo espressioni che rappresentano interi e identificatori, e indirettamente booleani in quanto le condizioni corrispondono a valori booleani.

Dal punto di vista sintattico, i costituenti elementari delle espressioni sono i letterali: costanti e identificatori. Essi sono composti tramite operatori, sia aritmetici che booleani.


### Caratterizzazione

Le caratteristiche delle espressioni sono:

- Notazione: specifica in che modo gli operatori sono rappresentati e indica su quali operandi opera un operatore.
- Regole di precedenza e associatività tra gli operatori
- Ordine di valutazione degli operandi
- Presenza di side-effects, ovvero un qualunque effetto che non sia la pura rappresentazione di valori
- Overloading degli operatori, ovvero simboli di operatore che hanno significato diverso a seconda del tipo
- Espressioni con tipi misti

### Notazione

Le espressioni possono essere denotate in diversi modi:

- **Notazione in-fissa:**  $a + b$   
È la più usata in matematica e dunque nei linguaggi, poiché più chiara, ma soggetta a grande ambiguità e necessita di regole di associatività e precedenza.
  - **Regole di associatività:** definiscono l'ordine con cui gli operatori allo stesso livello sono valutati
  - **Regole di precedenza:** definiscono l'ordine in cui operatori adiacenti a livelli diversi di precedenza vengono valutati.
- **Notazione pre-fissa (polacca):**  $+ a b$    
È semplice poiché non necessita di regole di precedenza né associatività e rappresenta in modo uniforme operatori di qualunque arietà.
- **Notazione post-fissa (polacca inversa):**  $a b +$   
È semplice poiché non necessita di regole di precedenza né associatività e rappresenta in modo uniforme operatori di qualunque arietà.

#### ALGORITMO DI VALUTAZIONE:

1. Leggi il prossimo simbolo dell'espressione e metti nella pila
2. Se il simbolo è un operatore  $op$ :  $C_{op} = n$  e torna a 1.
3. Se il simbolo letto è un operando inseriscilo nella pila e calcola  $C_{op} = C_{op} - 1$  (op ultimo operatore inserito)
4. Se  $C_{op} \neq 0$  torna a 1 (op ultimo operatore inserito).
5. Se  $C_{op} = 0$  allora
  - Applica l'ultimo operatore  $op$  agli operandi in cima alla pila (che vanno cancellati) e carica il risultato nella pila e se non ci sono più simboli vai a 6.
  - $C_{op} = n - m$  ( $n$  arietà del nuovo ultimo operatore  $op$  nella pila,  $m$  numero di operandi sopra l'operatore nella pila)
  - Torna a 4.
6. Se la pila non è vuota torna a 1.

### Regole di associatività

Definiscono l'ordine con cui gli operatori allo stesso livello di precedenza vengono valutati.

## Valutazione delle espressioni

L'implementazione di un linguaggio di programmazione deve ovviamente rappresentare internamente le espressioni in modo da poterle manipolare.

Tipicamente la rappresentazione interna delle espressioni consiste in una rappresentazione ad albero – abstract syntax tree. Ogni nodo interno è un operatore, e i suoi figli sono alberi che rappresentano le espressioni a cui l'operatore si applica. → Si eliminano eventuali ambiguità associate a precedenza e associatività, ma non può dare informazioni riguardanti l'ordine di valutazione dell'espressione – che invece è determinata dalla tipologia di visita che l'implementazione del linguaggio effettua sull'albero.

Il problema dell'ordine di valutazione non è matematico (tutte le operazioni sono commutative), ma informatico: ci sono vari aspetti che potrebbero influire sul risultato, quali:

- **Operatori non definiti:** potrebbe succedere che l'espressione risultante venga valutata con un preciso ordine. Ad esempio, per “ $a=0 ? b : b/a$ ”,  $b/a$  potrebbe essere una divisione su zero, ma valutata solo se  $a \neq 0$ . Quindi, in realtà, valutando le operazioni solo quando necessario (lazy) questa operazione è sempre definita.  
Nel caso di operatori booleani, la valutazione lazy si dice corto circuito.
- **Effetti collaterali:** sono effetti legati non all'oggetto che si sta manipolando – per esempio, se la valutazione di un'espressione causa anche un cambiamento di memoria.  
Si parla di side effects anche quando una funzione modifica i propri parametri, o comunque variabili non locali.
- **Aritmetica finita:** nelle macchine l'aritmetica è limitata dall'architettura, quindi i numeri non sono infiniti e c'è un numero massimo rappresentabile. Il problema di valutazione si ha quando le espressioni coinvolgono questo numero – ad esempio, somma prima di una differenza con una somma che calcola un numero maggiore del massimo numero rappresentabile.

Tipicamente l'ordine di valutazione risolve prima le variabili, poi le costanti e poi valuta gli operandi seguendo la struttura data dalle parentesi.

## Valutazione ed equivalenza

Le regole della semantica dinamica, ovvero del sistema di transizione, descrivono formalmente come vengono valutate le espressioni del nostro linguaggio.

**Valutazione delle espressioni con variabili:** la valutazione è una funzione  $Eval : \mathcal{E}^V \times Mem \rightarrow \mathbb{N} \cup \mathbb{B} \times Mem$  che descrive il comportamento dinamico delle espressioni restituendo il valore in cui esse sono valutate.

$$Eval(\langle e, \sigma \rangle) = k \Leftrightarrow \langle e, \sigma \rangle \rightarrow^* \langle k, \sigma \rangle$$

**Equivalenza di espressioni con variabili:** l'equivalenza di espressioni è una relazione  $\equiv \subseteq \mathcal{E}^V \times \mathcal{E}^V$  tale che

$$e_0 \equiv e_1 \Leftrightarrow \forall \sigma, Eval(\langle e_0, \sigma \rangle) = Eval(\langle e_1, \sigma \rangle)$$

Ne deriva, per esempio, che  $(3+5)*2 \equiv (1+3)*4$ , sebbene siano espressioni sintatticamente molto diverse

## Funzioni FI e DI in $\mathcal{E}$

### Identificatori liberi FI

**Definizione** (identificatori liberi). La funzione  $FI : Exp \rightarrow Id$ , che ad ogni espressione associa l'insieme degli identificatori liberi in essa contenuti, è definita per induzione da

$$\begin{aligned} FI(k) &= \emptyset \\ FI(id) &= \{id\} \\ FI(e_0 \text{ bop } e_1) &= FI(e_0) \cup FI(e_1) \\ FI(uop\ e) &= FI(e) \end{aligned}$$

### Identificatori definiti DI

Possiamo definire allo stesso modo anche gli identificatori legati DI, per induzione sulla struttura sintattica delle espressioni o di una categoria sintattica. Nel nostro caso la definizione è molto semplice poiché nelle espressioni non ci sono costrutti che creano legami, ma solo costrutti che le utilizzano: quindi l'insieme delle occorrenze legate/definite è vuoto.

Si noti che  $FI(e) \cup DI(e) = Id(e)$ . Questo concetto esiste in tutti i PL.

## $\mathcal{E}$ in IMP: grammatica

Per parlare di semantica guardiamo alla struttura induttiva, e ignoriamo gli aspetti puramente lessicali: ad esempio, con  $E + E$  rappresentiamo un albero sintattico con radice  $+$  e i cui sottoalberi sono espressioni, generate da  $E$ .

Le espressioni di IMP sono:

```
<Exp>
E → A | B
A → I | N | A op A
B → I | true | false | not B | B bop B
```

- **A**: espressioni aritmetiche
- **op**:  $\{+, -, *, /\}$
- **B**: espressioni booleane
- **bop**:  $\{or, \wedge\}$
- **I**: identificatori

## Semantica statica delle espressioni

Gli ingredienti sono:

	Metavariabile	Definizione
Espressioni $\mathcal{E}$	e	Insieme di espressioni (=alberi di espressioni) valutate ad intero o booleano. Deprecato: ci serve considerare gli identificatori!
Espressioni valutate $\mathcal{E}^v$	e	Insieme delle espressioni valutate, ovvero le espressioni valutate ad intero o booleano: costituiscono l'insieme delle nuove configurazioni del sistema di transizione.
Numeri $\mathcal{N}$	m,n,p	Insieme di naturali nella macchina sottostante.
Booleani $\mathcal{B}$	t	insieme di valori booleani {true, false}.

Usiamo infine questi insiemi per definire il sistema di transizione, le cui regole forniscono la semantica operativa delle espressioni di cui abbiamo dato la grammatica.

- Definiamo l'insieme delle configurazioni  $I'$  come l'insieme delle espressioni da valutare  $\mathcal{E}$ .
- L'insieme delle configurazioni terminali  $T$  sono i valori numerici e booleani.

**SISTEMA DI TRANSIZIONE:**  
 $I^v = \mathcal{E}^v \times \text{Mem}, T^v = (\mathcal{N} \cup \mathcal{B}) \times \text{Mem},$

Individuiamo in grassetto rosso il simbolo sintattico, e l'operazione nella macchina sottostante in normale. Di questo non ci interessa l'implementazione.

Tutte le regole precedentemente definite devono essere scritte integrando l'ambiente nella regola, poiché le espressioni con identificatori devono essere valutate dentro un ambiente. Infine, vogliamo aggiornarle anche con l'ambiente statico

Quindi, tutte le regole  $\mathcal{E}_i$  diventano del tipo

Dove:

$$\mathcal{E}_i: \rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle$$

- $\rho$  è l'ambiente di valutazione delle espressioni, ovvero la specifica dell'associazione fra identificatori e denotazioni.
- $\rightarrow_e$  indica quale sistema di transizione si sta usando, dato che ne avremo uno per ogni categoria sintattica.
- $\rho \vdash_{\Delta}$  è l'ambiente statico.

## REGOLE $\mathcal{E}_i$ : semantica statica delle espressioni

<div><math>\mathcal{E}s_1: \vdash n:\text{int}</math></div> <div><math>\mathcal{E}s_2: \vdash t:\text{bool}</math></div>	<p><b>Assioma tipi base:</b></p> <p>Nell'ambiente vuoto (e quindi in ogni ambiente possibile), un numero intero ha tipo int e una costante booleana ha tipo booleano.</p>																																	
<div><math>\mathcal{E}s_3: \Delta \vdash_v I:\tau \quad \text{se } \Delta(I) \in \{\tau, \tau_{\text{loc}}\}, I \in V</math></div>	<p><b>Assioma identificatori:</b></p> <p>L'identificatore ha come tipo quello dell'ambiente statico <math>\Delta</math> del contesto che gli si associa.</p>																																	
<div><div><math display="block">\mathcal{E}s_4: \frac{\Delta \vdash_v e_1:\tau_1 \quad \Delta \vdash_v e_2:\tau_2}{\Delta \vdash_v e_1 \text{ bop } e_2: \tau_{\text{bop}}(\tau_1, \tau_2)}</math></div><div><math display="block">\mathcal{E}s_5: \frac{\Delta \vdash_v e_1:\tau_1 \quad \Delta \vdash_v e_2:\tau_2}{\Delta \vdash_v e_1 \text{ op } e_2: \tau_{\text{op}}(\tau_1, \tau_2)}</math></div></div>	<p><b>Induttiva bop e op:</b></p> <p>Usiamo delle funzioni che determinano il tipo del risultato di un operatore booleano o aritmetico in funzione del tipo degli operandi.</p> <div><table><tr><td>+</td><td>-</td><td>*</td><td>int</td><td>bool</td></tr><tr><td>int</td><td>int</td><td>int</td><td>int</td><td>bool</td></tr><tr><td>bool</td><td>bool</td><td>bool</td><td>bool</td><td>bool</td></tr></table><table><tr><td>=</td><td>int</td><td>bool</td></tr><tr><td>int</td><td>bool</td><td>bool</td></tr><tr><td>bool</td><td>bool</td><td>bool</td></tr></table><table><tr><td>or</td><td>int</td><td>bool</td></tr><tr><td>int</td><td>bool</td><td>bool</td></tr><tr><td>bool</td><td>bool</td><td>bool</td></tr></table></div>	+	-	*	int	bool	int	int	int	int	bool	bool	bool	bool	bool	bool	=	int	bool	int	bool	bool	bool	bool	bool	or	int	bool	int	bool	bool	bool	bool	bool
+	-	*	int	bool																														
int	int	int	int	bool																														
bool	bool	bool	bool	bool																														
=	int	bool																																
int	bool	bool																																
bool	bool	bool																																
or	int	bool																																
int	bool	bool																																
bool	bool	bool																																
<div><math display="block">\mathcal{E}s_6: \frac{\Delta \vdash_v e_0:\text{bool}}{\Delta \vdash_v \text{not } e_0: \text{bool}}</math></div>	<p><b>Induttiva not</b></p>																																	

## Semantica dinamica delle espressioni

### REGOLE $\mathcal{E}_i$ : semantica dinamica delle espressioni

L'elenco è il seguente:

$\mathcal{E}_1: \rho \vdash_{\Delta} \langle m \text{ op } n, \sigma \rangle \rightarrow_e \langle k, \sigma \rangle$ <p>se <math>m \text{ op } n = k, m, n \in \mathcal{N} \quad k \in \mathcal{N} \cup \mathcal{B}</math></p>	<b>Assoima op:</b> Assioma che valuta un'espressione sintattica con operatore aritmetico nel valore che esso rappresenta. (Caso base)
$\mathcal{E}_2: \rho \vdash_{\Delta} \langle I, \sigma \rangle \rightarrow_e \langle n, \sigma \rangle$ <p>se <math>\rho(I) = n</math> o <math>(\rho(I) = l \text{ e } \sigma(l) = n)</math></p>	<b>Assioma identificatori</b>
$\mathcal{E}_3: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle e \text{ op } e_0, \sigma \rangle \rightarrow_e \langle e' \text{ op } e_0, \sigma \rangle}$	<b>Induttiva sinistra op:</b> e op e, valuta la prima e se gli operandi non sono valori primitivi allora devo valutarli, e in particolare valuto prima l'espressione a sinistra dell'operatore.
$\mathcal{E}_4: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle m \text{ op } e, \sigma \rangle \rightarrow_e \langle m \text{ op } e', \sigma \rangle}$	<b>Induttiva destra op:</b> m op e, valuta la e Se l'operando a sinistra è un valore, allora posso valutare l'operando a destra.
$\mathcal{E}_5: \rho \vdash_{\Delta} \langle t_1 \text{ bop } t_2, \sigma \rangle \rightarrow_e \langle t, \sigma \rangle$ <p>se <math>t_1 \text{ op } t_2 = t, t_1, t_2, t \in \mathcal{B}</math></p>	<b>Assioma bop:</b> Valuta l'espressione sintattica contenente un operatore booleano nell'risultato che rappresenta.
$\mathcal{E}_{3'}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle e \text{ bop } e_0, \sigma \rangle \rightarrow_e \langle e' \text{ bop } e_0, \sigma \rangle}$	<b>Induttiva sinistra bop</b> Estensione della regola 3 già vista anche agli operatori booleani, ma per espressioni + bop: stabiliamo che le cose si valutano da sinistra a destra.
$\mathcal{E}_6: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle t \text{ bop } e, \sigma \rangle \rightarrow_e \langle t \text{ bop } e', \sigma \rangle}$	<b>Induttiva destra bop</b> Analogo della regola e3, ma per valori booleani + op: Se l'operatore di sinistra è un booleano allora posso iniziare a valutare a destra
$\mathcal{E}_7: \rho \vdash_{\Delta} \langle \text{not } t_1, \sigma \rangle \rightarrow_e \langle t, \sigma \rangle$ <p>se <math>\text{not } t_1 = t, t_1 \in \mathcal{B}</math></p>	<b>Assioma not (unario)</b> Caso base nel caso di operatore booleano unario
$\mathcal{E}_8: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{not } e, \sigma \rangle \rightarrow_e \langle \text{not } e', \sigma \rangle}$	<b>Induttiva not (unario)</b> Caso induttivo nel caso di operatore booleano unario (valuto l'espressione)

## Chiusura transitiva

Ora riportiamo un sistema di regole equivalente ma che usa la chiusura transitiva nelle premesse.

$$\mathcal{E}_2: \frac{\rho \vdash_{\Delta} \langle I, \sigma \rangle \rightarrow_e^* \langle n, \sigma \rangle}{\text{se } \rho(I) = n \text{ o } (\rho(I) = l \text{ e } \sigma(l) = n)}$$

$$\mathcal{E}_{3-4}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle m, \sigma \rangle}{\rho \vdash_{\Delta} \langle e \text{ op } e_0, \sigma \rangle \rightarrow_e \langle k \text{ op } e_0, \sigma \rangle}$$

$$\mathcal{E}_{4-1}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle n, \sigma \rangle}{\rho \vdash_{\Delta} \langle m \text{ op } e, \sigma \rangle \rightarrow_e \langle k, \sigma \rangle}$$

$m \text{ op } n = k, m, n \in \mathcal{N} \quad k \in \mathcal{N}$

$$\mathcal{E}_{3'-6}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle t, \sigma \rangle}{\rho \vdash_{\Delta} \langle e \text{ bop } e_0, \sigma \rangle \rightarrow_e \langle t \text{ bop } e_0, \sigma \rangle}$$

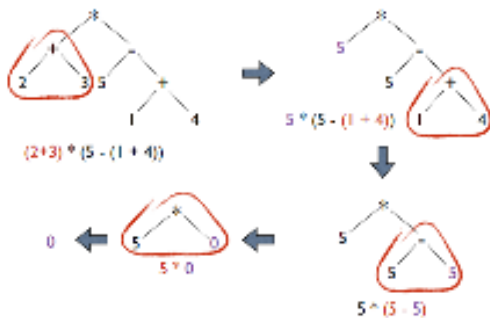
$$\mathcal{E}_{6-5}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle t_2, \sigma \rangle}{\rho \vdash_{\Delta} \langle t_1 \text{ bop } e, \sigma \rangle \rightarrow_e \langle t, \sigma \rangle}$$

se  $t_1 \text{ op } t_2 = t, t_1, t_2, t \in \mathcal{B}$

$$\mathcal{E}_{8-7}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle t_1, \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{not } e, \sigma \rangle \rightarrow_e \langle t, \sigma \rangle}$$

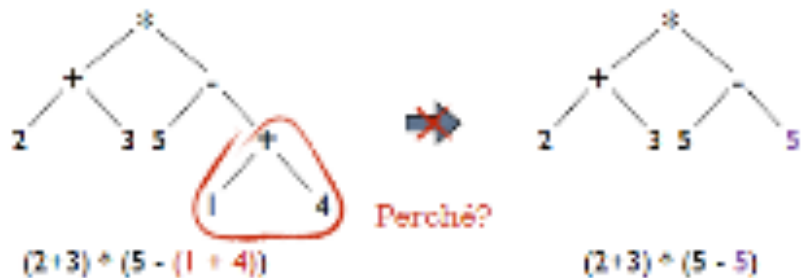
se  $\text{not } t_1 = t, t_1 \in \mathcal{B}$

## Esempio derivabile



## Esempio non derivabile:

Qui quello che succede è che viene valutata un'espressione a destra. Questo non va bene, poiché avremmo bisogno di una nuova regola, e con questa regola avremmo due diverse regole che gestiscono le situazioni dove ho exp op exp (bad!!)



## 4 – Dichiarazioni

Le dichiarazioni sono la categoria sintattica che riguarda la **creazione e la modifica degli ambienti**, ovvero degli insiemi di legami-**bindings associati agli identificatori**.

Le dichiarazioni devono essere **elaborate** per attuare la creazione e la trasformazione reversibile di legami; sono **reversibili** in quanto le trasformazioni valgono esclusivamente all'interno dello scope attuale, e le modifiche sono annullate automaticamente alla terminazione della validità dello scope.

Le dichiarazioni soffrono di **side effects**: potrebbero inizializzare una variabile e quindi modificare la memoria.

Di conseguenza **l'equivalenza deve richiedere che due dichiarazioni equivalenti generino le stesse modifiche a tutto lo stato di computazione**.

### Semantica delle dichiarazioni

Il significato di una dichiarazione è un ambiente (=insieme di legami).

Le dichiarazioni devono essere elaborate per ottenere la richiesta di creazione o modifica dell'ambiente.

Due dichiarazioni possono essere diverse ma essere elaborate nello stesso ambiente → equivalenti

Perché due dichiarazioni siano equivalenti devono rappresentare le stesse richieste **in tutti gli stati possibili**. Same 4 da side effects.

Per riferire valori generati dalle espressioni usiamo identificatori.

Gli identificatori sono nomi associati a un oggetto da riferire

Un nome è una sequenza di simboli usata per rappresentare qualcosa.

Per riferire i valori associati agli identificatori usiamo gli ambienti.

Un ambiente è un insieme di legami tra identificatori e oggetti denotabili.

Solitamente quando si parla di ambienti ci si riferisce solo alle associazioni stabilite dal programmatore; quindi a quella componente della macchina astratta che per ogni nome introdotto dal programmatore e per ogni punto di programma permette di determinare quale sia l'associazione corretta.

!!! L'ambiente non esiste a livello di macchina fisica: fa parte delle caratteristiche dei linguaggi ad alto livello.

La dichiarazione implementa la creazione dei legami.

Le occorrenze di uso sono, in genere, free occurrences. Se vogliamo dar loro significato bisogna renderle applied occurrences, inserendole nello scope di binding occurrences. Quindi le dichiarazioni forniscono le binding occurrences, che, dando significato all'identificatore, lo rendono legato.

Ambiente: insieme delle associazioni fra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma, e in uno specifico momento dell'esecuzione.

Dichiarazione: meccanismo implicito o esplicito col quale si crea un'associazione nell'ambiente.

### Termini chiusi e termini ground

<i>Termine chiuso</i>	<i>Termine ground</i>
In un linguaggio con identificatori, un termine in cui non ci sono identificatori liberi è detto chiuso. Significa che un programma pascal completo, ad esempio, non ha GI (identificatori liberi) e quindi può essere eseguito a partire da un ambiente vuoto, poiché non ci sono identificatori che richiedono un significato esterno. Un programma è un'occorrenza chiusa se ogni occorrenza d'uso è preceduta da un'occorrenza di definizione che stabilisce il significato dell'identificatore.	In un linguaggio con identificatori, un termine in cui non ci sono identificatori è detto ground.  Non avendo identificatori, non richiede nemmeno un ambiente!



## D in IMP: grammatica


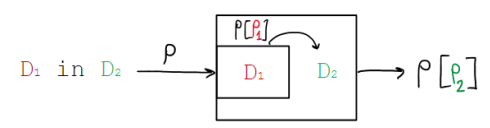
Ricordiamo che le dichiarazioni sono elaborate, e servono a creare legami tra identificatori e oggetti denotati. Le dichiarazioni

Introduciamo la sintassi delle dichiarazioni:

$\langle \text{Dec} \rangle \quad D \rightarrow \text{nil} \mid \text{const } I:\tau = e \mid \text{var } I:\tau = e$   
 $\quad \quad \quad D \text{ in } D \mid D ; D \mid \rho$   
 $\quad \quad \quad \text{proc } P(\text{form}) C \quad \text{form} = \text{ae}$

$\text{form} ::= \bullet \mid \text{const } x:\tau, \text{form} \mid \text{var } x:\tau, \text{form}$   
 $\text{ae} ::= \bullet \mid e, \text{ae}$

Dove

- $D$  è il simbolo del non terminale della grammatica che genera tutti i termini della categoria sintattica – ovvero tutte le dichiarazioni del nostro linguaggio.  $D^V$  è il nuovo insieme di dichiarazioni con variabili, che nella semantica poi costruiranno l'insieme delle configurazioni nel sistema di transizione. Mentre il valore dell'espressione serve solo per inizializzare la variabile – e quindi questo poi può cambiare – al contrario il tipo non può cambiare poiché il legame nell'ambiente statico avviene con una locazione di tipo  $\tau_0$ .
- $\text{nil}$  è la dichiarazione vuota, ovvero quella che crea l'ambiente vuoto.
- $\text{const } I:\tau = e$  serve a definire un identificatore  $I$  costante (=il cui valore non cambia durante l'esecuzione) di tipo  $\tau$ , inizializzato al valore  $e$ .
- $\rho$  rappresenta la dichiarazione completamente elaborata (così come i simboli numerici erano i valori terminali delle espressioni). Tra i simboli terminali delle dichiarazioni dobbiamo aggiungere anche gli ambienti: questo perché gli ambienti sono le configurazioni finali, e quindi devono essere generabili.
 
- $D_1; D_2$  definisce la composizione sequenziale:  $D_1$  definisce i legami che si uniscono a quelli di  $D_2$  e che possono essere utilizzati da  $D_2$ 

- $D_1 \text{ in } D_2$  definisce la composizione privata, ovvero i legami creati da  $D_1$  sono visibili e utilizzabili da  $D_2$  ma non sono visibili dopo l'elaborazione di  $D_2 \rightarrow D_1$  definisce i legami che risolvono le occorrenze libere presenti in  $D_2$ .
- $\text{proc } P(\text{form}) C$  serve a dichiarare le procedure.  $\text{form}$  dichiara i parametri formali delle procedure

## Valutazione ed equivalenza

**Elaborazione delle dichiarazioni con variabili:** l'elaborazione è una funzione  $\text{Elab} : \mathcal{D}^V \times \text{Mem} \rightarrow \text{Env} \times \text{Mem}$  che descrive il comportamento dinamico delle dichiarazioni l'ambiente in cui esse sono elaborate

$$\text{Elab}(\langle d, \sigma \rangle) = \langle \rho, \sigma' \rangle \Leftrightarrow \langle d, \sigma \rangle \rightarrow^* \langle \rho, \sigma \rangle$$

**Equivalenza di dichiarazioni con variabili:** l'equivalenza di dichiarazioni è una relazione  $\equiv \subseteq \mathcal{D}^V \times \mathcal{D}^V$  tale che

$$d_0 \equiv d_1 \Leftrightarrow \forall \sigma, \text{Elab}(\langle d_0, \sigma \rangle) = \text{Elab}(\langle d_1, \sigma \rangle)$$

## Funzioni FI e DI in $\mathcal{D}$

### Identificatori definiti DI

Stabiliamo quali identificatori sono liberi nelle dichiarazioni – ovvero non nello scope di nessuna definizione – e quali invece sono in posizione di definizione.

Visto che parliamo di dichiarazioni, partiamo dagli identificatori in posizione di definizione, in quanto l'obiettivo primario delle dichiarazioni è definire identificatori.

Dobbiamo definire una funzione  $DI : Dic \rightarrow \wp(\text{Id})$  che associa ad ogni dichiarazione l'insieme degli identificatori che definisce.

La definizione è induttiva:

$DI(\text{nil}) = \emptyset$	La dichiarazione nulla non definisce identificatori.
$DI(\text{const } x : \tau = e) = \{x\}$ $DI(\text{var } x : \tau = e) = \{x\}$	La dichiarazione di costante definisce precisamente l'identificatore che sta dichiarando; dove questa dichiarazione è visibile, l'identificatore $x$ è legato a questa definizione.
$DI(d_1; d_2) = DI(d_1) \cup DI(d_2)$	Nella composizione sequenziale, la composizione definisce tutto ciò che è definito nelle sue dichiarazioni. All'esterno della composizione, tutte le occorrenze di identificatori qui definite sono legate.
$DI(d_1 \text{ in } d_2) = DI(d_2)$	Ciò che viene definito nella prima dichiarazione rimane privato, dunque non risulta legato all'esterno della composizione; fuori da essa è legato solo ciò che è definito nella seconda dichiarazione.
$DI(\rho) = V, \text{ con } V \text{ dominio di } \rho$	Il valore terminale, ovvero gli ambienti, definisce tutti gli identificatori per i quali ha un'associazione.

$$DI(\text{proc } P(\text{form})C) = \{P\} \cup DI(\text{form}) \cup DI(C)$$

$$DI(\text{form} = ae) = DI(\text{form})$$

$$DI(\text{const } x : \tau, \text{form}) = \{x\} \cup DI(\text{form})$$

$$DI(\text{var } x : \tau, \text{form}) = \{x\} \cup DI(\text{form})$$

$$DI(\bullet) = \emptyset$$

### Identificatori liberi FI p

Gli identificatori liberi sono gli identificatori fuori dallo scope di una qualche dichiarazione.

Gli identificatori liberi sono calcolati dalla funzione  $FI : Dic \rightarrow \wp(\text{Id})$ , si possono ottenere anche sulle dichiarazioni e per induzione sulla struttura della grammatica.

FI applicata ad una dichiarazione restituisce l'insieme degli identificatori liberi nella dichiarazione.

$FI(\text{nil}) = \emptyset$	La dichiarazione nulla, non avendo identificatori, non ha identificatori liberi.
$FI(\text{const } x : \tau = e) = FI(e)$ $FI(\text{var } x : \tau = e) = FI\{e\}$	Gli identificatori liberi della dichiarazione costante sono tutti quelli usati nell'espressione che inizializza l'identificatore. Questi identificatori devono

	essere definiti nell'ambiente di valutazione affinché si possa valutare l'espressione.
$FI(d_1; d_2) = FI(d_1) \cup (FI(d_2) \setminus DI(d_1))$ $FI(d_1 \text{ in } d_2) = FI(d_1) \cup (FI(d_2) \setminus DI(d_1))$	Tutti gli identificatori di d1 sono liberi. Sono liberi anche tutti gli identificatori che sono liberi nella seconda dichiarazione d2 e non vengono definiti in d1
$FI(\rho) = \emptyset$	In un ambiente "completo" non abbiamo identificatori liberi, poiché contiene solo gli identificatori per cui ha definito associazioni.

$$FI(\text{proc } P(\text{form})C) = FI(C) \setminus DI(\text{form})$$

$$FI(\text{form} = ae) = FI(ae)$$

$$FI(e, ae) = FI(e) \cup FI(ae)$$

$$FI(\bullet) = \emptyset$$

## Semantica dinamica delle dichiarazioni

Per prima cosa definiamo l'insieme delle dichiarazioni con identificatori.

Dichiarazioni  $\mathcal{D}$  : insieme di associazioni tra identificatori e valori (oggetti denotati) derivabili nella grammatica.

L'insieme dei valori derivabili  $DVal : \{int, bool, \perp\}$  è già stato definito parlando degli identificatori.

### Sistema di transizione

Dove

#### SISTEMA DI TRANSIZIONE:

$$\Gamma = \mathcal{D}, T = Env$$

→ con memoria →

#### SISTEMA DI TRANSIZIONE:

$$\Gamma^V = \mathcal{D}^V \times Mem, T^V = Env \times Mem,$$

- $\Gamma$  : insieme delle configurazioni

È l'insieme delle dichiarazioni  $\mathcal{D}$  da elaborare in ambienti

- $T$ : insieme delle configurazioni finali

Sono gli ambienti (dinamici) che non hanno elementi da elaborare ulteriormente. Sono un sottoinsieme delle configurazioni.

Le regole del sistema di transazione, come le regole delle espressioni, si definiscono induttivamente sulla struttura sintattica delle dichiarazioni e avranno forma

Dove

$$\mathcal{D}_i : \rho \vdash_{\Delta} \langle d, \sigma \rangle \rightarrow_d \langle d', \sigma' \rangle$$

- $\rho$  è l'ambiente nel quale elaboriamo la dichiarazione, compatibile con l'ambiente statico  $\Delta$
- $\rightarrow_d$  con il pedice della freccia che serve ad indicare che la regola è del sistema delle dichiarazioni.

### REGOLE $\mathcal{D}_i$ : semantica dinamica delle dichiarazioni

$\mathcal{D}_1: \vdash \langle \mathbf{nil}, \sigma \rangle \rightarrow_d \langle \emptyset, \sigma \rangle$	<p><b>Assioma:</b> La dichiarazione nulla è elaborata nell'ambiente dinamico vuoto.</p>
$\mathcal{D}_2: \rho \vdash_{\Delta} \langle \mathbf{const} \ x:\tau=k, \sigma \rangle \rightarrow_d \langle [x \leftarrow k], \sigma \rangle$	<p><b>Assioma:</b> La dichiarazione costante (e ben formata) è elaborata nell'ambiente che associa all'identificatore definito dalla dichiarazione il valore inserito nella dichiarazione per l'inizializzazione. Se nella dichiarazione l'identificatore viene inizializzato con un'espressione non valutata dobbiamo prima valutare l'espressione, per poter arrivare a un valore costante.</p>
$\mathcal{D}_3: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle \mathbf{const} \ x:\tau=e, \sigma \rangle \rightarrow_d \langle \mathbf{const} \ x:\tau=e', \sigma \rangle}$	<p><b>Composizione di dichiarazioni:</b> in entrambi i casi dobbiamo prima elaborare la dichiarazione a sinistra, e solo quando questa è completamente elaborata possiamo iniziare ad elaborare quella a destra. Al contrario delle espressioni, qui non possiamo cambiare l'ordine di valutazione delle espressioni: nessuna delle due composizioni è commutativa!</p>
$\mathcal{D}_4: \frac{\rho \vdash_{\Delta} \langle d, \sigma \rangle \rightarrow_d \langle d', \sigma' \rangle}{\rho \vdash_{\Delta} \langle d; d_1, \sigma \rangle \rightarrow_d \langle d'; d_1, \sigma' \rangle}$ $\mathcal{D}_5: \frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle d_1, \sigma \rangle \rightarrow_d \langle d_1', \sigma' \rangle}{\rho \vdash_{\Delta} \langle \rho_0; d_1, \sigma \rangle \rightarrow_d \langle \rho_0; d_1', \sigma' \rangle}$ $\mathcal{D}_6: \rho \vdash_{\Delta} \langle \rho_0; \rho_1, \sigma \rangle \rightarrow_d \langle \rho_0[\rho_1], \sigma \rangle$	<p><b>Composizione sequenziale</b> D4 permette di iniziare la composizione a destra dopo aver terminato di valutare quella a sinistra. Quando anche la dichiarazione di destra è stata elaborata, allora possiamo applicare l'assioma D6 e ottenere come ambiente risultante l'ambiente associato alla prima dichiarazione aggiornato all'ambiente associato alla seconda dichiarazione.</p>

$\mathcal{D}_7: \frac{\rho \vdash_{\Delta} \langle d, \sigma \rangle \rightarrow_d \langle d', \sigma' \rangle}{\rho \vdash_{\Delta} \langle d \text{ in } d_1, \sigma \rangle \rightarrow_d \langle d' \text{ in } d_1, \sigma' \rangle}$ $\mathcal{D}_8: \frac{\rho[\rho_0] \vdash_{\Delta[\Delta^0]} \langle d_1, \sigma \rangle \rightarrow_d \langle d_1', \sigma' \rangle}{\rho \vdash_{\Delta} \langle \rho_0 \text{ in } d_1, \sigma \rangle \rightarrow_d \langle \rho_0 \text{ in } d_1', \sigma' \rangle}$ $\mathcal{D}_9: \rho \vdash_{\Delta} \langle \rho_0 \text{ in } \rho_1, \sigma \rangle \rightarrow_d \langle \rho_1, \sigma \rangle$	<p><b>Composizione privata</b></p> <p>Le regole sono analoghe, ma quanto entrambe le dichiarazioni sono state elaborate solo l'ambiente associato alla seconda dichiarazione viene restituito come ambiente risultante</p>
$\mathcal{D}_{10}: \rho \vdash_{\Delta} \langle \text{var } x:\tau=k, \sigma \rangle \rightarrow_d \langle [x \leftarrow l], \sigma[l \leftarrow k] \rangle$ <p style="text-align: center;"><math>l \in \text{Loc}_{\tau}</math> nuova locazione</p>	<p><b>Caso base var</b></p> <p>La configurazione generata consiste nella coppia composta dall'ambiente dinamico che associa all'identificatore variabile la nuova locazione <math>l</math> e alla memoria iniziale aggiornata con l'associazione tra <math>l</math> e il valore <math>k</math> calcolato.</p>
$\mathcal{D}_{11}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{var } x:\tau=e, \sigma \rangle \rightarrow_d \langle \text{var } x:\tau=e', \sigma \rangle}$	<p><b>Caso induttivo var</b></p> <p>Valutiamo l'espressione usata per inizializzare la variabile. Una volta che l'espressione è stata valutata ad un valore costante <math>k</math> allora possiamo effettivamente allocare lo spazio per la nuova variabile; in particolare, prendiamo una nuova locazione <math>l</math> di tipo <math>\tau</math>.</p>
$\mathcal{D}_{12}: \rho \vdash_{\Delta} \langle \text{proc } P()C, \sigma \rangle \rightarrow_d \langle [P \leftarrow \lambda \varepsilon. C'], \sigma \rangle$ <p>In caso di dichiarazione di una procedura:</p> <ul style="list-style-type: none"> <li>In caso di scope statico dobbiamo in qualche modo congelare e portarci dietro l'ambiente valido al momento della definizione, per poterlo usare quando la procedura viene chiamata;</li> <li>In caso di scope dinamico non è necessario, poiché la procedura viene eseguita nell'ambiente valido al momento della chiamata</li> </ul>	<p><math>C'</math> è determinato dallo scope:</p> <ul style="list-style-type: none"> <li><math>C' = \rho _{F(C)}</math> per scope statico (ambiente valido al momento della definizione)</li> <li><math>C' = C</math> per scope dinamico</li> </ul>
$\mathcal{D}_{13}: \rho \vdash_{\Delta} \langle \text{proc } P(\text{form})C, \sigma \rangle \rightarrow_d \langle [P \leftarrow \lambda \text{form}. C'], \sigma \rangle$ <p style="text-align: center;"><math>C' = \rho _{F(C)}; C</math> per scoping statico  <math>C' = C</math> per scoping dinamico</p>	<p>Inalterato <math>C'</math>; dobbiamo solo aggiungere i parametri formali</p>
$\mathcal{D}_{14}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle (e, ae), \sigma \rangle \rightarrow_{ae} \langle (e', ae), \sigma \rangle}$ $\mathcal{D}_{15}: \frac{\rho \vdash_{\Delta} \langle ae, \sigma \rangle \rightarrow_{ae} \langle ae', \sigma \rangle}{\rho \vdash_{\Delta} \langle (k, ae), \sigma \rangle \rightarrow_{ae} \langle (k, ae'), \sigma \rangle}$	<p>Nuova regola per associare parametri attuali e formali</p>

$\mathcal{D}_{16}: \frac{\rho \vdash_{\Delta} \langle ae, \sigma \rangle \rightarrow_{ae} \langle ae', \sigma \rangle}{\rho \vdash_{\Delta} \langle form=ae, \sigma \rangle \rightarrow_d \langle form=ae', \sigma \rangle}$	
$\mathcal{D}_{17}: \frac{ak, L \vdash form: \rho_0, \sigma_0}{\rho \vdash_{\Delta} \langle form=ak, \sigma \rangle \rightarrow_d \langle \rho_0, \sigma[\sigma_0] \rangle}$	Usiamo la lista degli attuali per crearne l'ambiente dinamico e la memoria, ovvero le associazioni corrispondenti. Questa regola dice che se la

## Semantica statica delle dichiarazioni

### Semantica statica: definizioni preliminari

Vogliamo innanzitutto associare una lista di tipi ai parametri attuali, una lista di tipi ai parametri formali. Poi una regola di semantica dovrà capire se questi due tipi coincidono.

Dobbiamo associare:

1. Come associamo una lista di tipo ai formali  $\rightarrow$  lista form.  
È banalmente la funzione che estrae la lista di tipi.

$$\begin{cases} T(\bullet) = \bullet \\ T(\text{const } x : \tau, form) = \tau, T(form) \\ T(\text{var } x : \tau, form) = \tau loc, T(form) \end{cases}$$

2. Associazione fra ambiente statico e lista di formali: costruisco l'ambiente statico

$$\begin{cases} \bullet : \emptyset \\ \frac{form : \Delta_0}{\text{const } x : \tau, form : \Delta_0[x = \tau]}, \Delta_0 : I_0, x \notin I_0 \\ \frac{form : \Delta_0}{\text{var } x : \tau, form : \Delta_0[x = \tau loc]}, \Delta_0 : I_0, x \notin I_0 \end{cases}$$

3. Lista dei tipi attuali:

$$\begin{cases} \Delta \vdash_I \bullet : \bullet \\ \frac{\Delta \vdash_I e : \tau, \Delta \vdash_I ae : aet}{\Delta \vdash_I e, ae : \tau, aet} \quad aet ::= \bullet \mid \tau, aet \end{cases}$$

### REGOLE $\mathcal{D}_{S1}$ : semantica statica delle dichiarazioni

Per quanto riguarda le dichiarazioni composte è necessario definire il concetto di aggiornamento degli ambienti. Per le dichiarazioni, la semantica statica deve associare un ambiente statico a ogni dichiarazione ben formata. Quindi, per le dichiarazioni, le regole hanno forma

$$\mathcal{D}_{S1}: \Delta \vdash_v d : \Delta$$

$\mathcal{D}_{S_1}: \vdash \text{nil}:\emptyset$	<p><b>Assioma</b></p> <p>La dichiarazione vuota genera un ambiente vuoto.</p>
$\mathcal{D}_{S_2}: \vdash \rho:\Delta \text{ se } \rho\vdash_{\Delta}$	<p><b>Assioma</b></p> <p>Nel caso di ambiente dinamico, l'ambiente statico è quello compatibile – ovvero che associa agli edintificatori esattamente i tipi dei valori che l'ambiente dinamico associa.</p>
$\mathcal{D}_{S_3}: \frac{\Delta\vdash_v e:\tau}{\Delta\vdash_v \text{const } x:\tau=e: [x\leftarrow\tau]}$	<p><b>Assioma: dichiarazione di costante</b></p> <p>Nella dichiarazione di costante, il tipo da associare è stabilito esplicitamente dalla dichiarazione, mentre abbiamo bisogno di valutare il tipo dell'espressione per verificare che la dichiarazione si sia ben formata.</p> <p>Quindi la dichiarazione è ben formata se l'espressione è ben formata ed è del tipo <math>\tau</math> presente nella dichiarazione.</p>
$\mathcal{D}_{S_4}: \frac{\Delta\vdash_v d_1:\Delta_1 \quad \Delta[\Delta_1]\vdash_{vuv'} d_2:\Delta_2}{\Delta\vdash_v d_1 \text{ in } d_2:\Delta_2}$	<p>Nelle premesse dobbiamo trovare innanzitutto l'ambiente statico <math>\Delta_1</math> associato alla dichiarazione <math>d_1</math> se questa è ben formata. Questo ambiente viene usato per aggiornare l'ambiente con le nuove associazioni create da <math>d_1</math>.</p> <p>L'ambiente statico risultante viene usato per associare l'ambiente <math>\Delta_2</math> alla dichiarazione <math>d_2</math>.</p> <p>A questo punto, l'ambiente associato alla dichiarazione composta è solo l'ambiente associato a <math>d_2</math>, in quanto <math>d_1</math> (e quindi <math>\Delta_1</math>) è esclusivamente visibile in <math>d_2</math>.</p>
$\mathcal{D}_{S_5}: \frac{\Delta\vdash_v d_1:\Delta_1 \quad \Delta[\Delta_1]\vdash_{vuv'} d_2:\Delta_2}{\Delta\vdash_v d_1;d_2:\Delta_1[\Delta_2]}$	<p>Dobbiamo associare l'ambiente statico <math>\Delta_1</math> corrispondente alla dichiarazione di <math>d_1</math>; poi si aggiorna l'ambiente esterno <math>\Delta</math> con questo ambiente <math>\Delta_1</math> e quindi nell'ambiente risultante <math>\Delta[\Delta_1]</math> si elabora la dichiarazione <math>d_2</math> che viene associata all'ambiente <math>\Delta_2</math>. Infine, l'ambiente che si associa alla dichiarazione composta sequenzialmente è esattamente <math>\Delta_1</math> aggiornato da <math>\Delta_2</math>, ovvero <math>\Delta_1[\Delta_2]</math>.</p> <p>Questo significa che tutto ciò che è dichiarato nella dichiarazione composta è visibile dall'esterno, tranne ciò che viene definito da <math>d_1</math> e ridefinito nella seconda dichiarazione.</p>

$$\mathcal{D}_{S_6}: \frac{\Delta \vdash e : \tau}{\Delta \vdash \text{var } x : \tau = e : [x \leftarrow \tau \text{loc}]}$$

### Assioma: dichiarazione di variabile

L'ambiente statico che si ottiene associa alla variabile  $x$  il tipo  $\tau \text{loc}$ . Quindi, per esempio, se dichiariamo una variabile  $x$  di tipo `int` inizializzata a 3, allora la dichiarazione è ben formata e l'ambiente statico associa a  $x$  il tipo `intloc`.

$$\mathcal{D}_{S_7}: \frac{\text{form} : \Delta_0 \quad \Delta[\Delta_0] \vdash_{\text{vuv}'} C}{\Delta \vdash_{\text{v}} \text{proc } P(\text{form})C : [P = \mathcal{T}(\text{form})\text{proc}]}$$

### Dichiarazione di procedura

( $\mathcal{T}$  è la funzione che estrae i tipi dalla lista dei formali!)

Dobbiamo costruire l'ambiente statico corrispondente alla dichiarazione. Quindi dobbiamo modificare la regola di dichiarazione di procedura per avere una lista di parametri: costruiamo l'ambiente associato ai formali, verifichiamo che il corpo  $C$  è ben formato. Il tipo che associamo alla procedura consiste nella lista dei tipi dei formali seguiti dal suffisso `proc`, che dice che l'identificatore è una procedura.

$$\mathcal{D}_{S_8}: \frac{\text{form} : \Delta_0 \quad \Delta \vdash_{\text{v}} a e : \mathcal{T}(\text{form})}{\Delta \vdash_{\text{v}} \text{form} = a e : \Delta_0}$$

Nova regola per associare formali e attuali. Se la lista dei tipi degli attuali coincide con la lista dei tipi dei formali, allora l'ambiente statico generato dai formali è associato alla dichiarazione.

## LE REGOLE DELLE DICHIARAZIONI CON VARIABILI E CHIUSURA TRANSITIVA

Ora riportiamo le regole che usano la chiusura transitiva nelle premesse.

$$\mathcal{D}_1: \vdash \langle \text{nil}, \sigma \rangle \rightarrow_d \langle \emptyset, \sigma \rangle$$

$$\mathcal{D}_{3-2}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle k, \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{const } x : \tau = e, \sigma \rangle \rightarrow_d \langle [x \leftarrow k], \sigma \rangle}$$

$$\mathcal{D}_{7-8}: \frac{\rho \vdash_{\Delta} \langle d, \sigma \rangle \rightarrow_d^* \langle \rho_0, \sigma' \rangle}{\rho \vdash_{\Delta} \langle d \text{ in } d_1, \sigma \rangle \rightarrow_d \langle \rho_0 \text{ in } d_1, \sigma' \rangle}$$

$$\mathcal{D}_{4-5}: \frac{\rho \vdash_{\Delta} \langle d, \sigma \rangle \rightarrow_d^* \langle \rho_0, \sigma' \rangle}{\rho \vdash_{\Delta} \langle d_1 d_2, \sigma \rangle \rightarrow_d \langle \rho_0 d_2, \sigma' \rangle}$$

$$\mathcal{D}_{8-9}: \frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle d_1, \sigma \rangle \rightarrow_d \langle \rho_1, \sigma' \rangle}{\rho \vdash_{\Delta} \langle \rho_0 \text{ in } d_1, \sigma \rangle \rightarrow_d \langle \rho_1, \sigma' \rangle}$$

$$\mathcal{D}_{5-6}: \frac{\rho[\rho_0] \vdash_{\Delta[\Delta_0]} \langle d_1, \sigma \rangle \rightarrow_d^* \langle \rho_1, \sigma' \rangle}{\rho \vdash_{\Delta} \langle \rho_0 d_1, \sigma \rangle \rightarrow_d \langle \rho_0[\rho_1], \sigma' \rangle}$$

$$\mathcal{D}_{11-10}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle k, \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{var } x : \tau = e, \sigma \rangle \rightarrow_d \langle [x \leftarrow 1], \sigma[1 \leftarrow k] \rangle}$$



## 5 – Comandi (contesto)

I comandi sono la categoria sintattica che denota la richiesta di modifica della memoria.

I comandi devono essere eseguiti per attuare la richiesta (irreversibile) di modifica della memoria. Sono irreversibili in quanto le trasformazioni valgono dal momento in cui sono effettuate in poi, e possono essere neutralizzate (! Non annullate) solo tramite un altro comando che effettua una richiesta di modifica che neutralizza la precedente.

Possiamo avere side effects: il comando potrebbe generare valori o creare legami, e per questo l'equivalenza deve necessariamente richiedere che due comandi equivalenti generino le stesse modifiche a tutto lo stato di computazione.

Come rappresentiamo lo stato? In generale, l'astrazione dello stato della macchina è rappresentato dall' memoria. La memoria è un insieme di associazioni tra locazioni e valori, e possiamo immaginarlo come un array illimitato in cui ogni indice è una locazione e il valore corrispondente è il contenuto della locazione. A questo punto, per trasformare lo stato, abbiamo bisogno di un meccanismo per trasformare memorie.

Questo meccanismo è l'assegnamento, ovvero il comando che permette di modificare il contenuto della memoria. Non è pensabile che chi programma possa usare direttamente le locazioni di memoria, quindi queste locazioni vengono riferite attraverso identificatori che chiameremo variabili.

Le variabili sono identificatori associati a locazioni, che a loro volta sono associate a valori nella memoria che possono cambiare durante l'esecuzione.

I comandi contengono altri costrutti per la composizione e il controllo dei flussi di esecuzione. Va tenuto presente che i comandi, come categoria sintattica, sono tipici del paradigma imperativo – ovvero non sono presenti in linguaggi logici o funzionali puri.

### Memoria fisica e variabili

Non è possibile agire sulle locazioni direttamente, in quanto questo renderebbe i programmi dipendenti dall'implementazione fisica; per poter agire sullo stato in modo indipendente dall'architettura abbiamo bisogno di astrarre il concetto di cella di memoria con il concetto di variabile, ovvero di identificatore usato per riferirsi al contenuto di una cella.

#### Associazione semplice identificatore → valore

Per il momento supponiamo che le variabili siano nomi astratti per le celle, e di associare direttamente il valore alla variabile.

Identificatore → Valore

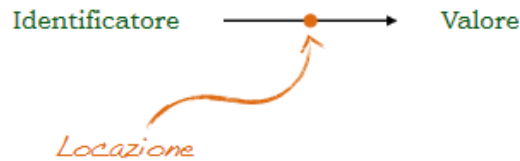
Questo modello è molto poco adatto:

- Quando ci riferiamo ad un valore modificabile tramite variabile, lo possiamo fare sia per accedere al suo valore in lettura sia per modificare il suo valore
- Per riferirsi a oggetti da modificare potremmo utilizzare funzioni del nome della variabile, e non il semplice nome (es.  $a[x+1]$ ), oppure procedure (es.  $\text{fattoriale}(x)$ )
- Gli identificatori possono essere ridefiniti, causando modifiche reversibili all'ambiente. Questo significa che per un certo intervallo di tempo una variabile potrebbe avere un altro significato, mentre il valore che aveva prima della ridefinizione non deve andare perso in quanto potrebbe tornare valido – ad esempio all'uscita da una procedura.

→ Senza separare nome da valore con un concetto intermedio non riusciamo a gestire queste situazioni.

## Associazione identificatore→locazione → valore

Per gestire queste situazioni raffiniamo lo stato: astrarre la cella di memoria in un nome è troppo forte.



Introduciamo quindi una locazione come astrazione della memoria fisica, e la variabile come astrazione della locazione. Questo significa che introduciamo qualcosa di intermedio tra il nome della variabile e il valore riferito, che modelli la memoria nella sua visione ideale, ovvero illimitata: ogni nuovo oggetto può usufruire di una nuova locazione. Così è possibile che una variabile abbia indirizzi differenti in momenti diversi dell'esecuzione e in diversi punti del programma.

Se nello stesso momento due nomi di variabili possono essere usati per accedere alla stessa locazione, questi sono detti alias.

Così, distinguiamo:

- **Nome-locazione:** la parte dell'associazione con la variabile che non cambia durante l'esecuzione se non in modo reversibile
- **Locazione-valore:** dalla parte che cambia in modo irreversibile durante l'esecuzione

E distinguiamo l'associazione propria del linguaggio tra nome e oggetto denotato dalla rappresentazione astratta della memoria. Quindi, anche logicamente, manteniamo la distinzione linguaggio-architettura.

I linguaggi sono astrazione dell'architettura sottostante, come composizione delle macchine astratte sottostanti il livello del linguaggio. I linguaggi ci permettono di eseguire azioni sulla macchina trasformandone lo stato.

A questo punto, lo stato della macchina viene rappresentato attraverso la sua memoria, ovvero attraverso l'insieme di associazioni tra locazioni (riferibili tramite variabili) e valori memorizzabili.

Quindi, la memoria è lo strumento che utilizziamo nella nostra semantica per tenere traccia dell'evoluzione dei valori delle variabili attraverso la descrizione dei valori che stanno nelle locazioni associate.

- **Locazione/indirizzo:** contenitore per i valori che può essere inutilizzata, indefinita o definita
- **Memoria:** collezione di associazioni con identificatori, che vengono aggiornate dinamicamente.

Quindi la locazione diventa un valore denotabile, che modella l'indirizzo di memoria a cui la variabile è associata. Questo significa che la locazione viene creata da una dichiarazione. Ergo, la locazione non deve necessariamente essere riferibile per sempre: ha un tempo di **vita** determinato dallo **scope** della dichiarazione.

Inoltre, il modo con cui si **rilascia** la locazione determina la **reversibilità** dei cambiamenti: dove il rilascio è implicito (automatico) il cambiamento è reversibile, mentre dove il rilascio deve essere esplicito (allocazione dinamica) anche questi cambiamenti diventano non reversibili.

Nella memoria parliamo esclusivamente di associazioni e non di bindings per distinguere il fatto che le associazioni della memoria possono cambiare nel valore associato.

→ Memoria e locazioni sono una conseguenza (quasi) inevitabile del modello imperativo di computazione.

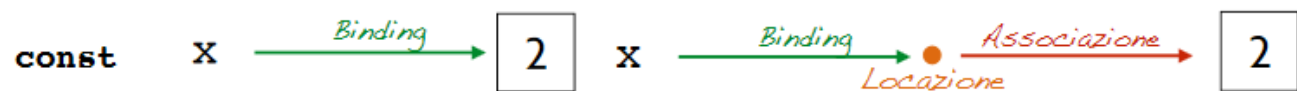
## Dichiarazioni: costanti e variabili

Dovendo creare legami anche con locazioni, la categoria sintattica delle dichiarazioni va arricchita; in particolare, ci serve un nuovo costrutto che ci permette di dichiarare variabili – ovvero associare locazioni a identificatori.

La dichiarazione già usata è quella di costante. La costante è un identificatore che può assumere un valore predefinito, e che non è ulteriormente modificabile

Vogliamo aggiungere la possibilità di legare un nome ad una locazione, per poter modificare il valore: in tal modo modelliamo l'opportunità di associare un valore modificabile alla locazione nella memoria da riferire mediante il nome.

In particolare, la differenza tra `const` e `var` è la seguente:



Dove

- I bindings sono i legami collezionati dentro l'ambiente dinamico, che contiene tutte le associazioni che non cambiano una volta creati
- Le associazioni con locazioni sono collezionate nella memoria, che contiene associazioni che cambiano valore durante l'esecuzione.

Ovviamente questo significa che ai valori denotabili dovremo aggiungere anche le locazioni  $\mathcal{L}$  con metavariable  $l$ . Di conseguenza,

$$DVal = \mathcal{N} \cup \mathcal{B} \cup \mathcal{L}$$

Le variabili, al contrario delle costanti, sono qualcosa più di un identificatore associato ad un oggetto denotabile. Sono un elemento complesso caratterizzato da vari aspetti:

- Come le costanti:
  - Nome/identificatore: la sequenza di caratteri utilizzato per riferirsi alla denotazione
  - Tipo: l'insieme di valori riferibili alla variabile, ovvero memorizzabili nella locazione legata all'identificatore e delle operazioni possibili su questi valori
  - Valore: l'oggetto memorizzato nella locazione associata all'identificatore; valore che deve essere tra quelli identificati dal tipo di variabile.  
→ L'uso di un identificatore per riferirsi al valore memorizzato è detto anche *r-value*.
- Tipici delle variabili:
  - **Indirizzo/Locazione**: astrazione della cella di memoria in cui il valore associato viene memorizzato. Consiste nell'oggetto denotabile legato all'identificatore della variabile.  
→ L'uso dell'identificatore per riferirsi all'alocazione è anche detto *l-value*
  - **Scope**: il range dei comandi a cui la variabile è visibile, ovvero dove è visibile la sua dichiarazione. Lo scope è determinato dall'ambiente e da regole specifiche dette regole di scope
  - **Tempo di vita**: tempo in cui il binding con la locazione è attivo.

## Memoria e locazioni

Le locazioni sono modellate come insieme finito ma illimitato di elementi che rappresentano le celle di memoria.

$Loc$  = collezione illimitata di locazioni

$L \subseteq Loc \quad New(L) \in Loc \setminus L$

La funzione  $New$  è una funzione che preso un insieme di locazioni già utilizzate ne genera una nuova.

Siano  $MVal$  i valori memorizzabili, ovvero quelli che possono essere associati ad una locazione. Nel nostro linguaggio definiamo  $MVal = \mathcal{N} \cup \mathcal{B}$  per definire formalmente il concetto di memoria:

Una *memoria* è un elemento dello spazio di funzioni  $Mem$  definito da

$$Mem = \bigcup_{L \subseteq_f Loc} Mem_L$$

dove  $Mem_L : L \rightarrow MVal$  ha metavariable  $\sigma$  e  $MVal$  sono i valori memorizzabili.

Dove  $Loc_\tau$  è l'unione degli insiemi  $Loc$  delle locazioni che possono memorizzare valori di tipo  $\tau$ .

Per definire la semantica dei comandi dobbiamo definire cosa significa aggiornare uno stato/memoria. Questi valori contengono, solitamente, anche un valore indefinito – quindi in generale l'aggiornamento è descritto come un aggiornamento puntuale che può essere generalizzato.

Si considerino due memorie  $\sigma, \sigma' \in Mem$  dove  $\sigma : L, \sigma' : L' (L, L' \subseteq Loc)$  ovvero  $\sigma$  è definito sulle locazioni in  $L$ ,  $\sigma'$  è definito sull'insieme di locazioni  $L'$ . L'aggiornamento della memoria  $\sigma$  mediante la memoria  $\sigma'$  è la memoria  $\sigma'' \in Mem$ , denotato  $\sigma[\sigma']$ , definita come

$$\sigma''(l) = \begin{cases} \sigma'(l) & \text{se } l \in L' \\ \sigma(l) & \text{else} \end{cases}$$

Ovvero la memoria che aggiorna ha la precedenza.

le altre regole aggiungono la memoria alla configurazione.

## 5 - Comandi

Il significato di un comando è essenzialmente una trasformazione di stato – ovvero di memorie.

I comandi devono essere eseguiti per ottenere la richiesta di trasformazione dello stato/memoria.

Due comandi possono essere DIVERSI ma, se eseguiti, richiedono la stessa trasformazione di stato – ovvero restituire lo stesso stato risultante per tutti gli stati da cui può partire. In tal caso, si dicono equivalenti.

→ In caso di presenza di side effects, anche questi devono essere esattamente gli stessi in tutti gli stati possibili.

### C in IMP: grammatica

```
<Com>  C – skip | I := e | C ; C |  
        if B then C else C |  
        while B do C | {D;C} | P(ae)  ae ::= • | e, ae
```

- **skip** non esegue alcuna trasformazione; ci serve come elemento neutro della composizione sequenziale di comandi.
- **I := e** è l'assegnamento, ovvero il comando base dei linguaggi imperativi, che assegna il valore di un'espressione ad un identificatore. È il comando primitivo per l'aggiornamento di variabili. Da questo costrutto la terminologia che vede l'uso di un identificatore a sinistra del simbolo di assegnamento (per aggiornarlo) come l-value, mentre l'uso a destra del simbolo di assegnamento (per accedere al valore in memoria) come r-value
- **;** è la composizione sequenziale, ovvero tutte le trasformazioni eseguite dal primo comando sono il punto di partenza per le trasformazioni che deve eseguire il secondo comando
- **if b then C else C** è detto selettore a due vie, e sceglie l'esecuzione in funzione del valore dell'espressione di guardia.
- **while b do C** è il comando iterativo e permette di ripetere un'esecuzione a comando finché la guardia rimane vera.
- **D; C** crea un ambiente locale al comando.
- **P(ae)** serve a poter richiamare una procedura. Ae sono i parametri attuali.

### Semantica di C

#### Sistema di transizione

I comandi sono eseguiti per trasformare memorie: dunque, le trasformazioni saranno coppie contenenti memorie (da trasformare) e comandi (che eseguono trasformazioni).

Il risultato di una transazione sarà una nuova configurazione, con ciò che resta da eseguire del comando e la memoria risultante della trasformazione; se non abbiamo più nulla da eseguire, il risultato sarà una semplice memoria.

SISTEMA DI TRANSIZIONE:  
 $IV = \mathcal{E}^V \times \text{Mem} \cup \text{Mem}, TV = \text{Mem},$

In generale, la struttura delle regole di transizione è

Dove:

- $C_i : \rho \vdash_{\Delta} \langle c, \sigma \rangle \rightarrow_c \langle e', \sigma' \rangle$
- $\rho$  ambiente esterno dinamico: associa valori e/o locazioni agli identificatori, ed è compatibile con l'ambiente statico
  - $\Delta$  ambiente statico
  - $\sigma$  memoria iniziale
  - $c$  comando da eseguire a partire dalla memoria iniziale  $\sigma$ , e che restituisce  $c'$  e  $\sigma'$

La semantica statica per i comandi serve solo a verificare che il comando sia ben formato, ovvero rispetti tutti i vincoli semantici del comando.

Con  $\Delta \vdash_v c$  diciamo che il comando è ben formato nell'ambiente statico  $\Delta$  definito sull'insieme di identificatori  $V$ .

Vediamo un comando per volta. Diocristo.

### Assegnamento $I := e$

L'esecuzione di un assegnamento non restituisce un valore, ma produce trasformazioni irreversibili della memoria.

Sintatticamente è rappresentato in modo diverso in linguaggi diversi: '=' in Fortran, BASIC; C; ':=' in Ada e IMP.

Supportato che lo stato è rappresentato in termini della memoria come associazione locazione – valore. A questo punto dobbiamo osservare che in un insegnamento il ruolo di un identificatore  $x$  a sinistra e a destra del simbolo è sintatticamente diverso:

- A sinistra denota una locazione – l-value
- A destra denota il contenuto di una locazione – r-value

L'assegnamento è l'unico comando atomico, ovvero non scritto in termini di altri comandi.

### Sintassi

**$I := e$**

### Identificatori liberi e definiti

Tutti gli identificatori presenti nel comando sono liberi, poiché un assegnamento non può contenere dichiarazioni.

$$FI(x := e) = FI(e) \cup \{x\} \quad DI(x := e) = \emptyset$$

### Semantica statica

$$\mathcal{C}_{S1} : \frac{\Delta \vdash_v e : \tau}{\Delta \vdash_v x := e}, \Delta(x) = \tau_{loc}$$

L'assegnamento è ben formato se l'identificatore a sinistra del simbolo di assegnamento è un identificatore variabile del tipo dell'espressione assegnata. Quindi dobbiamo valutare il tipo  $\tau$  dell'espressione nello stesso ambiente statico, e questo deve corrispondere al tipo associato nell'identificatore – che sarà dunque  $\tau_{loc}$ .

### Semantica dinamica

$\mathcal{C}_1: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle x := e, \sigma \rangle \rightarrow_c \langle x := e', \sigma \rangle}$	Dobbiamo innanzitutto valutare l'espressione contenuta.
$\mathcal{C}_2: \rho \vdash_{\Delta} \langle x := k, \sigma \rangle \rightarrow_c \sigma[l \leftarrow k], \rho(x) = l$	Una volta che l'espressione è completamente valutata possiamo aggiornare la memoria associando a x il nuovo valore ottenuto, e restituire la memoria come configurazione terminale.

### Selettori

Il controllo dell'esecuzione avviene nelle espressioni, tra comandi e tra unità di programma (procedure).

Il controllo avviene grazie a delle strutture di controllo, ovvero comandi che specificano l'ordine con cui le modifiche devono essere fatte: specificano alternative e ripetono comandi.

La prima classe di strutture di controllo sono i comandi di selezione, ovvero i comandi che forniscono gli strumenti per scegliere tra due o più cammini di esecuzione: selettori a due o più vie.

Questo costrutto consiste nello scegliere una o l'altra via in funzione della valutazione booleana di controllo. I quesiti di progettazione sono:

- Quale è la forma e il tipo delle espressioni di controllo? Di solito è un'espressione booleana, ma in alcuni può essere aritmetica
- Come sono specificate le clausole then e else? In molti linguaggi moderni possono essere comandi sia singoli che composti, e possono essere delimitate in vari modi a seconda dei linguaggi.
- Come devono essere specificati i selettori annidati per non avere ambiguità?

### Condizionale: if-then-else

#### Sintassi

**if** *b* **then** *c* **else** *c*

#### Identificatori liberi e definiti

Il comando condizionale non altera né l'insieme degli identificatori liberi (che quindi sono l'unione dei due rami e nella condizione), né l'insieme dei definiti.

$$FI(\text{if } e \text{ then } c_0 \text{ else } c_1) = FI(c_0) \cup FI(c_1) \cup FI(e)$$

$$DI(\text{if } e \text{ then } c_0 \text{ else } c_1) = DI(c_0) \cup DI(c_1)$$

### Semantica statica

$\mathcal{C}_{S2}: \frac{\Delta \vdash_v e : \text{bool} \quad \Delta \vdash_v c_0 \quad \Delta \vdash_v c_1}{\Delta \vdash_v \text{if } e \text{ then } c_0 \text{ else } c_1}$	Deve solo verificare se il comando è ben formato. I vincoli da verificare sono che l'espressione sia di tipo booleano, e che i rami siano ben formati.
--	--

### Semantica dinamica

$\mathcal{C}_3: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e \langle e', \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{if } e \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_c \langle \text{if } e' \text{ then } c_0 \text{ else } c_1, \sigma \rangle}$	Dobbiamo valutare l'espressione di guardia: se è true allora si sceglie un ramo, altrimenti l'altro.
$\mathcal{C}_4: \rho \vdash_{\Delta} \langle \text{if true then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_c \langle c_0, \sigma \rangle$	
$\mathcal{C}_5: \rho \vdash_{\Delta} \langle \text{if false then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_c \langle c_1, \sigma \rangle$	

### Composizione

#### Sintassi

**C ; C**

#### Identificatori liberi e definiti

È intuitivo: la composizione non altera gli identificatori, quindi è la semplice unione.

$$FI(c_0; c_1) = FI(c_0) \cup FI(c_1)$$

$$DI(c_0; c_1) = DI(c_0) \cup DI(c_1)$$

### Semantica statica

$\mathcal{C}_{S3}: \frac{\Delta \vdash_v c_0 \quad \Delta \vdash_v c_1}{\Delta \vdash_v c_0; c_1}$	Basta che i comandi siano ben formati.
$\mathcal{C}_{S4}: \Delta \vdash_v \text{skip}$	Comando nullo messo qui a caso.

### Semantica dinamica

$\mathcal{C}_7: \frac{\rho \vdash_{\Delta} \langle c, \sigma \rangle \rightarrow_c \langle c', \sigma' \rangle}{\rho \vdash_{\Delta} \langle c; c_0, \sigma \rangle \rightarrow_c \langle c'; c_0, \sigma' \rangle}$	
$\mathcal{C}_8: \frac{\rho \vdash_{\Delta} \langle c, \sigma \rangle \rightarrow_c \sigma'}{\rho \vdash_{\Delta} \langle c; c_0, \sigma \rangle \rightarrow_c \langle c_0, \sigma' \rangle}$	
$\mathcal{C}_6: \rho \vdash_{\Delta} \langle \text{skip}, \sigma \rangle \rightarrow_c \sigma$	Comando nullo messo qui a caso.



# REGOLE TRANSITIVE

$$\mathcal{C}_{1-2}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle k, \sigma \rangle}{\rho \vdash_{\Delta} \langle x := e, \sigma \rangle \rightarrow_c \sigma[l \leftarrow k]} \quad \rho(x) = l$$

$$\mathcal{C}_{3-4}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle \text{true}, \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{if } e \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_c \langle c_0, \sigma \rangle}$$

$$\mathcal{C}_{3-5}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle \text{false}, \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{if } e \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_c \langle c_1, \sigma \rangle}$$

$$\mathcal{C}_6: \rho \vdash_{\Delta} \langle \text{skip}, \sigma \rangle \rightarrow_c \sigma$$

$$\mathcal{C}_{7-8}: \frac{\rho \vdash_{\Delta} \langle c, \sigma \rangle \rightarrow_c^* \sigma'}{\rho \vdash_{\Delta} \langle c; c_0, \sigma \rangle \rightarrow_c \langle c_0, \sigma' \rangle}$$

## Strutture di controllo – comandi iterativi

Per ottenere linguaggi di turing completi abbiamo bisogno di inserire nel linguaggio la possibilità di ripetere comandi. Iterazione e ricorsione sono i due meccanismi che permettono di ottenere formalismi di calcolo di Turing completi.

La questione principale consiste nel determinare come avviene il controllo.

- Iterazione indeterminata: i cicli sono controllati logicamente, ovvero su un'espressione booleane (while, repeat)
  - Aspetti di progettazione:
    - Esecuzione di pre-test o post-test?
    - Il comando di iterazione non determinata deve essere un caso particolare di quello per l'iterazione determinata o un comando separato?
- Iterazione determinata: i cicli sono controllati numericamente con un numero di ripetizioni del ciclo determinate al momento dell'inizio del ciclo.
  - Solitamente ha una variabile di ciclo, e si basa su un valore iniziale, finale e un passo di incremento.
  - Aspetti di progettazione:
    - Qual è il tipo e la visibilità della variabile di ciclo?

- È da ritenersi legale la modifica della variabile o dei parametri di ciclo dentro il corpo del comando di controllo? La modifica allora ha effetto sul ciclo?
- I parametri di ciclo devono essere valutati una sola volta o a ogni iterazione?

### Iterazione con transitività: while

#### Sintassi

**while** **b** **do** **c**

#### Identificatori liberi e definiti

Anche il while non altera i due insiemi.

$$FI(\text{while } e \text{ do } c) = FI(c) \cup FI(e)$$

$$DI(\text{while } e \text{ do } c) = DI(c)$$

#### Semantica statica

$$\mathcal{C}_{S5}: \frac{\Delta \vdash_v e : \text{bool} \quad \Delta \vdash_v c}{\Delta \vdash_v \text{while } e \text{ do } c}$$

Deve solo verificare se il comando è ben formato, ovvero che l'espressione sia booleana e che il comando nel corpo sia ben formato.

#### Semantica dinamica

$$\mathcal{C}_9: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle \text{true}, \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{while } e \text{ do } c, \sigma \rangle \rightarrow_c \langle c; \text{while } e \text{ do } c, \sigma \rangle}$$

$$\mathcal{C}_{10}: \frac{\rho \vdash_{\Delta} \langle e, \sigma \rangle \rightarrow_e^* \langle \text{false}, \sigma \rangle}{\rho \vdash_{\Delta} \langle \text{while } e \text{ do } c, \sigma \rangle \rightarrow_c \sigma}$$

Dobbiamo valutare l'espressione di guardia, e se questa è true allora si sceglie di eseguire c seguito dal comando while; altrimenti si termina. La definizione è ricorsiva, ma dal punto computazionale non permette di dimostrare terminazione – quindi non è ben formata, e di conseguenza induttiva. È comunque calcolabile: la non induttività implica solo che la terminazione va dimostrata esplicitamente.

### Richiamo procedure

#### Identificatori liberi e definiti

$$DI(P(ae)) = \emptyset$$

$$FI(P(ae)) = FI(ae) \cup \{P\}$$

#### Semantica statica

$$\mathcal{C}_{S7}: \frac{\Delta \vdash_v ae : aet}{\Delta \vdash_v P(ae)} \quad \Delta(P) = aetproc$$

La chiamata è ben formata se la lista dei tipi degli attuali coincide con la lista dei tipi dei formali. La lista si trova esplicitata nel tipo della procedura che la dichiarazione ha associato alla procedura in dichiarazione.

## Semantica dinamica

Il controllo passa dal comando chiamante al corpo della procedura – che in caso di scope statico è un blocco contenente anche il proprio ambiente statico.

$$\mathcal{B}_{14}: \rho \vdash_{\Delta} \langle P(ae), \sigma \rangle \rightarrow_c \langle \text{form}=ae; C', \sigma \rangle$$

$$\rho(P) = \lambda \text{form}. C'$$

La chiamata deve eseguire il corpo della procedura, ma deve eseguire tale corpo nell'ambiente generato dall'associazione tra attuali e formali; per questo, prima di eseguire il corpo, deve elaborare la dichiarazione  $\text{form}=ae$ . In tale dichiarazione la lista dei formali è presente nel valore associato al nome della procedura, mentre la lista degli attuali è la chiamata.

```
proc esempio() {
  c { var x:int = 5;
    z := x+y
  } d
}
```

Definizione:

$$[y \leftarrow 3] \vdash \langle d, \sigma \rangle \rightarrow_d [esempio \leftarrow \lambda e. C']$$

scoping statico:  $C' = [y \leftarrow 3]; C$

scoping dinamico:  $C' = C$

Chiamata:

$$[y \leftarrow 4] \vdash \langle \text{esempio}(), \sigma \rangle \rightarrow_e \langle C', \sigma \rangle$$

scoping dinamico

$$\rightarrow [y \leftarrow 4] \vdash \langle C, \sigma \rangle \rightarrow_c \dots$$

scoping statico

$$\rightarrow [y \leftarrow 4] \vdash \langle [y \leftarrow 3]; C, \sigma \rangle \rightarrow_c \dots$$

## Esecuzione ed equivalenza

**Esecuzione di comandi:** l'esecuzione è una funzione  $Exec : \mathcal{C}^V \times Mem \rightarrow Mem$  che descrive il comportamento dinamico dei comandi restituendo la memoria, risultato delle trasformazioni effettuate dai comandi eseguiti:

$$Exec(\langle c, \sigma \rangle) = \sigma' \Leftrightarrow \langle c, \sigma \rangle \rightarrow^* \sigma'$$

**Equivalenza di comandi:** l'equivalenza di comandi è una relazione  $\equiv \subseteq \mathcal{C}^V \times \mathcal{C}^V$  tale che

$$c_0 \equiv c_1 \Leftrightarrow \forall \sigma, Exec(\langle c_0, \sigma \rangle) = Exec(\langle c_1, \sigma \rangle)$$

## Blocchi

Abbiamo bisogno di un costrutto che permetta di identificare l'ambiente in cui un comando viene eseguito. Questo costrutto è il blocco.

```
{int tmp = x;  
  x=y;  
  y=tmp  
}
```

Un blocco è una regione testuale che può contenere dichiarazioni locali e comandi. L'ambiente di un blocco è l'ambiente valido al momento dell'esecuzione del blocco costituito dalle associazioni dichiarate localmente. L'ambiente quindi può cambiare durante l'esecuzione, e questi cambiamenti avvengono all'entrata e uscita da un blocco. Grazie al concetto di blocco possiamo avere identificatori che denotano oggetti diversi in regioni di codice diverse: i blocchi permettono una gestione locale dei nomi.

Con un'opportuna allocazione della memoria i blocchi permettono anche di ottimizzare l'occupazione della memoria, permettono la ricorsione e l'esecuzione di comandi in ambienti modificati.

Introduciamo i blocchi come anonimi – ovvero non possono essere chiamati da altre parti del programma. Questo è un costrutto puramente locale; solitamente sono delimitati da parentesi che ne delimitano lo scope.

I blocchi non possono essere parzialmente sovrapposti: o sono disgiunti o sono annidati.

Possono esserci vincoli di annidamento in funzione del linguaggio: ad esempio in C non possono esserci procedure dentro procedure.



## Scope

I blocchi, dunque, servono a stabilire ambienti locali a comandi – ovvero fissano la visibilità delle dichiarazioni a un insieme di comandi.

**Regola di visibilità:** Una dichiarazione locale ad un blocco è **visibile in quel blocco (scope) e in tutti i blocchi a esso annidati**, a meno che non intervenga una nuova dichiarazione dello stesso nome che maschera la precedente.

Quindi ogni definizione annidata genera un buco nello scope precedente. I binding annidati possono creare problemi di leggibilità e rendere più complesso il recupero del binding corretto, e possono complicare il debugging; ma permettono ai programmatori di usare i propri nomi indipendentemente da altre porzioni di codice.

**Scope:** si riferisce **all'area di testo del programma (codice) nel quale tutte le occorrenze applicate di un identificatore si riferiscono alla stessa occorrenza di binding dell'identificatore**. Permette di legare un identificatore al binding a tempo di compilazione cercandolo nel blocco che testualmente lo contiene.

Concludendo:

- Quando si entra in un blocco si creano associazioni locali e si disattivano le sovrapposizioni
- Quando si esce si distruggono le associazioni locali e si riattivano quelle al livello superiore.

### Tipi di ambiente dei blocchi

Lo scope di una variabile definita è quindi il range di comandi ai quali è visibile; e l'esistenza di variabili di tipo diverso rispetto alla regione in cui sono definite determina la caratterizzazione di diversi tipi di sotto-ambienti

<b>Variabili locali:</b> quelle dichiarati nel blocco stesso	→	<b>Ambiente locale:</b> Associazioni create all'ingresso nel blocco; variabili locali e parametri formali
<b>Variabili non locali :</b> visibili al blocco ma non dichiarate in esso (=dichiarate in un blocco che lo contiene)	→	<b>Ambiente non locale:</b> Associazioni ereditate da altri blocchi e visibili al blocco stesso
<b>Variabili globali:</b> Speciale categoria di variabili non locali, dichiarate nel blocco globale che contiene l'intero programma.	→	<b>Ambiente globale:</b> Parte di ambiente non locale relativo alle associazioni comuni a tutti i blocchi, ovvero dichiarazioni esplicite di variabili globali, dichiarazioni del blocco più esterno e associazioni esportate da moduli

### Tempo di vita

Non sempre le variabili esistono per tutta la durata del programma: hanno un certo tempo di vita.

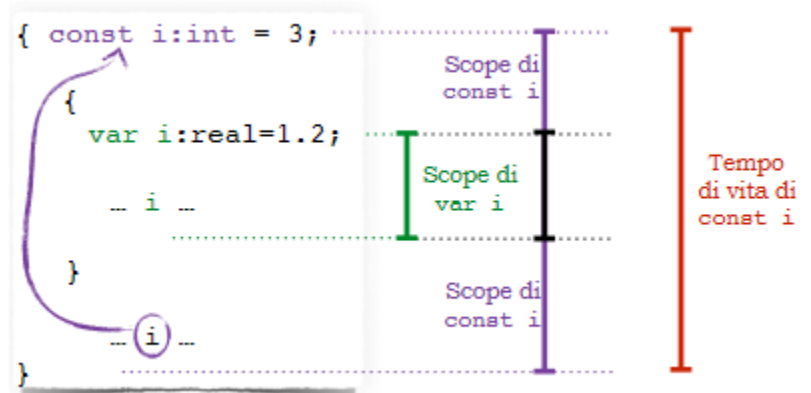
Il **tempo di vita** consiste nel tempo di esecuzione nel quale tutte le occorrenze applicate di un identificatore si riferiscono alla stessa locazione di memoria legata all'identificatore. Quindi il tempo di vita di una variabile inizia quando viene legata ad una cella (allocazione) e finisce quando il legame viene sciolto (deallocazione).

Scope e tempo di vita sono concetti vicini ma di natura diversa:

- **Scope** → concetto spaziale, definito a tempo di compilazione
- **Tempo di vita** → concetto temporale, definito a tempo di esecuzione

Il tempo di vita dipende dal tipo di allocazione degli identificatori, che a seconda del linguaggio può essere statico o dinamico. In particolare durante l'esecuzione il tempo di vita ci dice quando un identificatore è utilizzabile.

- Se allocazione dinamica, il tempo di vita inizia all'entrata in un blocco e termina all'uscita
- Se allocazione statica, termina alla fine dell'esecuzione del programma.



### Sintassi

**D;C**

### Identificatori liberi e definiti

Tutti gli identificatori liberi nella dichiarazione sono liberi nel blocco, mentre sono liberi gli identificatori del comando che non vengono definiti nella dichiarazione.

$$FI(d; c) = FI(d) \cup (FI(c) \setminus DI(d))$$

Gli identificatori definiti sono l'unione di quelli definiti nella dichiarazione e nel comandi.

$$DI(d; c) = DI(c) \cup DI(d)$$

#### Semantica statica

$\mathcal{C}_{S6}: \frac{\Delta \vdash_v d : \Delta' \quad \Delta[\Delta'] \vdash_{vvv'} c}{\Delta \vdash_v d; c} \Delta' \text{ su } v'$	<p>Dobbiamo generare un ambiente statico <math>\Delta'</math> associato alla dichiarazione <math>d</math>  Nell'ambiente statico <math>\Delta</math> del contesto aggiornato da <math>\Delta'</math> verifichiamo che il comando sia ben formato.</p>
---	---

#### Semantica dinamica

$\mathcal{C}_{11}: \frac{\rho \vdash_{\Delta} \langle d, \sigma \rangle \rightarrow_d \langle d', \sigma' \rangle}{\rho \vdash_{\Delta} \langle d; c, \sigma \rangle \rightarrow_c \langle d'; c, \sigma' \rangle}$	<p>Innanzitutto elaboriamo la dichiarazione, che può generare side effects nella memoria a causa della potenziale definizione di variabili. Poi eseguiamo il comando nell'ambiente della dichiarazione.</p>
$\mathcal{C}_{12}: \frac{\rho[\rho'] \vdash_{\Delta[\Delta']} \langle c, \sigma \rangle \rightarrow_c \langle c', \sigma' \rangle}{\rho \vdash_{\Delta} \langle \rho'; c, \sigma \rangle \rightarrow_c \langle \rho'; c', \sigma' \rangle} \rho' \vdash_{\Delta'}$	<p>Concetto di reversibilità: l'ambiente <math>\rho'</math> è temporaneo così come tutte le modifiche eseguite</p>
$\mathcal{C}_{13}: \frac{\rho[\rho'] \vdash_{\Delta[\Delta']} \langle c, \sigma \rangle \rightarrow_c \sigma'}{\rho \vdash_{\Delta} \langle \rho'; c, \sigma \rangle \rightarrow_c \sigma'} \rho' \vdash_{\Delta'}$	<p>No comment apparently</p>

Regole transitive:

$$\mathcal{C}_{11-12}: \frac{\rho \vdash_{\Delta} \langle d, \sigma \rangle \rightarrow_d^* \langle \rho', \sigma' \rangle}{\rho \vdash_{\Delta} \langle d; c, \sigma \rangle \rightarrow_c \langle \rho'; c, \sigma' \rangle}$$

$$\mathcal{C}_{13-12}: \frac{\rho[\rho'] \vdash_{\Delta[\Delta']} \langle c, \sigma \rangle \rightarrow_c^* \sigma'}{\rho \vdash_{\Delta} \langle \rho'; c, \sigma \rangle \rightarrow_c \sigma'} \rho' \vdash_{\Delta'}$$

## 6 - Procedure

L'uso delle procedure permette la cosiddetta astrazione del controllo. Astrarre significa nascondere dettagli concreti facendo emergere con più chiarezza concetti comuni. Abbiamo già sottolineato che i linguaggi sono una forma di astrazione della macchina fisica.

L'astrazione richiede di identificare le proprietà importanti di ciò che si vuole descrivere al fine di concentrarsi sulle questioni rilevanti, ed ignorare quelle non importanti. Le procedure sono anche uno strumento che permette la scomposizione di un problema in sottoproblemi, permettendo così di gestire la complessità.

### Astrazione

Nel processo di astrazione del controllo, quello che viene fatto consiste essenzialmente nel legare una sequenza di comandi – **corpo della procedura** – ad un identificatore – **nome della procedura**.

Mantenendo solo nome e corpo potremmo però associare solo esecuzioni che fanno sempre la stessa cosa ogni volta. Per rendere l'esecuzione dipendente da altri elementi abbiamo bisogno dei parametri, per poter svolgere la stessa computazione in contesti differenti.

- **Interfaccia:** Nomi e parametri, ovvero tutto ciò che è necessario conoscere per poter usare correttamente la procedura, mentre il corpo è proprio l'informazione che viene astratta
- **Corpo:** ciò che viene elaborato/valutato/eseguito ogni qualvolta il nome è richiamato, con appropriati parametri, in qualche punto del programma. È semplicemente analogo ai blocchi anonimi.

L'uso delle procedure si basa sulla chiamata del nome quando dobbiamo eseguire il suo corpo. Il processo di chiamata si basa su due passi:

- Il programma chiamante viene sospeso durante l'esecuzione del programma chiamato
- Il controllo ritorna sempre al chiamante quando l'esecuzione del risultato termina.

### Funzioni e procedure

*Sottoprogramma:* concetto base (funzione o procedura), porzione di codice identificato da un nome, dotato di un ambiente locale proprio e capace di scambiare informazioni con il resto del codice mediante canali fissati come parametri e valori di ritorno.

Quando si parla di astrazione, in realtà ci si riferisce a due categorie di sottoprogrammi:

Procedure	Funzioni
collezioni di comandi che definiscono una computazione parametrizzata. È una procedura ogni forma di astrazione di comandi che ha un nome senza necessariamente restituire il risultato, e che si limita a modificare lo stato	hanno una struttura simile alle procedure, ma sono semanticamente modellate come funzioni. Le funzioni, quindi, restituiscono un valore dopo aver eseguito modifiche di stato. In generale non dovrebbero produrre side effects (ma accade).

In entrambi i casi il processo di astrazione fa sì che, chiamata la funzione, possiamo ignorare i cambiamenti di stato intermedi dal punto di vista del chiamante, tenendo solo lo stato di chiamata che passa allo stato di ritorno.

→ Al contrario, se usassimo direttamente il codice anziché la funzione, siamo interessati a tutti i passaggi intermedi

Per caratterizzare un sottoprogramma dobbiamo definirlo (=specificarlo e scriverlo) e chiamarlo (poterlo usare).

## Visibilità e ambienti di riferimento

Ricordiamo il concetto di visibilità. Esso può essere esteso in presenza di procedure, cioè di blocchi eseguiti in posizioni diverse dalla loro definizione e in presenza di ambiente non locale e non globale.

In tal caso dobbiamo definire il concetto di ambiente di riferimento.

Ambiente di riferimento: l'ambiente di riferimento di un comando è la collezione di tutti i nomi che sono visibili al comando.

In alcuni linguaggi l'ambiente di riferimento può essere costituito dalle variabili locali più tutte le variabili negli scope più esterni; in altri, può essere l'insieme delle variabili locali più tutte quelle visibili nei sottoprogrammi attivi – ovvero la cui esecuzione è iniziata ma non terminata.

L'ambiente di riferimento di una procedura è determinato quindi da vari tipi di regole:

- Regole di scope statico e dinamico
- Regole specifiche (es. quando è visibile una dichiarazione nel blocco in cui compare?)
- Regole per il passaggio di parametri
- Regole di binding (shallow o deep)

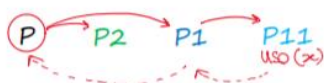
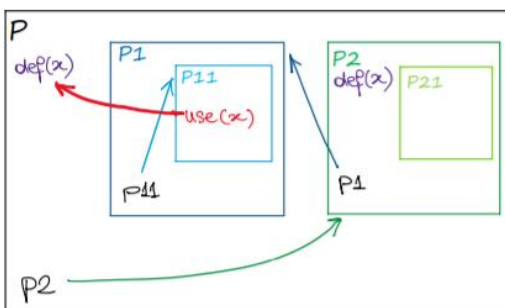
In particolare, le regole di scope e binding servono esclusivamente quando la procedura contiene un ambiente non locale

## Scope statico e dinamico

Le regole di scope possono essere distinte in due tipologie.

<b>Scope statico:</b> un nome non locale è risolto nel blocco che testualmente lo racchiude.	<b>Scope dinamico:</b> Un nome non locale è risolto nella chiamata attivata più di recente e non ancora terminata.
→ Pascal, ML Gli identificatori liberi della procedura sono staticamente legati all'ambiente, prima ancora che avvenga l'esecuzione. Si basa esclusivamente sul testo del codice e lega l'identificatore al blocco che lessicalmente lo contiene, ovvero in cui è stato dichiarato.	→ LISP, APL I legami non sono creati a tempo di esecuzione, ma dobbiamo eseguire il programma per individuarli e quindi i legami stessi sono dinamici. Si va a prendere quindi l'ultima chiamata aperta e non ancora chiusa. Ne consegue che diverse chiamate alla stessa procedura potrebbero essere eseguite in ambienti diversi.

### Scope statico



Nel caso dello scope statico, si risale la catena di annidamento dei blocchi seguendo la catena statica (nota a tempo di compilazione e mostrata tratteggiata a sx). Questo permette di indirizzare gli oggetti in funzione di un indirizzo relativo nel blocco, in cui la variabile è definita e della profondità del blocco.

Si creano tante catene statiche indipendenti che si incontrano all'origine, e in ogni istante solo una è percorribile dal punto di programma raggiunto.

In questo caso, usare variabili globali diventa molto costoso poiché devo seguire la catena fino all'origine.



Le caratteristiche sono:

- Garantisce l'indipendenza del risultato della procedura dalla posizione. Lo scope statico è l'unico dove foo può essere compilata in modo univoco, facendo riferimento, in entrambe le chiamate di foo, sempre alla x esterna.

```
{int x=10;
void foo () {
    x++;
}
void fie () {
    int x=0;
    foo();
}
fie();
```

- ! La visibilità dipende dal linguaggio. In certi linguaggi la dichiarazione è visibile in tutto il blocco che la contiene; potenzialmente quindi anche a comandi che compaiono sintatticamente prima (dichiarazione → uso funzione), mentre in altri solo se compare sintatticamente dopo (es. dichiarazione → uso variabile)

Svantaggi:

- Necessita di più accessi del necessario a variabili e sottoprogrammi
- Durante l'evoluzione del programma non tiene conto del fatto che la struttura iniziale viene distrutta, così come molte variabili locali
- Praticamente anche i sottoprogrammi diventano globali. (???)

### Esempi

Javascript

```
3 var n = 12;
4
5 function addn(){
6     return n + 30 ;
7 }
8
9 document.write(n+'\n');
10 var n = 17;
11 n = addn();
12
13 document.write(n);
```

Si stampa 12 e 47: la addn usa il valore 17. La seconda dichiarazione di n non è in un blocco, quindi semplicemente sovrascrive la prima dichiarazione di n.

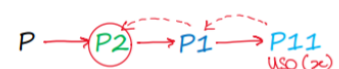
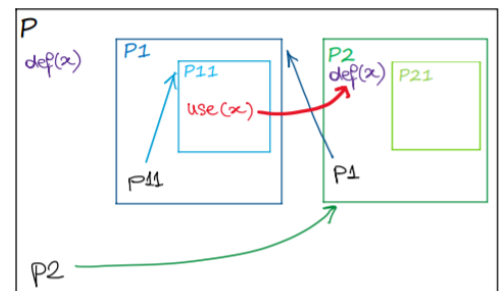
```
3 var n = 12;
4
5 function addn(){
6     return n + 30 ;
7 }
8
9 function set() {
10     var n = 17;
11     n = addn();
12     document.write(n+'\n');
13 }
14
15 //var n = 13;
16
17 set();
18
19 document.write(n);
```

42-12, ovvero la chiamata at addn() usa la prima dichiarazione di n, quella che vale al momento della definizione.

### Scope dinamico

In questo caso si risale la catena di chiamate dei blocchi seguendo la catena dinamica, nota solo a tempo di esecuzione; ovvero un nome non locale è risolto nella chiamata attivata più di recente e non ancora disattivata. Quindi lo scope dinamico è basato sulla sequenza di chiamate del programma e non sulla struttura del codice.

- Vantaggi: convenienza
- Svantaggi:



- Mentre un programma è in esecuzione le sue variabili sono visibili a tutti i sottoprogrammi che chiama
- Il riferimento di una variabile in un punto di programma può variare a seconda dell'esecuzione
- Impossibile da verificare staticamente
- Riduce la leggibilità: non posso determinare staticamente il tipo di una variabile

### Esempi

#### PERL

```
1 $n = 12;
2
3 sub addn() { return $n + 30};
4
5 sub set() {
6   $n = 17;
7   $n = addn();
8   print $n;
9 };
10
11 set();
12
13 print $n;
```

47 47: la dichiarazione dentro set() è globale quindi lo scope non interviene. Per vedere quale scope c'è bisogna dichiarare la variabile dentro set() come locale!

```
1 $n = 12;
2
3 sub addn() { return $n + 30};
4
5 sub set() {
6   local $n = 17;
7   $n = addn();
8   print $n;
9 };
10
11 set();
12
13 print $n;
```

47 12 → addn usa la dichiarazione locale valida al momento della chiamata di addn.

#### LISP

```
1 (defvar n 12)
2
3 (defun addn () (+ n 30))
4
5 (defun set ()
6   (let ((n 17))
7     (print (addn))))
8 )
9
10 (set)
11
12 (print n)
```

47 12: la chiamata ad addn usa il valore 17 per n, assegnato localmente in set() prima della chiamata di addn.

```
1 (defvar x 4)
2 (defvar z 3)
3
4 (defun f (y) (* x y))
5
6 (let ((x 5))
7   (print (+ (f z) x)))
8
9 (print x)
```

??

In sintesi:

- Scope statico: un nome non locale è risolto nel blocco che testualmente lo racchiude.
  - In tal caso l'informazione si deduce completamente dal testo del programma, ovvero le associazioni sono note a tempo di compilazione.
  - È più complesso ma più efficiente e soddisfa i principi di indipendenza.
  - Algol, C, Java
- Scope dinamico: un nome non locale è risolto nella chiamata attivata più di recente e non ancora disattivata.
  - In tal caso l'informazione è derivata dall'esecuzione; i programmi sono meno leggibili
  - Più semplice da implementare ma meno efficiente
  - Lisp, Perl

**Differiscono solo nel contesto delle procedure con presenza congiunta di ambiente non locale e non globale.** Questo significa che se non ci sono procedure (o se nelle procedure non c'è un ambiente non locale non globale) allora non ha senso distinguere i due metodi.

## Allocazione della memoria

Risolvere un riferimento significa tradurre o sostituire il nome con l'oggetto a cui si riferisce, attraverso la risoluzione del suo binding.

L'identificazione del legame corretto dipende dallo scope.

L'allocazione della memoria è come vengono allocati gli oggetti e dove vengono risolti i riferimenti non locali. Anche in questo caso abbiamo due casi:

<b>Allocazione statica:</b> memoria allocata a tempo di compilazione prima dell'inizio dell'esecuzione, i dati del programma vengono disposti in opportune zone di memoria dove rimangono fino alla fine	<b>Allocazione dinamica:</b> memoria allocata a tempo di esecuzione Si usano strutture dinamiche, che durante l'esecuzione vengono continuamente allocate e deallocate: pila e heap
---	--

### Allocazione statica

L'allocazione statica consiste nell'allocare lo spazio per tutti gli identificatori a tempo di compilazione. La memoria rimane quindi allocata per l'intera esecuzione, e il tempo di vita di tutti gli identificatori è l'intero programma.

È necessario sapere a priori quanta memoria si deve utilizzare.

- questo esclude a prescindere l'uso della ricorsione, poiché non possiamo sapere a priori quante volte verrà ripetuta una chiamata ricorsiva.
- Non ci possono essere blocchi annidati in quanto i riferimenti sono puramente statici: un oggetto ha un indirizzo assoluto che è mantenuto per tutta l'esecuzione del programma

Solitamente sono allocati staticamente:

- Variabili globali
- Variabili locali dei sottoprogrammi
- Costanti determinabili a tempo di compilazione
- Tabelle usate dal supporto a run-time (type checking, garbage collection...)

Esempi sono FORTRAN e COBOL, ma anche C e C++ con le variabili static, che hanno scope statico e sono locali alla funzione ma con tempo di vita dell'intera esecuzione del programma di cui fa parte. Ogni volta che un dato viene riutilizzato si trova nella stessa locazione, quindi non c'è nessuna creazione dinamica di dati → esecuzioni più veloci.

In presenza di allocazione statica, quindi, le procedure possono condividere la stessa memoria, avendo allocato a tempo di compilazione tutto quello di cui hanno bisogno.

```

SUBROUTINE ERROR(N)
    IF (N.LE.1) RETURN
yyy    CALL ERROR(N-1)
    PRINT N
    END
...
xxx    CALL ERROR(3)

```

Supponiamo legale:  
eseguiamolo nel modello  
di memoria statica



### Allocazione statica

L'allocazione dinamica consiste nel costruire un record di attivazione per ogni istanza di un sottoprogramma run-time, contenente le informazioni relative a tale istanza. Analogamente, ogni blocco ha un suo record di attivazione più semplice.

La pila è la struttura dati naturale per gestire i RdA, in quanto permette di implementare la struttura a blocchi dei programmi.

In questo modo si permette la ricorsione ma si aggiunge overhead per allocazione e deallocazione. In generale, anche un linguaggio senza ricorsione può usare la pila per risparmiare memoria.

### Record di attivazione

La gestione della pila è composta dalle fasi:

- Sequenza di chiamata
- Prologo
- Epilogo
- Sequenza di ritorno

Va osservato che l'indirizzo di un RdA non è noto a tempo di compilazione. Quindi, per gestire correttamente la pila, usiamo un puntatore detto Stack Pointer (SP) che punta al RdA del blocco attivo, ovvero alla cima dello stack.

Le informazioni contenute nella pila sono accessibili mediante l'uso di un offset rispetto allo SP, che è determinabile staticamente.

Le informazioni contenute nell'RdA sono:

- Link di controllo che puntano agli RdA precedenti nella catena
- Indirizzo di ritorno, ovvero indirizzo dell'istruzione da eseguire al termine della procedura
- Indirizzo del risultato, ovvero dove depositare il risultato se presente
- Parametri attuali della funzione
- Risultati intermedi calcolati durante l'esecuzione.

Puntatore di Catena Dinamica
Puntatore di Catena Statica
Indirizzo di Ritorno
Indirizzo del Risultato
Parametri
Variabili Locali
Risultati Intermedi

### Semantica delle chiamate e dei ritorni

La semantica della chiamata si basa sulle seguenti fasi da implementare:

- Metodi di passaggio di parametri
- Allocazione dinamica sullo stack delle variabili locali
- Salvataggio dello stato di esecuzione del programma chiamante
- Trasferimento del controllo e preparazione del ritorno
- Se è supportato l'annidamento di sottoprogrammi, deve essere gestito l'accesso a variabili non locali.

In queste fasi, quindi, abbiamo modifica al program counter, allocazione dell'RdA e la modifica del puntatore dello stack

Analogamente, anche la semantica del ritorno si basa su varie fasi:

- I parametri per valore e per valore-risultato devono restituire il loro valore
- Deallocazione dello stack per le variabili non locali
- Recupero dello stato di esecuzione del chiamante
- Ritorno del controllo al chiamante.

La collezione di link dinamici nello stack – validi in ogni punto dell'esecuzione – è detta **catena dinamica** o catena delle chiamate.

Dentro gli RdA è possibile accedere alle variabili locali attraverso il loro offset dall'inizio dell'RdA. Questo offset è detto `local_offset` della variabile locale e viene determinato a tempo di compilazione.

### Implementazione scope statico

Per lo scope statico esistono due principali implementazioni: catena statica e display. Noi guardiamo catena statica.

Il link statico è il puntatore all'RdA del blocco che contiene immediatamente il testo del blocco in esecuzione (??), e che dipende dall'annidamento statico.

- **CS – Catena statica:** catena di link statici che collegano un RdA a tutti i suoi antenati statici.

- **Link statico:** punta all'RdA antenato del sottoprogramma "A", ovvero quello che sintatticamente contiene A e viene detto generatore statico di A
- **SD - Profondità statica:** è un intero associato allo scope statico di ogni RdA, il cui valore è la profondità di annidamento della definizione della procedura corrispondente.

### Link statici

$  \begin{array}{c}  \left[ \begin{array}{c} B \\ \left[ \begin{array}{c} D \\ \left[ \begin{array}{c} E \end{array} \right] \end{array} \right] \end{array} \right] \\  A  \end{array}  $	<p>Dato l'annidamento a sinistra:</p> <p><math>Sd(A) = 0</math></p> <p><math>Sd(B) = Sd(C) = 1</math></p> <p><math>Sd(D) = Sd(E) = 2</math></p> <p>Se un sottoprogramma è annidato a livello k, allora la catena statica da quel sottoprogramma è lunga k.</p>
--	--

Per determinare il link statico del chiamato, è il chiamante (Ch) a determinare il link statico del chiamato usando le informazioni che ha a disposizione:

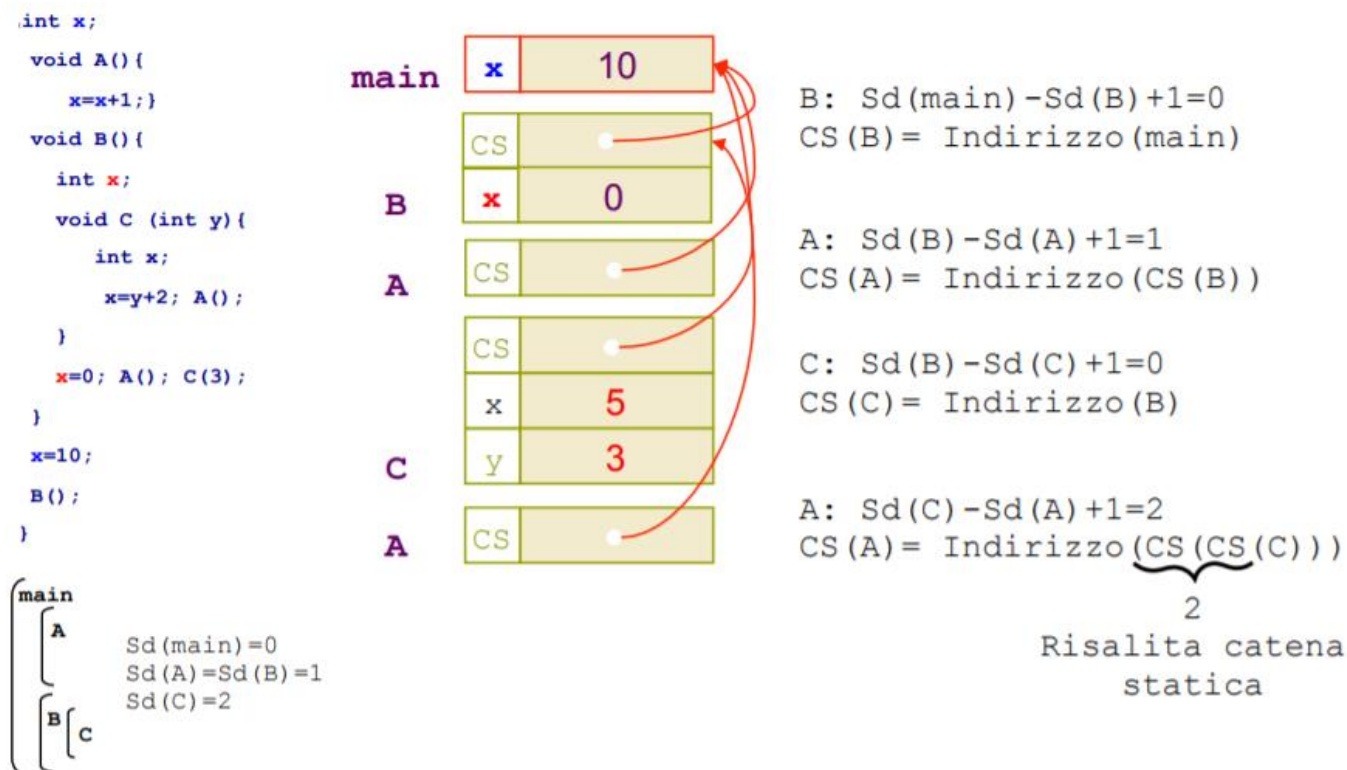
- Annidamento statico dei blocchi, determinato dal compilatore
- Indirizzo del proprio RdA

Ovvero il link statico si determina risalendo la catena statica del chiamante un numero di volte pari alla profondità di annidamento.

Quando una procedura chiamante Ch chiama una procedura P, Ch determina se la definizione di P è immediatamente inclusa in Ch o se è in un blocco che si trova k passi fuori da Ch. Il calcolo si fa nel seguente modo:

$k = Sd(Ch) - Sd(P) + 1$	<ul style="list-style-type: none"> <li>• Se <math>k = 0</math> allora il chiamante è il genitore statico <math>\rightarrow</math> passa il proprio indirizzo come link statico.</li> <li>• Se <math>k &gt; 0</math> allora Ch risale la propria catena statica di k passi e restituisce l'indirizzo RdA raggiunto come link statico di P.</li> </ul>
--------------------------	--

Esempio:



### Risolvere i riferimenti

Per ogni variabile sappiamo anche dove viene definita ogni variabile. Quindi, per ogni variabile sappiamo staticamente quali sono i sottoprogrammi che la definiscono. Quando usiamo la variabile  $x$  in un sottoprogramma  $P$ , possiamo determinare dalle informazioni note staticamente quale sia il sottoprogramma  $D$  più vicino che definisce  $x$ , e possiamo calcolare il numero di volte  $N$  in cui risalire la catena statica come

$$N = SD(P) - SD(D)$$

Ovvero, sapendo che per un identificatore  $x$  usato in  $P$  l'antenato statico più vicino che definisce  $x$  è  $D$ ,  $N$  ci dice che dobbiamo risalire  $N$  volte la catena statica per trovare esattamente l'RdA di  $D$ .

In questo modo, la ricerca di  $x$  non avviene tentando di trovare  $x$  in ogni antenato statico, ma trovando direttamente l'antenato corretto che contiene la giusta  $x$ . All'indirizzo raggiunto si somma l'offset dell'identificatore nel RdA raggiunto, per trovare la vera locazione da usare.

### Implementazione scope dinamico

La regola generale è semplice: l'associazione corrente per un nome è quella determinata epr ultima nell'esecuzione non ancora distrutta.

È chiaro che non possiamo usare informazioni statiche per sapere di quanto risalire, quindi ci servono strategie alternative quali:

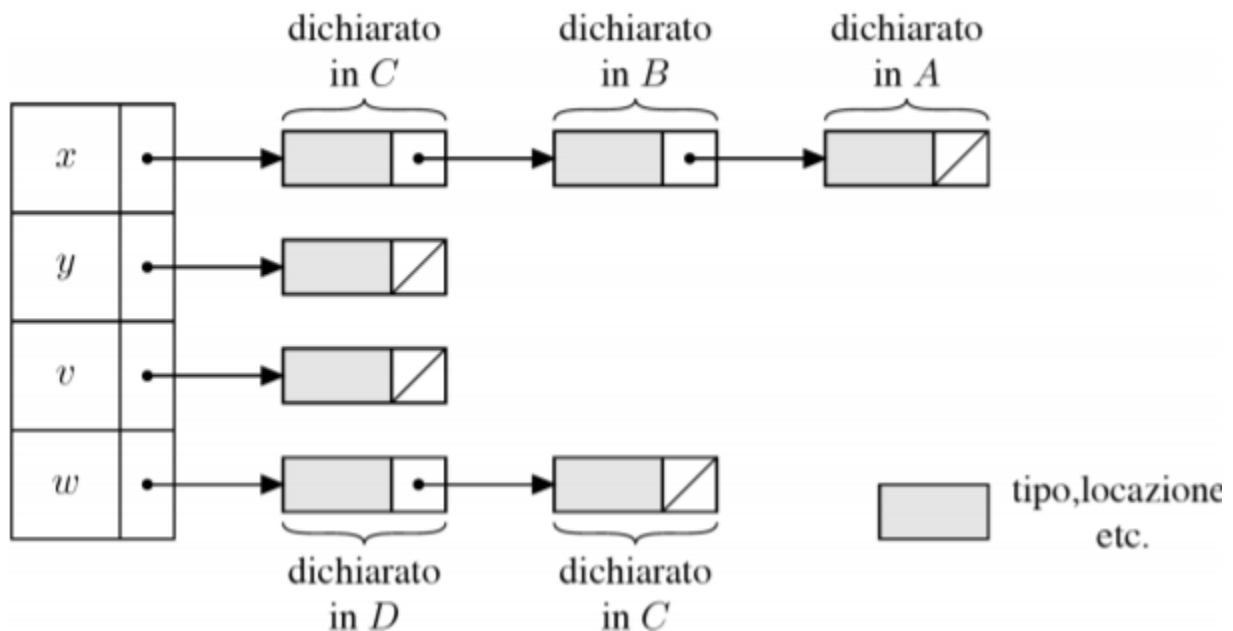
- **Deep access:** le variabili non locali vengono cercate negli RdA lungo la catena dinamica. I problemi sono evidenti:
  - La lunghezza della catena non può essere determinata, e ogni RdA deve avere i nomi delle variabili per poterli cercare risalendo la catena dinamica delle chiamate.

- **Shallow access:** in tal caso si pongono le variabili locali in strutture dati centrali, ovvero si costruisce uno stack per ogni nome di variabile e si mantiene una tabella centrale con una entry per ogni nome di variabile.
  - → CRT

### *CRT – tabella centrale dei riferimenti*

La CRT è una tabella con una entry per ogni variabile presente nel programma. Ogni entry punta ad una lista di elementi, ognuno dei quali contiene le informazioni necessarie per accedere a un possibile ambiente di riferimento per l'identificatore corrispondente alla entry.

Ogni entry punta ad una lista mantenuta in modo tale che la cima si trova sempre l'ultimo blocco che definisce l'identificatore



Ogni volta che aggiungo un RdA sullo stack, per ogni identificatore definito nella procedura chiamata, nella CRT si aggiunge in cima alla sua lista un elemento che fa riferimento all'ambiente della procedura chiamata. Questo significa che per ogni identificatore del programma l'ambiente di riferimento attivo è sempre quello in cima alla lista corrispondente all'identificatore nella CRT. All'uscita da una procedura vanno eliminati anche dalla CRT gli elementi in testa che si riferiscono alla procedura terminata.

Quindi, quando si usa l'identificatore, si accede alla testa della sua lista nella CRT e si recuperano le info necessarie per reperire nello stack l'ambiente di riferimento corretto.

Vantaggi:

- Evita le lunghe scansioni di liste della deep access
- Se i nomi delle variabili sono noti staticamente la tabella ha dimensione costante
- L'accesso è in tempo costante anche se con costi di gestione di ingresso e uscita.



## 6 – Procedure e parametri

### Procedure

Le procedure sono un processo di astrazione del controllo. Come abbiamo già detto, consistono nell'uso di un identificatore per far riferimento a una porzione di codice da eseguire ogni volta che si usa il riferimento.

Questo significa che tra identificatore e codice che questa rappresenta dobbiamo creare un'associazione, e questa associazione è analoga a quella degli identificatori costanti – ovvero associa direttamente il valore denotabile e che in questo caso è una procedura.

### Parametri

Permettono di usare la stessa computazione in contesti diversi, e inseriscono nella procedura un canale di comunicazione con l'ambiente della chiamata.

<p><i>Parametro formale</i>: usato nella definizione della procedura</p>	<p><i>Parametro attuale</i>: usato nella chiamata della procedura. Può essere sia r-value che l-value</p>
<pre>int f (int x){...}</pre> <p>parametro formale: <i>l-value</i></p>	<p><math>y = f(y + 2) \rightarrow</math> parametro attuale: <i>r-value</i>  <math>y = f(y) \rightarrow</math> può essere un <i>l-value</i>:          dipende dal metodo di passaggio dei parametri</p>

### Caratteristiche

#### Corrispondenza formale-attuale

- Posizionale**: è la più comune; il legame attuale-formale è dato dalla posizione; il primo parametro formale si lega al primo parametro formale e così via.
  - Dobbiamo comunque sapere l'interfaccia, ma ci basta l'ordine.
- Keyword**: il nome del parametro formale è usato nella lista dei parametri, per esplicitare a quale parametro vada associato.
  - Non dipende più dall'ordine, ma bisogna conoscere anche i nomi dei parametri. Eventualmente si può avere un parametro di default.

#### Comunicazione fra chiamato e chiamante

In-mode Passaggio per valore	In-out mode - Passaggio per valore-risultato - Passaggio per riferimento	Out-mode Passaggio per risultato
 <p>solo main <math>\rightarrow</math> procedura</p>		 <p>Solo main <math>\leftarrow</math> procedura</p>

#### Passaggio "fisico"

- Copiamo fisicamente il dato**: tipico del passaggio per valore
- Copiare un accesso al dato**: tipico del passaggio per riferimento.

## Passaggio per valore

Il valore del parametro attuale viene usato per inizializzare il parametro formale. → Il parametro formale è una variabile locale.

- In-mode: va solo dal chiamante al chiamato
- Le modifiche effettuate nella procedura ai valori restano locali alla procedura.
- Implementazione: si copia il valore  
→ Svantaggio: se ho dati di grandi dimensioni me li trovo in copia multipla. Potrei implementarlo tramite l'utilizzo di un cammino ma non è raccomandato poiché aumenterei lo spazio utilizzato e dovrei pure promettere questo link (non devo poterlo modificare globalmente).

## Passaggio per riferimento

Viene passato al parametro formale il cammino/indirizzo del parametro attuale.

- In-out: va in entrambi i sensi.
- Nessuna copia
- Effetti collaterali:
  - Creiamo un alias: due diversi identificatori agiscono sullo stesso oggetto
  - Collisioni

## Funzioni di ordine superiore

Alcuni linguaggi permettono di passare funzioni come argomenti di procedure, e di restituire funzioni come risultato.

Questo pone il problema di come gestire l'ambiente della funzione, e in particolare quello non locale:

- Caso semplice: **le funzioni sono passate come argomento**, e quindi è sufficiente considerare un puntatore al record di attivazione all'interno della pila, ovvero si passa un valore (detto chiusura) che è l'associazione tra nome e corpo della procedura.
- Caso complicato: la funzione viene ritornata da una chiamata a procedura. In tal caso occorre mantenere il record di attivazione della funzione restituita, e in tal caso la pila non funziona!  
NON LA TRATTIAMO!! :D

Un linguaggio che permette questo è Pascal. C, al contrario, permette di passare funzioni ma non di annidarle.

Ci sono tre dichiarazioni di x (non locale in f). Quale viene utilizzata quando f viene chiamata tramite h?

**! QUANDO SERVE PARLARE DI BINDING? In presenza di una funzione passata come parametro e con ambiente non locale, poiché le regole di scoping non sono più sufficienti per determinare l'ambiente di riferimento.**

- Se lo scope è statico, sicuramente quella più esterna. Tipicamente non cambia nulla.

- Se lo **scope è dinamico** bisogna specificare la politica di binding: Con scope dinamico, allora abbiamo due ambienti validi:

**Deep binding:** l'ambiente di riferimento è quello creato al momento della creazione del legame fra parametro formale e attuale → JS, python, PERL  
→ Qui è "g(f)" : legame tra h e f; quindi fa riferimento a x=5

**Shallow binding:** L'ambiente di riferimento è quello valido al momento della chiamata effettiva → LISP, PERL  
→ Qui è quando chiamo h(3) (che sarebbe f(3) come parametro attuale), quindi x=7

```
int x=4; int z=0;
int f (int y){
    return x*y;}
void g ( int h(int n) ) {
    int x;
    x = 7;
    z = h(3) + x ;
    end;
...
{int x = 5;
  g(f);
}
```

## Ricorsione

La ricorsione è un metodo alternativo di iterazione per ottenere il potere espressivo delle MdT; tuttavia, con la ricorsione non sappiamo quante iterazioni avremo e continuiamo ad allocare nuova memoria  
→ non può essere usato nei linguaggi a locazione statica poiché perdol'indirizzo di ritorno.

*Ricorsivo:*

```
int fatt (int n){
    if (n <= 1)
        return 1;
    else
        return n * fatt(n-1);
}
```

*Induttivo:*

```
fattoriale (0) = 1
fattoriale (n) = n*fattoriale(n-1)
```

In generale, una funzione (procedura) si dice ricorsiva se viene definita in termini di se stessa.

La differenza fra ricorsivo e induttivo è che l'induttivo deve garantire la convergenza, mentre per il ricorsivo è sufficiente garantire la modalità di definizione.

La ricorsione è possibile se:

- Il linguaggio che permette ad una funzione di chiamare sé stessa
- Il linguaggio usa allocazione dinamica della memoria.

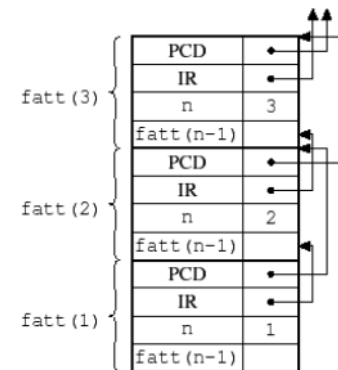
Normalmente, un programma ricorsivo può essere tradotto in un iterativo equivalente e viceversa; tuttavia i ricorsivi sono meno efficienti.

### Ricorsione in coda

Migliora l'efficienza della ricorsione, poiché permette di usare lo stesso RdA per tutte le chiamate ricorsive.

Una chiamata di g dentro f si dice *chiamata in coda* se f restituisce il valore ritornato da g senza ulteriori complicazioni.  
È una proprietà della chiamata.

Si ha *ricorsione in coda* quando la chiamata ricorsiva è una chiamata in coda.



L'idea è che se non faccio nulla non ho bisogno di ricordarmi nulla, ergo posso sovrascrivere il record di attivazione.

Non lo capisce il linguaggio da solo: è il programmatore che (in alcuni linguaggi) può specificarlo.

Per convertire qualunque funzione ricorsiva in ricorsiva in coda si usa il seguente trucchetto: fra i parametri che passo non c'è solo il valore  $n, n-1$  ma anche il risultato parziale!

```
int fattorc (int n, int res){
    if (n <= 1)
        return res;
    else
        return fattorc(n-1, n * res)
}
```

q

Il risultato calcolato din ora dentro res è quello parziale. Arrivare alla base significa che il risultato che mi sono portato dietro è quello definitivo

```
Fib(1) = 0;
Fib(1) = 1;
Fib(n) = Fib(n-1)+Fib(n-2)

int fibrc (int n, int res1, int res2){
    if (n == 0)
        return res2;
    else
        if (n == 1)
            return res2;
        else
            return fibrc(n-1, res2, res1+res2)
}

int fib (int n){
    if (n == 0)
        return 1;
    else
        if (n == 1)
            return 1;
        else
            return fib(n-1) + fib(n-2);
}
```

Complessità in tempo lineare in  $n$   
e in spazio costante (un solo RdA)

Complessità in tempo e spazio  
esponenziale in  $n$  (ad ogni  
chiamata due nuove chiamate)

## 7 - Paradigmi di programmazione

I principi di astrazione richiedono di trattare assieme i dati e le operazioni su di essi. Tuttavia i tipi di dati astratti sono tipi a cui l'utente può abbinare caratteristiche uniche, ed è possibile implementarli in IMP ma sono molto rigidi.

Una nuova prospettiva e nuovi supporti linguistici ci dà:

- Information hiding e incapsulamento
- Riutilizzo del codice (ereditarietà)
- Compatibilità fra tipi (sottotipi)
- Selezione dinamica delle operazioni

### Orientato agli oggetti

Sono caratterizzati da **ADT** (tipi di dati astratti), **ereditarietà** e **polimorfismo**.

#### Oggetti

I principi di organizzazione permettono di raggruppare gli oggetti con la stessa struttura - le classi - consentendo estensibilità e riutilizzo. Un oggetto è una capsula che contiene:

- **Dati nascosti:** variabili, valori, a volte operazioni
- **Operazioni pubbliche:** metodi
  - Le chiamate ai metodi sono dette messaggi
  - L'intera collezione di metodi di un oggetto è chiamata message protocol o message interface
  - I messaggi hanno due parti: un nome di metodo ed un oggetto destinazione.

#### Classi

Una classe è un modello di un insieme di oggetti, caratterizzato da nome, segnatura e visibilità dei metodi. Gli oggetti sono creati dinamicamente per istanziazione di una classe, mediante dei metodi costruttori.

#### Incapsulamento

Le classi garantiscono l'incapsulamento: ho opportuni modificatori che modificano la visibilità: pubblica (→ interfaccia), privata (→ implementazione), protetta

Gli altri meccanismi assicurano che l'incapsulamento sia compatibile con estensibilità e modificabilità del codice.

#### Ereditarietà

L'ereditarietà permette la definizione di nuove classi in termini di classi esistenti, ovvero permettendo che ereditino le parti comuni. Rinforza l'idea di riutilizzo delle ADT in caso di cambiamenti minimi, e la definizione di una gerarchia.

Una classe che eredita è detta classe derivata o sottoclasse.

La classe dalla quale un'altra eredita è una classe genitore o superclasse.

Le differenze classe-genitore possono essere:

- Nella classe parente, definizione di sue variabili o metodi con accesso privato, ovvero non visibili nelle sottoclassi
- La sottoclasse aggiunge variabili e/o metodi a quelli ereditati
- La sottoclasse modifica il comportamento di metodi ereditati.

## Funzionale

La progettazione dei linguaggi funzionali è **basata sulle funzioni matematiche**: è una solida base teorica, vicina all'utente ma lontana dall'architettura.

Una funzione è una mappa fra dominio e codominio. Un'**espressione lambda** specifica i parametri e la mappatura di una funzione senza nome.

Applicazione:

<i>Applicazione <math>\lambda</math></i>	<i>Operazione <math>\alpha</math> :</i>	<i>Composizione <math>h = f \circ g</math>:</i>
dice di sostituire il valore ad ogni occorrenza del nome della variabile. (Quindi lo stesso valore $x$ rappresenta sempre lo stesso valore!)	Prende una funzione come parametro e associa ad una lista di valori ottenuti applicando la funzione ad ogni elemento della lista di valori.	Prende due funzioni come parametri e restituisce una funzione il cui valore è il primo parametro applicato all'applicazione del secondo.
$\lambda x. x * x * x$	$h(x) = x * x$ $\alpha(h, (2,3,4)) = (4,9,16)$	$f(x) = x + 2, g(x) = 3 * x$ $h = f \circ g = (3 * x) + 2$

L'obiettivo della progettazione di un linguaggio funzionale è quello di **mimare le funzioni matematiche**. Questo implica che il progetto base di computazione è diverso da quello di un linguaggio imperativo: non serve gestire le variabili, poiché non si memorizzano.

- Non sono necessarie variabili: eseguo le operazioni, non ho bisogno di salvare risultati intermedi
- La valutazione di una funzione da sempre lo stesso risultato negli stessi parametri.

Questo tipo di linguaggio, benché turing-completo, nei linguaggi reali si integrano funzionalità imperative per accessibilità.

### LISP

È uno dei linguaggi funzionali più famosi;

- Tipi di dati: originariamente si basa su atomi (elementi) e liste  $(A B (C D) E)$

È un linguaggio funzionale, ergo si usa la notazione lambda per specificare funzioni. L'applicazione di funzioni e dati ha la stessa forma:

- se la lista  $(A B C)$  è interpretata come dato, è una semplice lista di tre atomi
- se è interpretata come applicazione di funzione, allora la funzione di nome  $A$  è applicata ai due parametri  $B$  e  $C$

### Scheme

LISP nasce come linguaggio per dimostrare la potenza del calcolo funzionale; nascono poi diversi dialetti.

SCHEME è un dialetto di LISP progettato per essere più chiaro.

- Solo scope statico  $\rightarrow$  abbiamo le variabili
- Le funzioni possono essere valori di espressioni o elementi di liste; passate a variabili o passate come parametri.

L'interprete Scheme è un ciclo infinito REPL (read-evaluate-print).

Le espressioni si valutano con la funzione EVAL e i letterali vengono valutati in sé stessi.

- Valutazione delle primitive:
  - **Read:** I parametri sono valutati senza un ordine particolare. I valori dei parametri sono sostituiti nel corpo della funzione
  - **Evaluate:** Il corpo della funzione viene valutato
  - **Print:** Il valore dell'ultima espressione della funzione è il risultato della funzione

### Sintassi

- $+$ ,  $-$ ,  $*$ ,  $/$ , *ABS*, *SQRT*, *REMAINDER*, *MIN*, *MAX* in forma prefissa → es.  $(+ 5 2)$  restituisce 7
- *LAMBDA* ( $x$ )( $* x x$ ) espressioni lambda.
- *DEFINE* elega simboli ed espressioni.  
Il processo di valutazione per *DEFINE* è diverso: il primo parametro non è mai valutato poiché è il nome. Il secondo viene valutato e legato al primo parametro.
  - Per legare un simbolo a una espressione. I simboli non sono variabili! Come se fossero final
  - Per legare un nome a una lambda espressione
- *PRINTF* Di solito non sono necessarie dato che l'interprete visualizza sempre il risultato di una funzione valutata al livello top. Comunque c'è *PRINTF*.  
→ Nota: input e output espliciti non
- *#T*, *#F* sono vero e falso
- $=$ ,  $<$ ,  $>$ ,  $<=$ ,  $>=$ , *NOT*, *EVEN?*, *ODD?*, *ZERO?*, *NEGATIVE?*, *LIST?* *member*
- *COND* è tipo il case dei linguaggi imperativi

Altri linguaggi:

- ML scope statico e fortemente tipato  
Haskell liste infinite e insieme  
Altri linguaggi permettono di includere anche parti funzionali

## Logico

I programmi nei linguaggi logici sono espressi tramite logica simbolica, e usano il processo di inferenza simbolica e per produrre risultati.

Sono dichiarativi: do solo la specifica del risultato.

- Proposizione: un comando logico può o non può essere vero; consiste di oggetti e relazioni tra oggetti.

Usiamo il linguaggio logico per:

- Esprimere proposizioni
- Esprimere relazioni tra proposizioni

- Descrivere come le nuove proposizioni possono essere inferite da altre

Gli oggetti sono rappresentati da

- Costanti: un simbolo che rappresenta un oggetto
- Variabile: un simbolo che può rappresentare diversi oggetti in momenti differenti.  
! Diverse dalle variabili dei linguaggi imperativi! È come quello del funzionale.
- Termini composti: sono le proposizioni atomiche. Un elemento di una relazione matematica scritto come funzione matematica  $\rightarrow$  mappa o tabella.  
È un predicato!  
È composto da
  - Funtore: simbolo di funzione che dà nome alla relazione
  - Lista ordinata di parametri (tupla)

Le proposizioni possono essere descritte come:

- Fatti: proposizioni assunte per vere
- Query: la verità della proposizione deve essere determinata

Le proposizioni composte hanno due o più proposizioni atomiche connesse da operatori.

### Clausole di Horn

Ci sono troppi modi di dire le stesse cose: quindi usiamo una forma standard per le proposizioni.

#### ☀ Clausole:

☀  $B_1 \cup B_2 \cup \dots \cup B_n \subset A_1 \cap A_2 \cap \dots \cap A_m$

☀ Significa che se tutte le proposizioni A sono vere, allora almeno una B è vera

#### ☀ Antecedente: lato destro

#### ☀ Conseguente: lato sinistro

Un uso delle proposizioni consiste nel coprire nuovi teoremi che possono essere inferiti da assiomi e teoremi noti. Risoluzione: un principio di inferenza che permette di computare proposizioni da proposizioni note.

La risoluzione si basa su:

- Unificazione: trovare valori per le variabili in proposizioni che permettono di portare a successo il processo di matching
- Istanziamento: assegnare valori temporanei alle variabili per permettere il successo dell'unificazione. Se fallisce si può fare backtracking con un valore differente.



## Prolog

Si fissano le ipotesi come clausole di horn senza antecedente.

Usiamo le regole delle clausole di horn: se sono vere tutte le clausole antecedenti allora è vero il termine singolo a destra.

I goal sono i teoremi da dimostrare.

Approcci:

- Bottom-up\_ inizia dai fatti e dalle regole per tentare di trovare una sequenza che porta al goal. Lavora bene con insiemi grandi di possibili risposte corrette.
- Top-down: inizia con il goal e tenta di trovare una sequenza che porta ad un insieme di fatti. Lavora bene con insiemi piccoli di soluzioni corrette. Lo usa prolog.

Tipicamente il goal è un and di sottogoal:

- DFS: dopo aver dimostrato del tutto il primo inizio il secondo.
- BFS: alora sui sotto goal in parallelo.
- Is: operatore che prende un'espressione aritmetica come operando destro e variabile come sinistro. NON è assegnamento: do solo un nome a un'espressione.

Carenze:

- ➔ No controllo dell'ordine di risoluzione
- ➔ Assunzione di un mondo chiuso: cose non descritte sono assunte false!!