Università degli Studi di Verona

Dipartimento di Informatica
Corso di Laurea in Informatica

Riassunto del corso di Linguaggi

Creato da:
Davide Zampieri

Anno Accademico 2019-2020

Indice

Capitolo 1

Introduzione

1.1 Tipi di linguaggio

- *Linguaggio matematico*: notazione molto rigorosa; può rappresentare funzioni ma non computazioni (ovvero passi di calcolo); non sempre permette rappresentazioni finite di oggetti infiniti; non sempre fornisce direttamente un metodo di calcolo di ciò che si vuole rappresentare.
- *Linguaggio logico*: è costituito da regole di inferenza e assiomi che rendono possibile specificare il processo di computazione (ancora in modo implicito); permette di rappresentare formalmente oggetti infiniti in modo finito (ma non computazioni infinite).
- *Linguaggio di programmazione*: permette di specificare in modo accurato esattamente le primitive del processo di computazione, con la rigorosità e la potenza della logica.

1.2 Contesti di programmazione

- Applicazioni scientifiche (es. Fortran).
- Applicazioni economiche (es. COBOL).
- Intelligenza artificiale (es. LISP).
- Programmazione di sistemi (es. C).
- Web Software (es. HTML, PHP, Java).

1.3 Aspetti di progettazione

- *Leggibilità (Readability)*: sintassi chiara, nessuna ambiguità, facilità di lettura e comprensione dei programmi.
- *Scrivibilità (Writability)*: facilità di utilizzo di un linguaggio per creare programmi, facilità di analisi e verifica dei programmi.

- *Affidabilità (Reliability)*: conformità alle sue specifiche.
- *Costo*: costo complessivo di utilizzo.

1.3.1 Leggibilità

Contribuiscono alla leggibilità:

1. *Semplicità*, in quanto la possibilità di poter fare la stessa cosa in più modi (es. x=x+1, x++, ...) e l'overloading degli operatori (cioè quando un singolo simbolo di operatore ha più significati) complicano i linguaggi.
2. *Ortogonalità*, in quanto più ortogonale è la progettazione di un linguaggio, meno eccezioni alle regole ci sono.
3. *Definizione dei tipi di dati.*
4. *Struttura della sintassi.*

1.3.2 Scrivibilità

Contribuiscono alla scrivibilità:

1. *Semplicità e ortogonalità.*
2. *Supporto per l'astrazione.*
3. *Espressività.*

1.3.3 Affidabilità

Contribuiscono all'affidabilità:

1. *Type checking.*
2. *Gestione delle eccezioni.*
3. *Assenza di aliasing*, in quanto la presenza di due o più metodi di riferimento per la stessa locazione di memoria è sicuramente un potenziale problema.

1.4 Classificazione dei linguaggi

I linguaggi si possono classificare, per *metodo di computazione*, in:

- A basso livello (*linguaggio binario*, *Assembly*).
- Ad alto livello
  - *Linguaggi imperativi*, che descrivono come concetto chiave la cella di memoria e in cui la variabile è la sua astrazione logica che viene modificata con assegnamenti in modo sequenziale, condizionale o ripetuto.

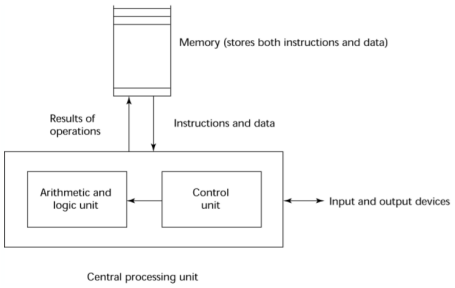
- *Linguaggi funzionali*, che descrivono i passi di calcolo come funzioni matematiche (compongono e applicano funzioni) e in cui le variabili non possono cambiare nel tempo.
- *Linguaggi logici*, che usano pattern matching e unificazione/sostituzione come passo di calcolo primitivo.

I linguaggi si possono inoltre classificare in base alle loro *caratteristiche* in:

- Sequenziali.
- Concorrenti.
- Modulari.
- Paralleli/distribuiti.
- Orientati agli oggetti/classi.
- Interpretati/scripting.

1.5 Implementare linguaggi

Un *linguaggio di programmazione* **L** è un insieme di costrutti e regole per descrivere algoritmi e dati. Un *programma* è un insieme finito di istruzioni/costrutti del linguaggio di programmazione.



(a) Architettura di von Neumann

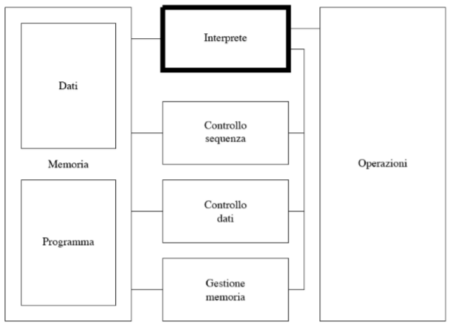
(b) Ciclo fetch-execute su una macchina di von Neumann

1.5.1 Macchina astratta

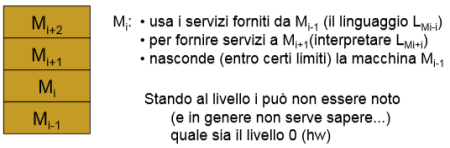
Implementare un linguaggio significa realizzare la *macchina astratta* che interpreta il linguaggio. Dato un linguaggio **L** di programmazione, la macchina astratta $\mathbf{M_L}$ per **L** è un insieme di strutture dati ed algoritmi che permettono di memorizzare ed eseguire i programmi scritti in **L**.

Quindi la macchina astratta è la combinazione di una *memoria* che immagazzina i programmi e di un *interprete* che esegue le istruzioni dei programmi.

Data una macchina astratta **M**, il linguaggio $\mathbf{L_M}$ che ha come stringhe legali tutte le stringhe interpretabili da **M** è detto *linguaggio macchina*. Infine, le macchine astratte possono essere realizzate a livello HW, FW o SW.



(a) Macchina astratta



(b) Struttura a livelli cooperanti

1.6 Come realizzare la macchina astratta

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

Realizzare $\mathbf{M_L}$ consiste quindi nel realizzare una macchina che "traduce" $\mathbf{L}$ in $\mathbf{L_O}$, ovvero che interpreta tutte le istruzioni di $\mathbf{L}$ come sequenza di istruzioni di $\mathbf{L_O}$. Possiamo distinguere due modalità:

- *Soluzione interpretativa*, cioè tramite traduzione "implicita" realizzata dalla simulazione dei costrutti di $\mathbf{M_L}$ (ovvero di $\mathbf{L}$) mediante programmi scritti in $\mathbf{L_O}$.
- *Soluzione compilativa*, cioè tramite traduzione "esplicita" dei programmi di $\mathbf{L}$ in corrispondenti programmi di $\mathbf{L_O}$.

1.6.1 Soluzione interpretativa: interprete

Un interprete è un programma $\mathbf{int^{L_O,L}}$ che esegue, sulla macchina astratta per $\mathbf{L_O}$, il programma $\mathbf{P^L}$ con input $\mathbf{d} \in \mathbf{D}$. Data la seguente notazione:

- $\mathbf{Prog^L}$ è l'insieme di programmi scritti in $\mathbf{L}$
- $\mathbf{D}$ è l'insieme di dati (input e output)
- Se $\mathbf{P^L} \in \mathbf{Prog^L}$ e $\mathbf{in,out} \in \mathbf{D}$ allora $\mathbf{[P^L]: D \to D}$ è la semantica di $\mathbf{P^L}$ tale che $\mathbf{[P^L](in) = out}$, ovvero se $\mathbf{P^L}$ viene eseguito a partire da $\mathbf{in}$ restituisce $\mathbf{out}$

l'interprete $\mathbf{int^{L_O,L}}$ da $\mathbf{L}$ a $\mathbf{L_O}$ è un programma tale che

$$[int^{L_O,L}] : (Prog^L \times D) \to D$$

e

$$[int^{L_O,L}](P^L, in) = [P^L](in)$$

Un interprete è di fatto un ciclo che, attraverso una *serie di operazioni*, esegue per simulazione le istruzioni del linguaggio.

Tali operazioni sono:

- *Elaborazione dei dati primitivi.*
- *Controllo di sequenza delle esecuzioni* (strutture dati manipolate con operazioni specifiche).
- *Controllo dei dati.*
- *Controllo della memoria.*

1.6.2 Soluzione compilativa: compilatore

Un compilatore è un programma $\mathbf{comp^{L_O,L}}$ che traduce programmi scritti in $\mathbf{L}$ in programmi scritti in $\mathbf{L_O}$, e quindi eseguibili direttamente sulla macchina astratta per $\mathbf{L_O}$. Il compilatore $\mathbf{comp^{L_O,L}}$ da $\mathbf{L_O}$ a $\mathbf{L}$ è un programma tale che

$$[comp^{L_O,L}] : Prog^{L_O} \rightarrow Prog^{L}$$

e

$$[comp^{L_O,L}](P^L) = P^{L_O} \text{ tale che } [P^{L_O}](in) = [P^L](in)$$

L'esecuzione di un compilatore passa attraverso varie fasi:

- *Analisi lessicale (scanner)*, che spezza un programma nei componenti sintattici primitivi chiamati tokens (i quali formano linguaggi regolari).
- *Analisi sintattica (parser)*, che crea una rappresentazione ad albero della sintassi del programma dove le foglie lette da sinistra a destra costituiscono frasi ben formate del linguaggio (le quali formano linguaggi CF).

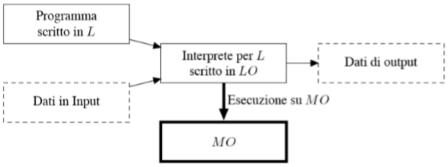
1.6.3 Interprete vs Compilatore

L'*implementazione interpretativa pura* comporta:

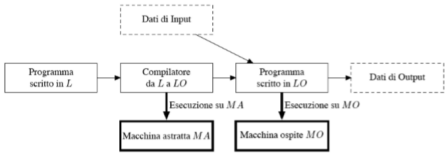
- Nessun costo di traduzione.
- Esecuzione lenta.
- Scarsa efficienza della macchina $\mathbf{M_L}$.
- Buona flessibilità e portabilità.
- Facilità di interazione a run-time (es. debugging).

L'*implementazione compilativa pura* comporta:

- Costi di traduzione.
- Esecuzione veloce.
- Buona efficienza.
- Scarsa flessibilità.
- Debugging più difficile.



(a) Implementazione interpretativa



(b) Implementazione compilativa

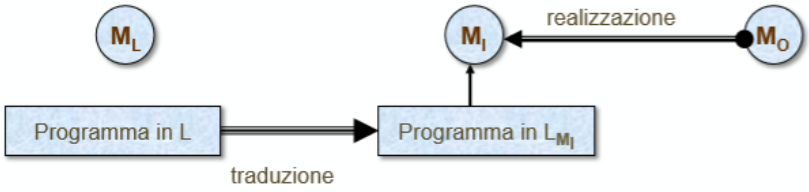
1.6.4 Soluzione reale: ibrida

Nella realtà, il linguaggio ad alto livello viene *compilato* in un linguaggio a più basso livello che poi viene *interpretato*.

Consideriamo il linguaggio **L**, ad alto livello, per il quale dobbiamo realizzare la macchina astratta $\mathbf{M_L}$: **L** viene tradotto in un linguaggio intermedio $\mathbf{L_{M_I}}$ la cui macchina astratta $\mathbf{M_I}$ consiste in un interprete del linguaggio $\mathbf{L_{M_I}}$ sulla macchina ospite $\mathbf{M_O}$.

In sintesi, nell'evoluzione dei linguaggi di programmazione esistono tre situazioni possibili:

- *Interprete puro*, quando $\mathbf{M_L} = \mathbf{M_I}$ e quindi l'interprete di **L** è realizzato su $\mathbf{M_O}$.
- *Compilatore puro*, quando la macchina intermedia $\mathbf{M_I}$ è realizzata per estensione sulla macchina ospite $\mathbf{M_O}$.
- *Implementazione mista*, quando si traducono i programmi da **L** a $\mathbf{L_{M_I}}$ e i programmi scritti in $\mathbf{L_{M_I}}$ sono interpretati su $\mathbf{M_O}$.



1.6.5 Specializzazione

Lo specializzatore *trasforma programmi*, cioè valuta il programma su una parte dell'input, ottenendo un programma specializzato rispetto a tale input e, per questo, più efficiente. Uno specializzatore $\mathbf{spec^L}$ per **L** è un programma tale che

$$[spec^L] : (Prog^L \times D) \rightarrow Prog^L$$

e

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

Specializzando un interprete rispetto al programma otteniamo un compilatore, in quanto: se $\mathbf{spec^{L'}}$ è uno specializzatore per **L'** e $\mathbf{int^{L,L_O}}$ è un interprete da $\mathbf{L_O}$ a **L** scritto in **L'**, allora $\mathbf{P^{L'} = [spec^{L'}](int^{L,L_O},P^L)}$; infatti $\mathbf{P^{L'}}$ eredita l'algoritmo di $\mathbf{P^L}$ e ne esegue fedelmente le operazioni nello stesso ordine, ma con lo stile di programmazione di $\mathbf{int^{L,L_O}}$.

Capitolo 2

Descrivere i linguaggi

La descrizione di un linguaggio avviene su queste *dimensioni*:

- Sintassi.
- Semantica.
- Pragmatica.
- Implementazione.

2.1 Sintassi

2.1.1 Terminologia nella linguistica

Una *parola* è una stringa di caratteri su un alfabeto, mentre una *frase* è una sequenza (ben formata) di parole. Un *linguaggio*, quindi, è un insieme di frasi. Nell'ambito dei *linguaggi di programmazione*, le parole diventano *lessemi* e le frasi diventano *token*. Inoltre, la frase diventa *programma* quando appartiene alla categoria sintattica dei comandi.

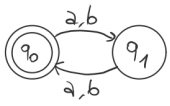
2.1.2 Descrivere la sintassi

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

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

$L = \{\sigma \in \Sigma^* \mid |\sigma| \text{ È PARI}\}$

$\Sigma = \{a, b\}$



$S \to \varepsilon \mid aaS \mid abS \mid baS \mid bbS$

$abbb \Rightarrow S \to abS \to abbbS \to abbb$
$abbbbb \Rightarrow NO$

(a) Esempio di linguaggio regolare

$L = \{\sigma \in \Sigma^* \mid \sigma \text{ È PALINDROMA}\}$
$\Sigma = \{a, b\}$
NON ESISTE UN AUTOMA $\Rightarrow G = \langle \{S\}, \Sigma, P, S \rangle$
$P \Rightarrow S \to \varepsilon \mid a \mid b \mid aSa \mid bSb$

(b) Esempio di linguaggio CF

2.1.3 Grammatiche context-free

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

- **V** è un insieme finito di variabili (dette anche simboli *non terminali*).
- **T** è un insieme finito di simboli *terminali* ($V \cap T = \varnothing$).
- **P** è un insieme finito di *produzioni*, in cui ogni produzione è nella forma $A \to \alpha$ dove:
  - $A \in V$ è una variabile.
  - $\alpha \in (V \cup T)^*$.
- **S** è una variabile speciale detta *simbolo iniziale* ($S \in V$).

Nella grammatica: i simboli non terminali sono le *categorie sintattiche* (le quali rappresentano i diversi tipi di elementi che possono essere usati per comporre frasi); i simboli terminali costituiscono il *vocabolario* (ovvero le parole del nostro linguaggio); le produzioni sono le *regole di composizione* degli elementi in una frase; il simbolo iniziale rappresenta la *categoria delle frasi* legali nel linguaggio.

Context-free è un vincolo sulle produzioni, infatti, a sinistra possiamo trovare un solo non-terminale; ciò implica che in una frase, dovunque troviamo quel terminale, ovvero in *qualunque contesto* troviamo quel terminale, lo possiamo sostituire/rimpiazzare applicando una delle produzioni per il non terminale.

Il *vantaggio* delle grammatiche CF è che esistono strumenti di parsing automatici ed efficienti. Lo *svantaggio* è che tali grammatiche non riescono a catturare vincoli contestuali.

$G = \langle \{E\}, \{or, and, not, (,), 0, 1\}, P, E \rangle$

$P \Rightarrow E \rightarrow 0 \mid 1 \mid (E \text{ or } E) \mid (E \text{ and } E) \mid (\text{not } E)$

(not((0 or 1) and 1))

$E \rightarrow (\text{not } E)$
$E \rightarrow (\text{not } (E \text{ and } E))$
$E \rightarrow (\text{not } (E \text{ and } 1))$
$E \rightarrow (\text{not } ((E \text{ or } E) \text{ and } 1))$
$E \rightarrow (\text{not } ((E \text{ or } 1) \text{ and } 1))$
$E \rightarrow (\text{not } ((0 \text{ or } 1) \text{ and } 1))$

2.1.4 Notazione BNF

La *BNF* è un metalinguaggio per descrivere linguaggi di programmazione. È stata introdotta a partire da *Algol60*. Questa notazione è usata per descrivere grammatiche CF, dove: si usano intere parole come simboli terminali; i non terminali sono identificati racchiudendoli tra parentesi angolari; per le produzioni si usa il simbolo ::= al posto della freccia. Inoltre, esistono le seguenti *opzioni*:

- Parentesi quadre [ ], che indicano 0 o 1 occorrenza di quanto contenuto.
  Esempio:
  $$< A > ::= \alpha < B > \gamma \mid \alpha$$
  $$< B > ::= \epsilon \mid \beta_1 \mid \beta_2$$

  diventa

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

- Parentesi graffe { }, che indicano 0 o più occorrenze di quanto contenuto.
  Esempio:
  $$< A > ::= \alpha < B > \gamma \mid \alpha$$
  $$< B > ::= \epsilon \mid \beta < B >$$

  diventa

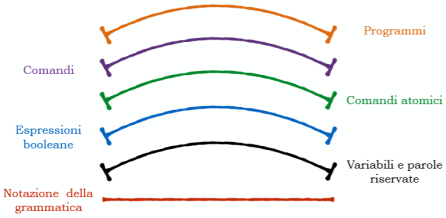
  $$< A > ::= \alpha\{\beta\}\gamma \mid \alpha$$

2.1.5 Un semplice linguaggio imperativo

Un linguaggio può essere descritto anche in modo informale, descrivendo quali *caratteristiche* vogliamo inserire e quale è il loro significato. Per esempio:

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

Quindi, ogni linguaggio è descritto a più livelli di astrazione ognuno dei quali è a sua volta descritto da una grammatica.



(a) Livelli di astrazione

```
<program>      S → C
<com>          C → A | C ; C | if B then C else C
                 | while B do C
<atomic com>   A → v := B
<b-expr>       B → true | false | v | ( not B )
                 | ( B and B ) | ( B or B )
```

(b) Grammatiche

2.2 Semantica

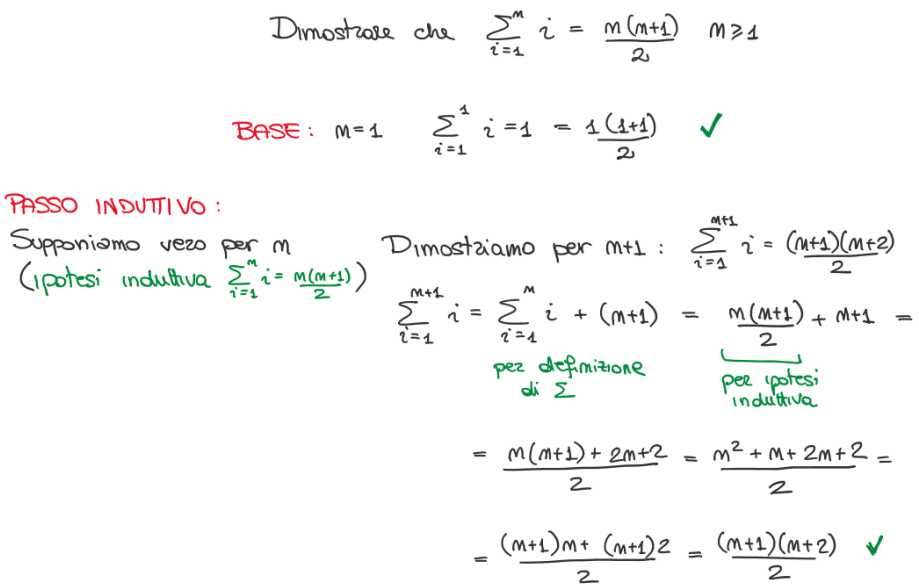
La *semantica* è più complessa della sintassi perché ricerca esattezza e flessibilità. La semantica, infatti, attribuisce un significato ad ogni frase sintatticamente corretta. I *significati* sono entità autonome che esistono indipendentemente dai segni che usiamo per descriverle. La semantica del linguaggio di programmazione è anche chiamata semantica *dinamica*, per distinguerla da quella *statica* (cioè i vincoli semantici). Quindi, per ogni costrutto del linguaggio 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 e indipendente dalla macchina.

2.2.1 Induzione

Abbiamo visto che la semantica va data seguendo la struttura della sintassi. La sintassi è definita descrivendo gli elementi base e componendo questi elementi (attraverso regole) in elementi composti. Questa forma di definizione ha una connotazione ben precisa nella matematica, ed è chiamata *induzione*. Dato un insieme A ed una relazione binaria $< \subseteq A \times A$ ben fondata (cioè senza catene discendenti infinite):

- Se $A = \mathbb{N}$ si ha *induzione matematica*.
- Se $A = L(G)$, dove $L(G)$ è un linguaggio generato da una grammatica G, si ha *induzione strutturale*.

Secondo il *principio di induzione strutturale*, per dimostrare che una proprietà è valida per tutti gli elementi di una categoria sintattica, per prima cosa si dimostra la proprietà per tutti gli *elementi base* della categoria (base induttiva), e poi si dimostra la proprietà per tutti gli *elementi composti* assumendo che la proprietà sia verificata da tutti i loro componenti immediati (ipotesi induttiva).



2.2.2 Descrivere i significati

Per capire come la semantica descrive il significato, dobbiamo utilizzare i seguenti *punti di vista*:

- *Comportamento I/O*, che interessa l'implementatore.
- *Funzione descritta dall'algoritmo*, che interessa il progettista.
- *Proprietà invarianti*, che interessa lo sviluppatore.

Quindi, a seconda di quali strumenti matematici usiamo per descrivere i significati, otteniamo tipi di semantiche diverse:

- *Semantica denotazionale*, che descrive funzionalità.
- *Semantica assiomatica*, che descrive proprietà.
- *Semantica operazionale*, che descrive trasformazioni di stato.

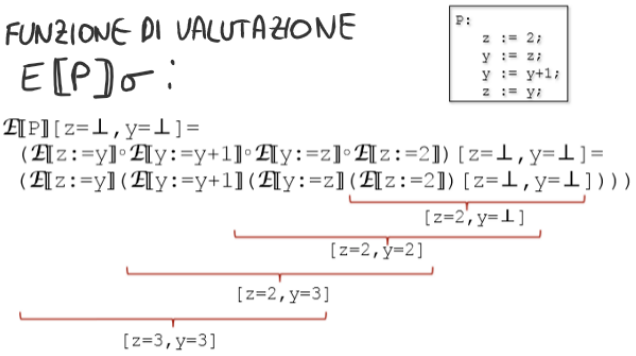
2.2.3 Semantica denotazionale

Si tratta di un modello matematico basato sulla *ricorsione*, ed è la semantica più "*astratta*" con cui descrivere i programmi. Il processo di costruzione della semantica denotazionale per un linguaggio consiste nel definire un oggetto

matematico per ogni entità del linguaggio e nel definire poi una *funzione* che mappa istanze delle entità del linguaggio in istanze dei corrispondenti oggetti matematici. Formalmente, il modello matematico usato per descrivere la semantica denotazionale è quello delle funzioni matematiche (ricorsive), ovvero un programma corrisponde ad una funzione del tipo

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

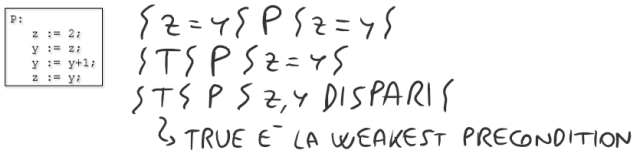
e l'*equivalenza di programmi* si dimostra mediante *uguaglianza tra funzioni*.



2.2.4 Semantica assiomatica

Si tratta di un modello matematico basato sulla *logica formale* (calcolo dei predicati). La semantica assiomatica, infatti, nasce con l'obiettivo di fare *verifica formale di programmi*. Le espressioni logiche della semantica sono chiamate *asserzioni*, le quali possono essere:

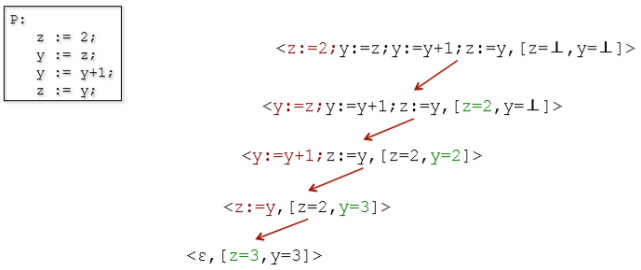
- *Precondizioni*, cioè asserzioni che precedono un comando e che dichiarano le relazioni e i vincoli validi prima dell'esecuzione del comando stesso.
- *Postcondizioni*, cioè asserzioni che seguono il comando.
- *Weakest precondition*, cioè la precondizione meno restrittiva che garantisce la postcondizione.



2.2.5 Semantica operazionale

Si tratta di un modello matematico basato sui *sistemi di transizione* (come i risultati finali vengono calcolati). La semantica operazionale, infatti, descrive il significato dei programmi come *trasformazioni di stato*.

Si consideri la funzione *memoria* $\sigma : Var \to Val$ che associa valori alle variabili. Lo *stato* è una coppia $< P, \sigma >$ che consiste nel programma ancora da eseguire e nello stato in cui si deve eseguire il programma. Ad ogni passo si esegue un'operazione e si cambia stato applicando una *relazione tra stati* (o relazione di transizione) $\to \subseteq < P, \sigma > \times < P, \sigma >$. Tale semantica opera eseguendo i comandi separati da **;** sequenzialmente e nell'ordine in cui compaiono da sinistra a destra.



2.2.6 Composizionalità

La composizionalità è una *proprietà* della semantica necessaria per caratterizzare i comportamenti e i significati di sistemi che possono avere infiniti elementi. Si noti che la semantica *denotazionale* e la semantica *assiomatica* rispettano banalmente tale principio. La composizionalità della semantica *operazionale* è invece meno immediata.

Tale proprietà dice che il significato di ogni programma deve essere funzione del significato dei costituenti immediati. Infatti, se la semantica su cui si basa una analisi è composizionale, allora è possibile analizzare il software separatamente nei suoi moduli, per poi ricomporre il risultato dell'analisi componendo i risultati ottenuti sui singoli moduli.

2.2.7 Equivalenza

Due programmi sono equivalenti quando hanno la *stessa semantica* (funzionalità, esecuzione, proprietà). L'equivalenza serve a varie *fasi di analisi*:

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

2.3 Categorie sintattiche

Dal punto di vista formale, le categorie sintattiche sono i simboli *non terminali* della grammatica e servono per classificare i costrutti in funzione del loro significato atteso, ovvero della classe di effetti che la loro esecuzione causa.

Possiamo identificare tre fondamentali *effetti* derivanti dall'esecuzione di un programma: generazione di *valori*; creazione di *legami*; trasformazioni di *stato*.

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. Partendo da questo otteniamo esattamente tre *categorie*: espressioni, comandi e dichiarazioni.

2.3.1 Stato

Lo *stato* è composto da due entità: ambiente e memoria. L'*ambiente* (environment) è l'insieme dei legami (bindings) tra identificatori e denotazioni. La *memoria* (store) è l'insieme degli effetti sugli identificatori (causati dagli assegnamenti). Quindi, l'ambiente specifica quali nomi sono usati e per quali oggetti, mentre la memoria fornisce un binding tra locazione e valore.

2.3.2 Espressioni

Le espressioni *denotano valori* e devono essere *valutate* per restituire un valore. Due espressioni sono *equivalenti* se vengono valutate nello stesso valore in tutti gli stati di computazione (anche eventuali side-effects devono essere gli stessi). Ad esempio, **not(a and b)** è logicamente equivalente a **(not a) or (not b)** anche se le due espressioni sono sintatticamente diverse.

2.3.3 Dichiarazioni

Le dichiarazioni *denotano richieste di modifica/creazione di legami* associati agli identificatori, ovvero gli ambienti. Le dichiarazioni devono essere *elaborate* per attuare le modifiche agli ambienti, che sono *trasformazioni reversibili*. Due dichiarazioni sono *equivalenti* se producono lo stesso ambiente (e la stessa memoria in caso di side-effects) in tutti gli stati di computazione.

2.3.4 Comandi

I comandi *denotano richieste di modifica della memoria*. I comandi rappresentano funzioni di trasformazione e devono essere *eseguiti* per attuare le modifiche della memoria, che sono *trasformazioni irreversibili*. Due comandi sono *equivalenti* se per ogni stato (memoria) in input producono lo stesso stato (memoria) in output.

2.3.5 Riassunto

Le *categorie sintattiche* dei linguaggi di programmazione sono:

- *Espressioni*, che vengono valutate in valori e non modificano lo stato di computazione e gli ambienti.
- *Dichiarazioni*, che vengono elaborate per creare/modificare ambienti/-legami (modifiche reversibili) e non producono valori né modificano la memoria.
- *Comandi*, che eseguono modifiche irreversibili della memoria e non producono valori né modificano ambienti.

2.4 Sistemi di transizione

La semantica deve essere *specificata* in funzione della sintassi. Il modo migliore per descrivere la semantica è attraverso la *manipolazione di simboli*. Lo strumento formale utilizzato è quello dei *sistemi di transizione*, i quali hanno le seguenti caratteristiche:

- Sono *matematicamente precisi*.
- Sono molto *concisi*.
- Sono un *metodo di specifica generale* che permette *astrazione*.
- Sono espressi mediante una *collezione di regole* date in funzione della sintassi (*induttivamente*), cioè *specificano cosa viene calcolato* tramite induzione sulla struttura sintattica del linguaggio.

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

$\gamma \rightarrow \gamma'$ per $\langle \gamma, \gamma' \rangle \in \rightarrow$

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

(a) Notazione

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

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

(b) Generazione di $(\Gamma, \rightarrow)$ a partire da una grammatica CF

2.5 PL0 - un semplice linguaggio reale

Le *caratteristiche* di PL0 (un linguaggio *Pascal-like*) sono:

- *Singolo tipo di dato* INTEGER (i Booleani vengono rappresentati come interi).
- *Strutture di controllo standard* (if, while).
- *Astrazione del controllo* (procedure senza parametri).

La *sintassi* di PL0 dice che:

- I programmi **P** sono blocchi.
- I blocchi **B** sono una dichiarazione **D** seguita da un comando **S**.
- Le dichiarazioni **D** sono dichiarazioni di valori costanti con nome (const), dichiarazioni di variabili (var) e dichiarazioni di procedure (ovvero di blocchi riferibili con un nome); inoltre, le produzioni di **D** sono tutte opzionali (un programma può essere un blocco con nessuna dichiarazione).

- I comandi **S** possono essere il comando vuoto, l'assegnamento di un valore (denotato da una espressione) ad un identificatore, la chiamata ad una procedura mediante il suo nome, il comando condizionale che esegue un comando al verificarsi di una condizione booleana **C**, il ciclo che ripete un comando finché una data condizione **C** è vera, e la composizione sequenziale di comandi.
- Le condizioni **C** sono espressioni dal valore booleano; odd è un predicato unario che verifica se l'espressione è 0, poi ci sono vari operatori di confronto tra espressioni **E**.
- Le espressioni **E** sono identificatori **I** (considerati parte del vocabolario), valori naturali **N** (considerati parte del vocabolario), operazioni tra espressioni, e espressioni tra parentesi; inoltre, con bop sintetizziamo la metavariabile dell'insieme {+,-,*,/}.

Per la grammatica delle espressioni dobbiamo fare attenzione all'*ambiguità*. Se non consideriamo le parentesi, la grammatica di **E** descritta sopra è ambigua. Se invece mettiamo tutte le parentesi, allora non abbiamo più l'ambiguità ma il linguaggio diventa pesante da usare e difficile da leggere. Nella *figura (b)* si può vedere una grammatica delle espressioni non ambigua, in quanto rispetta tutte le convenzioni (precedenze matematiche in assenza di parentesi).

Inoltre, nella stessa figura, si può vedere un esempio di completamento della grammatica in cui l'identificatore **I** è una stringa che deve necessariamente iniziare con un carattere non numerico. Tale completamento serve al *compilatore*, in quanto esso necessita che tutti gli elementi siano specificati.

```
<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 )
```

(a) Sintassi di PL0

```
<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
```

(b) Espressioni non ambigue e grammatica completa

Capitolo 3

Espressioni

Abbiamo detto che le espressioni sono oggetti sintattici usati per ragionare sui loro significati, che sono i *valori*. Per associare significati/valori alle espressioni, esse devono essere *valutate*.

3.1 Valori esprimibili

Il valore ottenuto mediante la valutazione di una espressione è detto valore esprimibile. In generale, abbiamo espressioni che rappresentano *interi* e *booleani*.

NOME	DESCRIZIONE	METAVARIABILE
Valori esprimibili	$EVal = bool \cup int$	$ev ::= k$

3.2 Espressioni aritmetiche

Gli aspetti che le caratterizzano sono:

- *Regole di precedenza e associatività* per gli operatori.
- *Ordine di valutazione* degli operandi.
- Presenza di *side-effects* (qualunque effetto che non consiste puramente nella rappresentazione di valori).
- *Overloading* degli operatori (simboli di operatore che hanno significato diverso a seconda del tipo degli operatori a cui sono applicati).
- *Espressioni con tipi misti* (espressioni che combinano tipi diversi di valori).
- *Arietà* dell'operatore (numero di operandi) e *notazione*.

3.3 Notazione

La notazione specifica in che modo gli operandi e gli operatori vengono rappresentati, per indicare su quali operandi un operatore opera.

Abbiamo tre notazioni possibili:

- Notazione *in-fissa*: $a + b$.
- Notazione *post-fissa*: $ab+$.
- Notazione *pre-fissa*: $+ab$.

La più usata nelle espressioni aritmetiche è quella in-fissa perché più facile ed intuitiva; però è anche quella in cui bisogna specificare altre regole (associatività, precedenza) o strumenti (parentesi) per eliminare l'*ambiguità*. Nelle altre notazioni, infatti, l'arietà dell'operatore è spesso sufficiente ad evitare l'ambiguità.

3.3.1 Notazione in-fissa

Per la notazione in-fissa è necessario stabilire le regole di precedenza e associatività. Inoltre è necessario l'utilizzo delle parentesi. Le *regole di precedenza* di un operatore definiscono l'ordine in cui vengono valutati gli operatori "adiacenti" a diversi livelli di precedenza. I *livelli di precedenza* più tipici sono:

1. Parentesi
2. Operatori unari
3. $**$
4. $*, /$
5. $+, -$

Le *regole di associatività* definiscono l'ordine con cui vengono valutati operatori allo stesso livello di precedenza. La regola di associatività più tipica è la valutazione *da sinistra verso destra*.

3.3.2 Notazione post-fissa

La notazione post-fissa è più semplice di quella in-fissa, in quanto non necessita né di regole di precedenza e associatività né di parentesi. Serve però conoscere l'arietà degli operatori. La *valutazione* evolve da sinistra a destra usando una pila LIFO.

3.3.3 Notazione pre-fissa

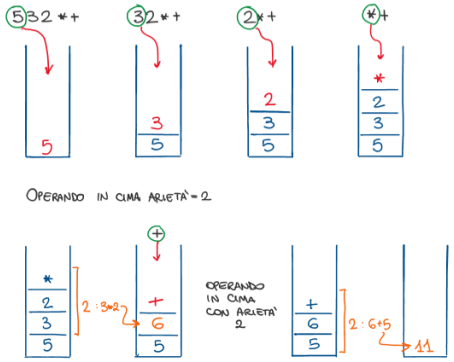
Anche la notazione pre-fissa è più semplice di quella in-fissa, in quanto non necessita né di regole di precedenza e associatività né di parentesi. Anche per essa serve però conoscere l'arietà degli operatori. La *valutazione* evolve sempre da sinistra verso destra usando una pila LIFO, ma in questo caso ogni volta che viene letto un operatore devo contare gli operandi da leggere prima di eseguire il calcolo.

- Leggi il prossimo simbolo dell'exp. e mettilo sulla pila
- Se il simbolo letto è un operatore:
  - applica a operandi immediatamente precedenti sulla pila,
  - memorizza il risultato in R,
  - elimina operatore ed operandi dalla pila
  - memorizza il valore di R sulla pila.
- Se la sequenza da leggere non è vuota torna a (1).
- Se il simbolo letto è un operando torna a (1).

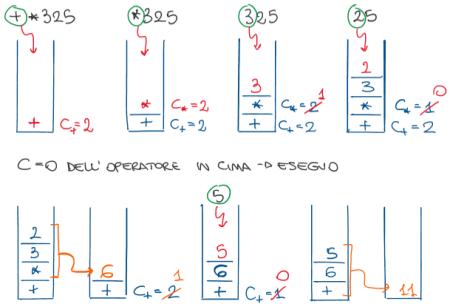
(a) Algoritmo di valutazione (post-fissa)

1. Leggi prox simbolo e metti sulla pila
2. Se è un operatore: C=n (arietà) e torna a 1
3. Se è un operando: C- -
4. Se C != 0 torna a 1
5. Se C=0
   A. Applica l'ultimo operatore ai successivi operandi e carica sulla pila il risultato eliminando gli oggetti usati
   B. Se non vi sono simboli vai a 6
   C. C=n-m (n= arietà primo operatore pila, m = operandi sopra operatore)
   D. Torna a 4
6. Se la pila è diversa da vuoto torna a 1

(b) Algoritmo di valutazione (pre-fissa)



(c) Esempio (post-fissa)



(d) Esempio (pre-fissa)

3.4 Valutazione delle espressioni

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

Tipicamente, le espressioni sono rappresentate da *alberi* che ne forniscono la struttura (con l'eliminazione di eventuali ambiguità legate a precedenze e associatività), ma che non possono dare informazioni riguardanti l'*ordine di valutazione*.

Siccome ci sono vari aspetti che possono influire sul risultato delle espressioni, è importante sapere se un linguaggio usa una valutazione:

- *Lazy*, cioè valuta gli operandi solo quando necessario.
- *Eager*, cioè se tutti gli operandi vengono comunque valutati.

Gli aspetti citati in precedenza sono:

- *Operandi non definiti* (divisione per 0).

```
a == 0 ? b : b/a
```

- *Effetti collaterali* (quando una funzione cambia i propri parametri).

```
fun(var) {
  var++;
}

a = 10;
b = a + fun(&a);
```

- *Aritmetica finita* (esiste un massimo numero rappresentabile).

3.5 Semantica delle espressioni

Dobbiamo dare significato alla seguente *grammatica* delle espressioni:

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

Per prima cosa dobbiamo definire alcuni *insiemi*:

- Espressioni $\mathcal{E}$, insieme di espressioni valutate ad intero o booleano (alberi di valutazione) con metavariabile e
- Numeri $\mathcal{N}$, insieme di numeri della macchina sottostante con metavariabili m, n, p
- Booleani $\mathcal{B}$, insieme di valori booleani (true, false) con metavariabile t

Ora usiamo questi insiemi per definire il *sistema di transizione*, le cui regole forniranno esattamente la semantica operazionale delle espressioni di cui abbiamo dato la grammatica:

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

3.5.1 Regole

$\mathscr{E}_1$: m op n → k se m op n = k, m,n ∈ $\mathcal{N}$ k ∈ $\mathcal{N}\cup\mathcal{B}$

$\mathscr{E}_2$: $\dfrac{e \to e'}{e\ \mathbf{op}\ e_0 \to e'\ \mathbf{op}\ e_0}$

$\mathscr{E}_3$: $\dfrac{e \to e'}{m\ \mathbf{op}\ e \to m\ \mathbf{op}\ e'}$

$\mathscr{E}_4$: $t_1\ \mathbf{bop}\ t_2 \to t$ se t_1 op t_2 = t, $t_1, t_2, t \in \mathcal{B}$

$\mathscr{E}_{3'}$: $\dfrac{e \to e'}{e\ \mathbf{bop}\ e_0 \to e'\ \mathbf{bop}\ e_0}$

$\mathscr{E}_5$: $\dfrac{e \to e'}{t\ \mathbf{op}\ e \to t\ \mathbf{op}\ e'}$

$\mathscr{E}_6$: $\mathbf{not}\ t_1 \to t$ se $\mathbf{not}\ t_1 = t$, $t_1 \in \mathcal{B}$

$\mathscr{E}_7$: $\dfrac{e \to e'}{\mathbf{not}\ e \to \mathbf{not}\ e'}$

3.5.2 Valutazione ed equivalenza

La funzione di *valutazione* di espressioni $Eval : \mathcal{E} \to \mathcal{N} \cup \mathcal{B}$ che descrive il comportamento dinamico delle espressioni restituendo il valore in cui esse sono valutate, è definita come:

$$Eval(e) = k \iff e \to^* k$$

La funzione di *equivalenza* di espressioni $\equiv \subseteq \mathcal{E} \times \mathcal{E}$ è definita come:

$$e_0 \equiv e_1 \iff Eval(e_0) = Eval(e_1)$$

Capitolo 4

Ambienti

Gli ambienti sono insiemi di legami (*bindings*) associati agli *identificatori*.

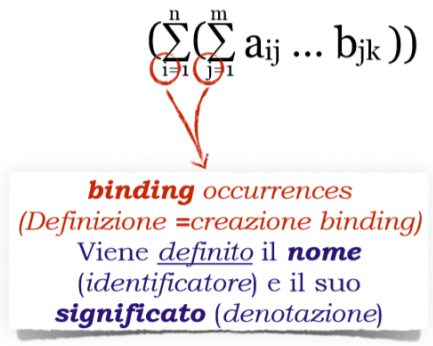
4.1 Identificatori

Gli identificatori sono sequenze di caratteri usate per rappresentare o denotare un altro oggetto. L'uso di nomi è fondamentale in quanto gli oggetti simbolici sono più facili da ricordare e permettono di attuare un processo di *astrazione*. Inoltre, un identificatore può denotare più elementi, e un elemento può essere denotato da più identificatori diversi (*aliasing*). Infine, gli identificatori non possono essere stringhe di caratteri qualunque, ma devono seguire delle regole in modo che il parser del linguaggio li possa riconoscere come tali.

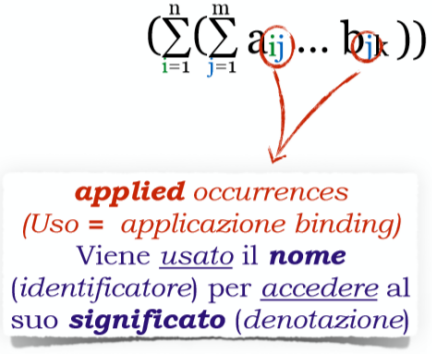
NOME	DESCRIZIONE	METAVARIABILE
Identificatori	`Id = {rate, a25, b, x, ...}`	id, x

4.2 Bindings

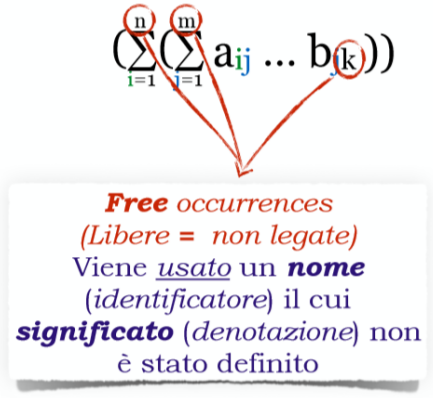
Il concetto di binding viene ereditato dalla matematica. Per prima cosa si "dichiara" il nome e il suo significato (*binding occurrences*); successivamente il nome viene usato per rappresentare il suo significato (*applied occurrences*); inoltre, un nome non dichiarato è un nome che può rappresentare qualunque cosa e quindi una sua occorrenza può essere legata a qualunque oggetto (*free occurrences*). Infine, il raggio di azione di una definizione (*scope*) definisce lo spazio in cui si può usare un nome per rappresentare il significato associato da un binding.



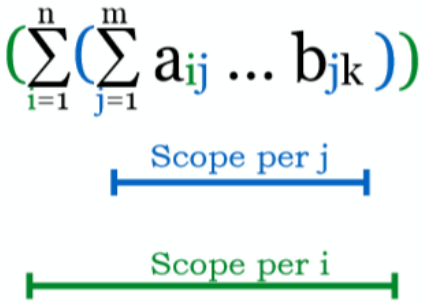
(a) Binding occurrences



(b) Applied occurrences



(c) Free occurrences



(d) Scope di un binding

4.2.1 Tipi di bindings

Nei *linguaggi di programmazione* i concetti di binding e scope prendono una connotazione propria:

- Un identificatore è in posizione di *definizione* (binding occurrence) quando si (ri)definisce il significato dell'identificatore.
- Un identificatore è in posizione di *uso* (applied occurrence) quando si denota il significato definito da una definizione.
- Un identificatore è in posizione *libera* (free occurrence) se il suo uso non è nel raggio di azione (scope) di una definizione.

I bindings possono legare nomi a diversi tipi di significato:

- *Nome-Valore*, quando il legame non può cambiare (costante).
- *Nome-Locazione*, quando abbiamo una variabile (per non modificare il binding alla memoria).
- *Locazione-Valore*, quando abbiamo una variabile (per modificare il valore del suo contenitore).

Esistono quindi i seguenti tipi di bindings:

- *Binding statico*, se occorre per la prima volta prima dell'esecuzione e rimane invariato durante tutta l'esecuzione del programma.
- *Binding dinamico*, se occorre durante l'esecuzione e può variare.

4.2.2 Tempi di bindings

È importante anche stabilire quando avviene la creazione dei bindings, ovvero a tempo di compilazione (*early binding*) o a tempo di esecuzione (*late binding*). Nel primo caso abbiamo: allocazione statica della memoria, esecuzione veloce, programmazione poco flessibile. Nel secondo caso, invece, abbiamo: allocazione dinamica della memoria, esecuzione lenta, programmazione flessibile. Altri esempi di tempi di bindings sono:

- Tempo di *progettazione* (legami ai simboli).
- Tempo di *implementazione* (legame del tipo floating point con la propria rappresentazione).
- Tempo di *compilazione* (legami di tipo).
- Tempo di *caricamento* (legami di variabili statiche alle celle di memoria).
- Tempo di *esecuzione* (legami di variabili non statiche alle celle di memoria).

4.3 Ambiente dinamico

Ora possiamo dire che un *ambiente* è l'insieme delle associazioni fra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma ed in uno specifico momento dell'esecuzione.

La *dichiarazione* quindi è un meccanismo (implicito o esplicito) col quale si crea un'associazione nell'ambiente; infatti, quando abbiamo delle occorrenze senza nessun legame in una espressione (free occurrences), vogliamo dar loro significato (per renderle applied occurrences), e con le dichiarazioni costruiamo le binding occurrences che rendono l'identificatore legato e non più libero.

L'*ambiente dinamico* associa identificatori a valori denotabili ($\perp$ va associato all'identificatore che non è associato ad alcun valore). Un ambiente dinamico (con metavariabile ρ) è un elemento dello spazio di funzioni

$$Env = \bigcup_{I \subseteq_f Id} Env_I$$

dove

$$Env_I : I \to DVal \cup \{\perp\}$$

4.4 Semantica per espressioni con identificatori

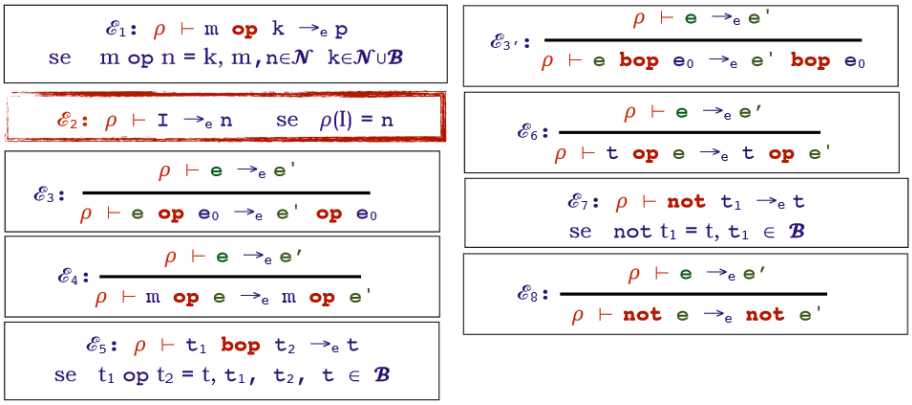
Dobbiamo dare significato alla seguente *grammatica* delle espressioni (arricchita con gli identificatori):

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

Usando il nuovo insieme $\mathcal{E}^V$ delle espressioni (con identificatori) valutate ad intero o booleano (alberi di valutazione) con metavariabile e, definiamo il *sistema di transizione* come:

$$\Gamma = \mathcal{E}^V,\ \mathrm{T} = \mathcal{N} \cup \mathcal{B},\ \mathrm{op} \in \{+,-,*,/\},\ \mathrm{bop} \in \{=,\mathrm{or}\}$$

4.4.1 Regole



4.4.2 Identificatori nelle espressioni

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

$$FI(k) = \varnothing$$

$$FI(id) = \{id\}$$

$$FI(e_0 \texttt{ bop } e_1) = FI(e_0) \cup FI(e_1)$$

$$FI(\texttt{uop } e) = FI(e)$$

4.4.3 Termini chiusi e ground

- In un linguaggio (con identificatori), un termine in cui non ci sono identificatori liberi è detto *termine chiuso*.
- In un linguaggio (con identificatori), un termine in cui non ci sono identificatori è detto *termine ground*.

4.5 Il tipo

Il tipo determina il *range di valori* che un identificatore può denotare e l'*insieme di operazioni* definite su quei valori. I tipi sono utili, per motivi diversi, a vari livelli:

- Livello di *progetto*, perché organizzano l'informazione.
- Livello di *programma*, perché identificano e prevengono errori.
- Livello di *implementazione*, perché permettono alcune ottimizzazioni.

4.5.1 Type binding

Il *tipo* può essere specificato in due modi:

- Dichiarazione *esplicita*, ovvero quando esiste un comando del linguaggio che permette di dichiarare il tipo delle variabili.
- Dichiarazione *implicita*, ovvero quando esiste un meccanismo di default che specifica il tipo delle variabili attraverso convenzioni.

Anche il *legame* che associa il tipo può avvenire in due modi:

- *Statico*, quando il tipo è associato a tempo di compilazione (rimane inalterato per l'intera esecuzione).
- *Dinamico*, quando il tipo è associato con l'assegnamento (può cambiare durante l'esecuzione).

4.6 Ambiente statico

L'*ambiente statico* associa agli identificatori il tipo degli oggetti che denoteranno ($\perp$ rappresenta il tipo di un identificatore non inizializzato). Un ambiente statico (con metavariabile Δ) è un elemento dello spazio di funzioni

$$TEnv = \bigcup_{I \subseteq_f Id} TEnv_I$$

dove

$$TEnv_I : I \to DTyp$$

4.6.1 Compatibilità di ambienti

Sia $\rho : I$ un ambiente dinamico e $\Delta : I$ un ambiente statico con $I \subseteq_f Id$. Gli ambienti ρ e Δ sono *compatibili* (scritto $\rho : \Delta$) se e soltanto se

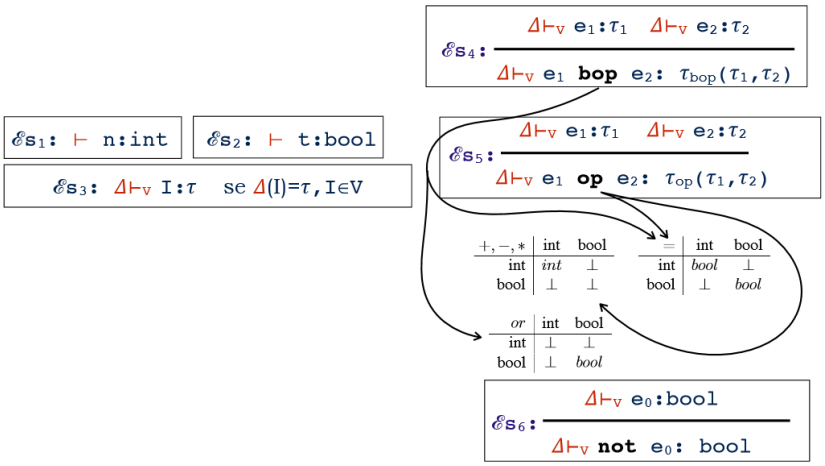
$$\forall id \in I.\, (\Delta(id) = \tau \wedge \rho(id) \in \tau)$$

Ovvero, un ambiente statico e uno dinamico sono compatibili se parlano in modo coerente degli stessi identificatori.

4.7 Semantica statica delle espressioni

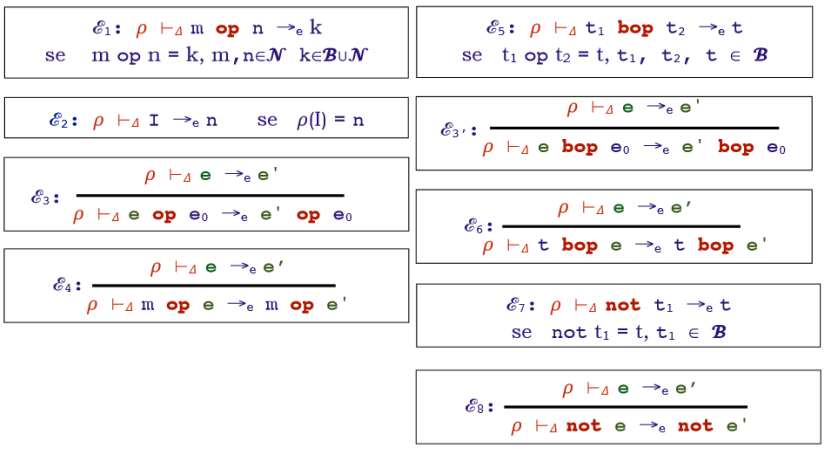
Dobbiamo definire uno strumento formale che permetta la creazione di legami di tipo e la verifica della corretta forma delle espressioni. Questo strumento consiste nella cosiddetta *semantica statica* che permette di associare un tipo ad ogni espressione corretta; in tal caso l'espressione è detta ben formata. A questo punto, la semantica per la valutazione delle espressioni diventa *semantica dinamica*.

4.7.1 Regole (semantica statica)



4.7.2 Regole (semantica dinamica)

L'ultima modifica di forma delle regole per la valutazione delle espressioni riguarda il fatto che ora non guardiamo più all'ambiente dinamico specificando l'insieme degli identificatori, ma specificando l'ambiente statico compatibile.



Capitolo 5

Dichiarazioni

Ricordiamo che le dichiarazioni denotano richieste di modifica/creazione di ambienti (legami tra identificatori e oggetti denotati); le dichiarazioni devono quindi essere elaborate per ottenere l'associazione che descrivono.

5.1 Grammatica delle dichiarazioni

Dobbiamo dare significato alla seguente *grammatica* delle dichiarazioni:

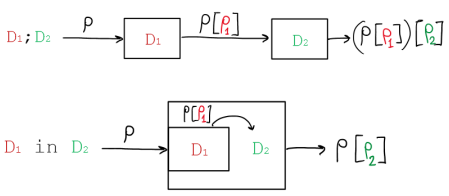
$$\mathcal{D}\text{: <Dec> D} \rightarrow \texttt{nil} \mid \texttt{const I:}\tau\texttt{ = e} \mid \texttt{D in D} \mid \texttt{D ; D} \mid \rho$$

Descriviamo ora le *regole* una per una:

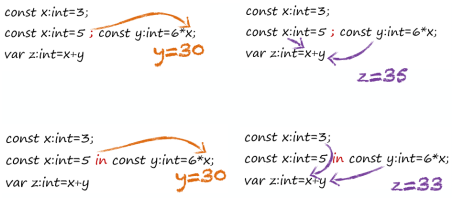
REGOLA	DESCRIZIONE	SEMANTICA
`nil`	Dichiarazione vuota	$\mathcal{D}_{S_1}$ e $\mathcal{D}_1$
`const I:`τ`=e`	Costante I di tipo τ e valore e	$\mathcal{D}_{S_3}$ e $\mathcal{D}_2$, $\mathcal{D}_3$
ρ	Ambiente	$\mathcal{D}_{S_2}$
D_1`;`D_2	Composizione sequenziale	$\mathcal{D}_{S_5}$ e $\mathcal{D}_4$, $\mathcal{D}_5$, $\mathcal{D}_6$
D_1 `in` D_2	Composizione privata	$\mathcal{D}_{S_4}$ e $\mathcal{D}_7$, $\mathcal{D}_8$, $\mathcal{D}_9$

5.2 Dichiarazioni composte

Nella *composizione sequenziale* di dichiarazioni D_1`;`D_2, tutto ciò che è definito da D_1 è visibile da D_2 e anche all'esterno. Nella *composizione privata* di dichiarazioni D_1 `in` D_2, invece, i legami creati da D_1 sono visibili e utilizzabili da D_2 ma non sono visibili all'esterno.



(a) Visibilità degli ambienti



(b) Esempio

5.3 Identificatori definiti

La funzione $DI : Dic \to \rho(Id)$ che associa ad ogni dichiarazione l'insieme degli identificatori che definisce, è definita per induzione da:

$$DI(\texttt{nil}) = \varnothing$$

$$DI(\texttt{const x:}\tau\texttt{=e}) = \{x\}$$
$$DI(d_1\,;d_2) = DI(d_1) \cup DI(d_2)$$
$$DI(d_1 \texttt{ in } d_2) = DI(d_2)$$
$$DI(\rho) = I \text{ dove } I \text{ dominio di } \rho$$

5.4 Identificatori liberi

La funzione $FI : Dic \to \rho(Id)$ che associa ad ogni dichiarazione l'insieme dei suoi identificatori liberi, è definita per induzione da:

$$FI(\texttt{nil}) = \varnothing$$

$$FI(\texttt{const x:}\tau\texttt{=e}) = FI(e)$$
$$FI(d_1\,;d_2) = FI(d_1) \cup (FI(d_2) \setminus DI(d_1))$$
$$FI(d_1 \texttt{ in } d_2) = FI(d_1) \cup (FI(d_2) \setminus DI(d_1))$$
$$FI(\rho) = \varnothing$$

5.5 Semantica delle dichiarazioni

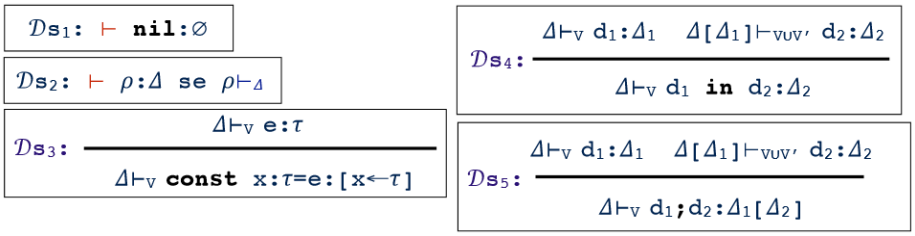
La *semantica statica* per le dichiarazioni deve associare un ambiente statico ad ogni dichiarazione scritta correttamente (ben formata). La *semantica dinamica*, invece, stabilisce come vengono elaborate le dichiarazioni.

5.5.1 Aggiornamento degli ambienti

Si considerino due ambienti $\beta, \beta' \in Env$ dove $\beta : V$, $\beta' : V'$ con $V, V' \subseteq Id$, ovvero β è definito sugli identificatori in V e β' è definito sugli identificatori in V'. L'*aggiornamento dell'ambiente* β mediante l'ambiente β' è l'ambiente $\beta'' \in Env$, denotato con $\beta[\beta']$ e definito come

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

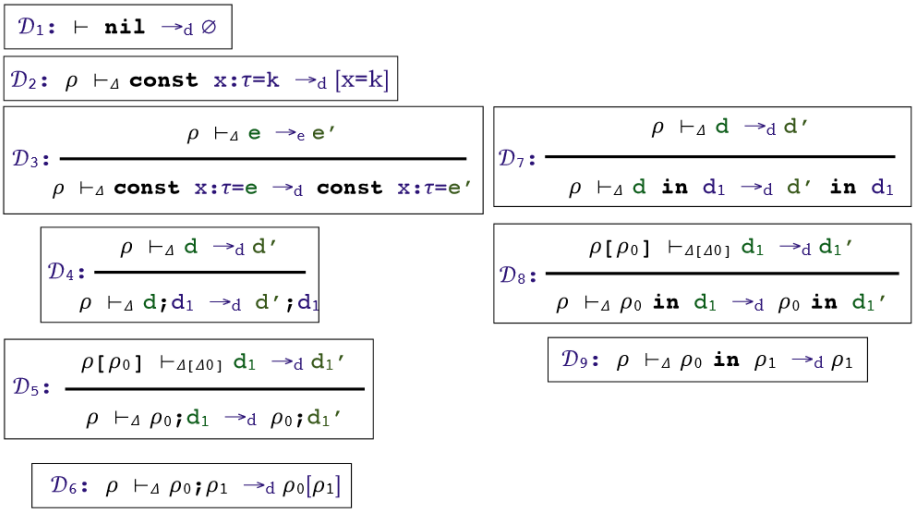
5.5.2 Regole (semantica statica)



5.5.3 Regole (semantica dinamica)

Usando l'insieme $\mathcal{D}$ delle dichiarazioni con identificatori elaborate in ambienti dinamici con metavariabile d, definiamo il *sistema di transizione* come:

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



5.5.4 Elaborazione ed equivalenza

La funzione di *elaborazione* $Elab : \mathcal{D} \rightarrow Env$ che descrive il comportamento dinamico delle dichiarazioni restituendo l'ambiente in cui esse sono elaborate, è definita come:

$$Elab(d) = \rho \iff d \rightarrow^* \rho$$

La funzione di *equivalenza* di dichiarazioni $\equiv\ \subseteq \mathcal{D} \times \mathcal{D}$ è definita come:

$$d_0 \equiv d_1 \iff Elab(d_0) = Elab(d_1)$$

Capitolo 6

Memoria

I comandi sono la categoria sintattica i cui elementi sono eseguiti per generare trasformazioni irreversibili di stato. In generale, l'astrazione dello stato della macchina è rappresentato dalla memoria. La *memoria* non è altro che un insieme di associazioni tra locazioni e valori.

A questo punto, per poter 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. Siccome non è pensabile che chi programma possa utilizzare direttamente le locazioni di memoria, queste ultime vengono riferite attraverso identificatori che chiameremo variabili. Le *variabili* quindi non sono altro che identificatori associati a locazioni, che a loro volta nella memoria sono associate a valori che possono cambiare durante l'esecuzione.

6.1 Locazioni

Per gestire tutta una serie di situazioni che coinvolgono identificatori variabili, abbiamo bisogno delle *locazioni*. Alcuni esempi di queste situazioni sono:

- Diversi oggetti chiamati con lo stesso nome (ridefinizione di variabili).
- Diversi nomi per lo stesso oggetto (aliasing).
- Nomi e valori di oggetti che cambiano nel tempo.

Abbiamo detto che lo stato della macchina viene rappresentato attraverso la sua memoria, ovvero attraverso l'insieme di associazioni tra locazioni (riferibili mediante variabili) e valori memorizzabili. La memoria, infatti, 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 alle variabili.

La locazione diventa quindi 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* e avrà un tempo di vita determinato dallo scope (visibilità) della dichiarazione stessa.

6.2 Variabili

Le variabili, come le costanti, sono caratterizzate dai seguenti aspetti:

- *Nome/identificatore*, ovvero la sequenza di caratteri utilizzata per riferirsi all'oggetto denotato.
- *Tipo*, ovvero l'insieme dei valori riferibili dall'oggetto e delle operazioni possibili su questi valori.
- *Valore*, ovvero l'oggetto memorizzato nella locazione associata all'identificatore.

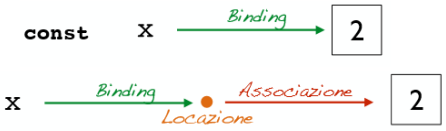
Gli aspetti propri delle sole variabili, invece, sono i seguenti:

- *Indirizzo/locazione*, ovvero l'astrazione della cella di memoria in cui il valore associato viene memorizzato.
- *Scope*, ovvero il range dei comandi a cui la variabile è visibile.
- *Tempo di vita*, ovvero tempo in cui il binding con la locazione è attivo.

Avendo introdotto un nuovo tipo di identificatore, ovvero la variabile, denotiamo con $\mathcal{D}^V$ il nuovo insieme delle *dichiarazioni con variabili* e aggiungiamo che:

$$DI(\texttt{var x:}\tau\texttt{=e}) = \{x\}$$

$$FI(\texttt{var x:}\tau\texttt{=e}) = FI(e)$$



(a) Costanti e variabili

$\mathcal{D}^V$: <Dec> D → nil | const I:τ = e | var I:τ = e | D in D | D ; D | ρ

(b) Nuova grammatica per le dichiarazioni

6.2.1 Regole (semantica statica)

Aggiungiamo un nuovo tipo per permetterci di distinguere identificatori variabili e costanti. Definiamo quindi il tipo τloc utilizzato per gli identificatori variabili di tipo τ.

$$\mathcal{E}s_3: \ \Delta \vdash_V I:\tau \quad \text{se } \Delta(I) \in \{\tau, \tau loc\}, I \in V$$

$$\mathcal{D}s_6: \ \frac{\Delta \vdash e:\tau}{\Delta \vdash \texttt{var } x:\tau\texttt{=}e:[x \leftarrow \tau loc]}$$

6.3 Aggiornare la memoria

Formalmente, una *memoria* è un elemento dello spazio di funzioni Mem definito da

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

dove $Mem_L : L \to MVal$ ha metavariabile σ e $MVal = \mathcal{N} \cup \mathcal{B}$ sono i valori memorizzabili.

Quindi, le *locazioni* sono modellate come un insieme finito ma illimitato di elementi che rappresentano celle di memoria. Per garantire questo esiste la funzione $New(L) \in (Loc \setminus L)$ che preso un insieme di locazioni già utilizzate, ne genera una nuova da utilizzare.

Si considerino ora due memorie $\sigma, \sigma' \in Mem$ dove $\sigma : L$, $\sigma' : L'$ con $L, L' \subseteq Loc$, ovvero σ è definito sulle locazioni in L e σ' è definito sulle locazioni in L'. L'*aggiornamento della memoria* σ mediante la memoria σ' è la memoria $\sigma'' \in Mem$, denotato con $\sigma[\sigma']$ e definito come

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

6.4 Semantica con memoria

Introdurre le variabili significa poter scrivere espressioni e dichiarazioni che contengono identificatori variabili. Questo implica che per valutare queste espressioni ed elaborare queste dichiarazioni abbiamo bisogno di integrare nel sistema di transizione anche la memoria da usare per valutare gli identificatori variabili. Quindi, il *sistema di transizione delle espressioni* diventa:

$$\Gamma^V = \mathcal{E}^V \times Mem,\ \mathrm{T}^V = (\mathcal{N} \cup \mathcal{B}) \times Mem$$

Mentre il *sistema di transizione delle dichiarazioni* diventa:

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

Osserviamo che nelle espressioni la memoria non può essere trasformata dalla valutazione, che accede in memoria sempre e solo in lettura. Nelle dichiarazioni, invece, la memoria può essere diversa dopo la transizione, questo perché inizializzando le variabili si modifica la memoria.

6.4.1 Valutazione ed equivalenza (con variabili)

La funzione di *valutazione* di espressioni $Eval : \mathcal{E}^V \times Mem \to (\mathcal{N} \cup \mathcal{B}) \times Mem$ che descrive il comportamento dinamico delle espressioni restituendo il valore in cui esse sono valutate (insieme alla *memoria*), è definita come:

$$Eval(\langle e, \sigma \rangle) = \langle k, \sigma \rangle \iff \langle e, \sigma \rangle \to^* \langle k, \sigma \rangle$$

La funzione di *equivalenza* di espressioni con variabili $\equiv \subseteq \mathcal{E}^V \times \mathcal{E}^V$ è definita come:

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

6.4.2 Elaborazione ed equivalenza (con variabili)

La funzione di *elaborazione* $Elab : \mathcal{D}^V \times Mem \to Env \times Mem$ che descrive il comportamento dinamico delle dichiarazioni restituendo l'ambiente in cui esse sono elaborate, è definita come:

$$Elab(\langle d, \sigma\rangle) = \rho \iff \langle d, \sigma\rangle \to^* \langle \rho, \sigma'\rangle$$

La funzione di *equivalenza* di dichiarazioni con variabili $\equiv \subseteq \mathcal{D}^V \times \mathcal{D}^V$ è definita come:

$$d_0 \equiv d_1 \iff \forall\sigma.(Elab(\langle d_0, \sigma\rangle) = Elab(\langle d_1, \sigma\rangle))$$

6.4.3 Regole (espressioni con variabili)

$\mathcal{E}_1$: $\rho \vdash_\Delta \langle \mathtt{m\ op\ n}, \sigma\rangle \to_e \langle \mathtt{k}, \sigma\rangle$ se $\mathtt{m\ op\ n} = \mathtt{k}, \mathtt{m}, \mathtt{n} \in \mathcal{N}\ \ \mathtt{k} \in \mathcal{N} \cup \mathcal{B}$

$\mathcal{E}_2$: $\rho \vdash_\Delta \langle \mathtt{I}, \sigma\rangle \to_e \langle \mathtt{n}, \sigma\rangle$ se $\rho(\mathtt{I}) = \mathtt{n}$ o $(\rho(\mathtt{I}) = \mathtt{l}\ \mathrm{e}\ \sigma(\mathtt{l}) = \mathtt{n})$

$$\mathcal{E}_3: \frac{\rho \vdash_\Delta \langle \mathtt{e}, \sigma\rangle \to_e \langle \mathtt{e'}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{e\ op\ e_0}, \sigma\rangle \to_e \langle \mathtt{e'\ op\ e_0}, \sigma\rangle}$$

$$\mathcal{E}_4: \frac{\rho \vdash_\Delta \langle \mathtt{e}, \sigma\rangle \to_e \langle \mathtt{e'}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{m\ op\ e}, \sigma\rangle \to_e \langle \mathtt{m\ op\ e'}, \sigma\rangle}$$

$\mathcal{E}_5$: $\rho \vdash_\Delta \langle \mathtt{t_1\ bop\ t_2}, \sigma\rangle \to_e \langle \mathtt{t}, \sigma\rangle$ se $\mathtt{t_1\ op\ t_2} = \mathtt{t}, \mathtt{t_1}, \mathtt{t_2}, \mathtt{t} \in \mathcal{B}$

$$\mathcal{E}_{3'}: \frac{\rho \vdash_\Delta \langle \mathtt{e}, \sigma\rangle \to_e \langle \mathtt{e'}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{e\ bop\ e_0}, \sigma\rangle \to_e \langle \mathtt{e'\ bop\ e_0}, \sigma\rangle}$$

$$\mathcal{E}_6: \frac{\rho \vdash_\Delta \langle \mathtt{e}, \sigma\rangle \to_e \langle \mathtt{e'}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{t\ bop\ e}, \sigma\rangle \to_e \langle \mathtt{t\ bop\ e'}, \sigma\rangle}$$

$\mathcal{E}_7$: $\rho \vdash_\Delta \langle \mathtt{not\ t_1}, \sigma\rangle \to_e \langle \mathtt{t}, \sigma\rangle$ se $\mathtt{not\ t_1} = \mathtt{t}, \mathtt{t_1} \in \mathcal{B}$

$$\mathcal{E}_8: \frac{\rho \vdash_\Delta \langle \mathtt{e}, \sigma\rangle \to_e \langle \mathtt{e'}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{not\ e}, \sigma\rangle \to_e \langle \mathtt{not\ e'}, \sigma\rangle}$$

6.4.4 Regole (dichiarazioni con variabili)

$\mathcal{D}_1$: $\vdash \langle \mathtt{nil}, \sigma\rangle \to_d \langle \varnothing, \sigma\rangle$

$\mathcal{D}_2$: $\rho \vdash_\Delta \langle \mathtt{const\ x{:}\tau{=}k}, \sigma\rangle \to_d \langle [\mathrm{x} \leftarrow \mathrm{k}], \sigma\rangle$

$$\mathcal{D}_3: \frac{\rho \vdash_\Delta \langle \mathtt{e}, \sigma\rangle \to_e \langle \mathtt{e'}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{const\ x{:}\tau{=}e}, \sigma\rangle \to_d \langle \mathtt{const\ x{:}\tau{=}e'}, \sigma\rangle}$$

$$\mathcal{D}_4: \frac{\rho \vdash_\Delta \langle \mathtt{d}, \sigma\rangle \to_d \langle \mathtt{d'}, \sigma'\rangle}{\rho \vdash_\Delta \langle \mathtt{d;d_1}\sigma\rangle \to_d \langle \mathtt{d';d_1}, \sigma'\rangle}$$

$$\mathcal{D}_5: \frac{\rho[\rho_0] \vdash_{\Delta[\Delta 0]} \langle \mathtt{d_1}, \sigma\rangle \to_d \langle \mathtt{d_1'}, \sigma'\rangle}{\rho \vdash_\Delta \langle \rho_0\mathtt{;d_1}, \sigma\rangle \to_d \langle \rho_0\mathtt{;d_1'}, \sigma'\rangle}$$

$\mathcal{D}_6$: $\rho \vdash_\Delta \langle \rho_0\mathtt{;}\rho_1, \sigma\rangle \to_d \langle \rho_0[\rho_1], \sigma\rangle$

$$\mathcal{D}_7: \frac{\rho \vdash_\Delta \langle \mathtt{d}, \sigma\rangle \to_d \langle \mathtt{d'}, \sigma'\rangle}{\rho \vdash_\Delta \langle \mathtt{d\ in\ d_1}, \sigma\rangle \to_d \langle \mathtt{d'\ in\ d_1}, \sigma'\rangle}$$

$$\mathcal{D}_8: \frac{\rho[\rho_0] \vdash_{\Delta[\Delta 0]} \langle \mathtt{d_1}, \sigma\rangle \to_d \langle \mathtt{d_1'}, \sigma'\rangle}{\rho \vdash_\Delta \langle \rho_0\ \mathtt{in\ d_1}, \sigma\rangle \to_d \langle \rho_0\ \mathtt{in\ d_1'}, \sigma'\rangle}$$

$\mathcal{D}_9$: $\rho \vdash_\Delta \langle \rho_0\ \mathtt{in}\ \rho_1, \sigma\rangle \to_d \langle \rho_1, \sigma\rangle$

$\mathcal{D}_{10}$: $\rho \vdash_\Delta \langle \mathtt{var\ x{:}\tau{=}k}, \sigma\rangle \to_d \langle [\mathrm{x} \leftarrow \mathrm{l}], \sigma[\mathrm{l} \leftarrow \mathrm{k}]\rangle$ $\mathrm{l} \in \mathrm{Loc}_\tau$ nuova locazione

$$\mathcal{D}_{11}: \frac{\rho \vdash_\Delta \langle \mathtt{e}, \sigma\rangle \to_e \langle \mathtt{e'}, \sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{var\ x{:}\tau{=}e}, \sigma\rangle \to_d \langle \mathtt{var\ x{:}\tau{=}e'}, \sigma\rangle}$$

Capitolo 7

Comandi

Ricordiamo che il significato di un comando è essenzialmente quello di una trasformazione di stato (ovvero di memorie); i comandi devono quindi essere eseguiti per ottenere la richiesta di trasformazione della memoria.

7.1 Grammatica dei comandi

Dobbiamo dare significato alla seguente *grammatica* dei comandi:

$\mathcal{C}^V$: <Com> C → skip | I := e | C ; C |
if b then C else C |
while b do C | D;C

Descriviamo ora le *regole* una per una:

REGOLA	DESCRIZIONE	SEMANTICA
`skip`	Comando nullo	$\mathcal{C}_{S_4}$ e $\mathcal{C}_8$
`I := e`	Assegnamento	$\mathcal{C}_{S_1}$ e $\mathcal{C}_1$, $\mathcal{C}_2$
`C ; C`	Composizione sequenziale	$\mathcal{C}_{S_3}$ e $\mathcal{C}_9$, $\mathcal{C}_{10}$
`if b then C else C`	Comando condizionale	$\mathcal{C}_{S_2}$ e $\mathcal{C}_3$, $\mathcal{C}_4$, $\mathcal{C}_5$
`while b do C`	Comando iterativo	$\mathcal{C}_{S_5}$ e $\mathcal{C}_6$, $\mathcal{C}_7$
`D ; C`	Blocco	$\mathcal{C}_{S_6}$ e $\mathcal{C}_{11}$, $\mathcal{C}_{12}$, $\mathcal{C}_{13}$

7.2 Assegnamento

L'*assegnamento* è il comando base che assegna il valore di una espressione ad un identificatore, chiamato anche variabile. Le *variabili*, infatti, sono contenitori di valori con un nome particolare, il cui contenuto può essere modificato tramite un assegnamento. L'assegnamento, quindi, è costruito utilizzando un termine che denota la locazione da modificare (left-value) e un termine che contiene il nuovo valore da associare alla locazione (right-value).

7.3 Comandi iterativi

I comandi possono essere *composti* sequenzialmente. In quel caso si avrà che tutte le trasformazioni eseguite dal primo comando sono il punto di partenza per le trasformazioni che deve eseguire il secondo. Inoltre, esiste anche il *comando condizionale*, che sceglie tra l'esecuzione di due possibili comandi in funzione del valore (booleano) di una espressione detta guardia. Tuttavia, se vogliamo ripetere l'esecuzione di un comando (o di un comando composto) finché il valore di una espressione booleana rimane vero, abbiamo bisogno dell'*iterazione* che, insieme alla ricorsione, permette di ottenere formalismi di calcolo Turing completi. L'iterazione può essere di due tipologie:

- *Indeterminata*, quando i cicli sono controllati logicamente su una espressione booleana (es. while).
- *Determinata*, quando i cicli sono controllati numericamente con un numero di ripetizioni determinate all'inizio del ciclo (es. for).

7.4 Blocchi

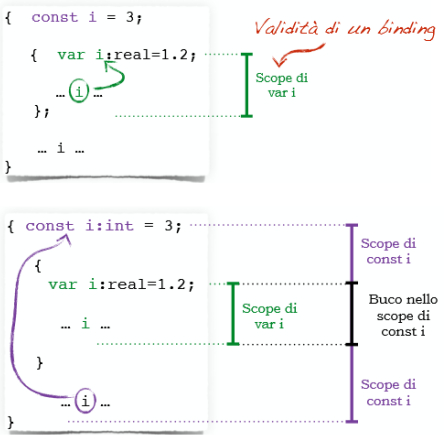
Il *blocco* serve per creare un ambiente locale al comando. Infatti, per eseguire un comando che coinvolge variabili abbiamo bisogno di combinare *ambienti* (creati e aggiornati dalle dichiarazioni con modifiche reversibili) e *memorie* (aggiornate mediante comandi con modifiche irreversibili). Il blocco, quindi, è il costrutto che ci permette di identificare l'ambiente in cui un comando viene eseguito e, in particolare, è una regione testuale che può contenere dichiarazioni locali e comandi. L'uso dei blocchi permette di:

- Gestire localmente i nomi.
- Aumentare la chiarezza.
- Usare i nomi indipendentemente gli uni dagli altri.
- Eseguire comandi in ambienti modificati.

7.4.1 Scope

Solitamente, i blocchi sono identificati da delimitatori che ne determinano lo scope. Inoltre, i blocchi non possono essere parzialmente sovrapposti, ovvero o sono disgiunti o sono annidati. È quindi importante stabilire la seguente *regola di visibilità* (scope) delle dichiarazioni: una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome (che nasconde la precedente).

(a) Blocchi annidati



(b) Validità dei bindings nei blocchi

Come si vede dalla *figura (b)*, ogni definizione annidata genera un buco nello scope precedente, ovvero nella porzione di codice in cui il binding è valido e il nome è utilizzabile con il significato associato.

Possiamo ora definire lo *scope statico*, che si riferisce all'area del testo del programma nella quale tutte le occorrenze applicate (di uso) di un identificatore si riferiscono alla stessa occorrenza di binding dell'identificatore.

7.4.2 Tipi di ambienti

Lo scope di una variabile è quindi il range di comandi ai quali è visibile. Esistono quindi variabili di tipo diverso rispetto alla regione in cui sono definite:

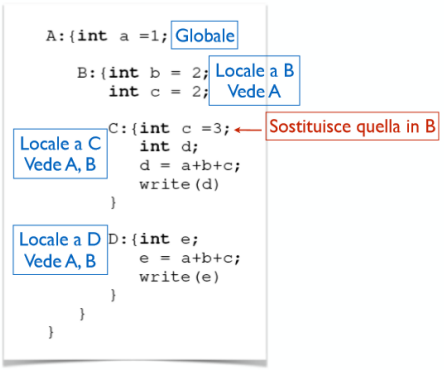
- Le *variabili locali* di un blocco sono quelle dichiarate nel blocco stesso.
- Le *variabili non locali* ad un blocco sono quelle visibili al blocco ma non dichiarate in esso.
- Le *variabili globali* sono quelle dichiarate nel blocco che contiene l'intero programma.

Tali tipologie di variabili determinano la caratterizzazione di diversi tipi di ambienti:

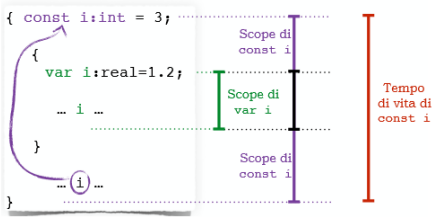
- *Ambiente locale*, che contiene le associazioni create all'ingresso del blocco (variabili locali).
- *Ambiente non locale*, che contiene le associazioni ereditate da altri blocchi ma visibili al blocco stesso.
- *Ambiente globale*, che contiene quella parte di ambiente non locale relativo alle associazioni comuni a tutti i blocchi.

7.4.3 Tempo di vita

Alla luce di quanto visto per ambienti e visibilità, dovrebbe essere chiaro che non sempre le variabili esistono per tutta la durata di un programma. Esse, infatti, hanno un *tempo di vita*, ovvero il tempo di esecuzione nel quale tutte le occorrenze applicate di un identificatore si riferiscono alla stessa locazione di memoria. Il tempo di vita di una variabile inizia quando viene legata ad una specifica cella (allocazione) e termina quando il legame viene sciolto (deallocazione). Osserviamo quindi che scope e tempo di vita sono concetti molto vicini, ma di natura diversa. Infatti, lo scope è un concetto spaziale (definito a tempo di compilazione) mentre il tempo di vita è un concetto temporale (definito a tempo di esecuzione).



(a) Tipi di ambienti



(b) Tempo di vita nei blocchi

7.5 Identificatori liberi

$$FI(\texttt{x := e}) = FI(e) \cup \{x\}$$

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

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

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

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

7.6 Identificatori definiti

$$DI(\texttt{x := e}) = \varnothing$$

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

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

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

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

7.7 Semantica dei comandi

La *semantica statica* per i comandi serve solo a verificare che il comando sia ben formato, che rispetti cioè tutti i vincoli semantici. La *semantica dinamica*, invece, stabilisce come vengono eseguiti i comandi. Definiamo quindi il seguente *sistema di transizione*:

$$\Gamma^V = \mathcal{C}^V \times Mem \cup Mem,\ \mathrm{T}^V = Mem$$

7.7.1 Regole (semantica statica)

$$\mathcal{CS}_1: \frac{\Delta \vdash_V \mathtt{e}:\tau}{\Delta \vdash_V \mathtt{x:=e}},\ \Delta(\mathtt{x}) = \tau\mathtt{loc} \qquad \mathcal{CS}_3: \frac{\Delta \vdash_V \mathtt{c}_0,\ \Delta \vdash_V \mathtt{c}_1}{\Delta \vdash_V \mathtt{c}_0;\mathtt{c}_1}$$

$$\mathcal{CS}_2: \frac{\Delta \vdash_V \mathtt{e:bool} \quad \Delta \vdash_V \mathtt{c}_0 \quad \Delta \vdash_V \mathtt{c}_1}{\Delta \vdash_V \mathtt{if\ e\ then\ c}_0\ \mathtt{else\ c}_1} \qquad \mathcal{CS}_5: \frac{\Delta \vdash_V \mathtt{e:bool},\ \Delta \vdash_V \mathtt{c}}{\Delta \vdash_V \mathtt{while\ e\ do\ c}}$$

$$\mathcal{CS}_6: \frac{\Delta \vdash_V \mathtt{d}:\Delta' \quad \Delta[\Delta'] \vdash_{V \cup V'} \mathtt{c}}{\Delta \vdash_V \mathtt{d;c}}\ \Delta'\ \mathtt{su}\ V' \qquad \mathcal{CS}_4: \Delta \vdash_V \mathtt{skip}$$

7.7.2 Regole (semantica dinamica)

$$\mathcal{C}_1: \frac{\rho \vdash_\Delta \langle \mathtt{e},\sigma\rangle \to_e \langle \mathtt{e}',\sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{x:=e},\sigma\rangle \to_c \langle \mathtt{x:=e}',\sigma\rangle}$$

$$\mathcal{C}_2: \rho \vdash_\Delta \langle \mathtt{x:=k},\sigma\rangle \to_c \sigma[\mathtt{l} \leftarrow \mathtt{k}],\ \rho(\mathtt{x})=\mathtt{l}$$

$$\mathcal{C}_3: \frac{\rho \vdash_\Delta \langle \mathtt{e},\sigma\rangle \to_e \langle \mathtt{e}',\sigma\rangle}{\rho \vdash_\Delta \langle\ \mathtt{if\ e\ then\ c}_0\ \mathtt{else\ c}_1,\sigma\rangle \to_c \langle\ \mathtt{if\ e}'\ \mathtt{then\ c}_0\ \mathtt{else\ c}_1,\sigma\rangle}$$

$$\mathcal{C}_4: \rho \vdash_\Delta \langle \mathtt{if\ true\ then\ c}_0\ \mathtt{else\ c}_1,\sigma\rangle \to_c \langle \mathtt{c}_0,\sigma\rangle$$

$$\mathcal{C}_5: \rho \vdash_\Delta \langle \mathtt{if\ false\ then\ c}_0\ \mathtt{else\ c}_1,\sigma\rangle \to_c \langle \mathtt{c}_1,\sigma\rangle$$

$$\mathcal{C}_6: \frac{\rho \vdash_\Delta \langle \mathtt{e},\sigma\rangle \to_e^* \langle \mathtt{true},\sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{while\ e\ do\ c},\sigma\rangle \to_c \langle \mathtt{c;while\ e\ do\ c},\sigma\rangle}$$

$$\mathcal{C}_7: \frac{\rho \vdash_\Delta \langle \mathtt{e},\sigma\rangle \to_e^* \langle \mathtt{false},\sigma\rangle}{\rho \vdash_\Delta \langle \mathtt{while\ e\ do\ c},\sigma\rangle \to_c \sigma}$$

$$\mathcal{C}_8: \rho \vdash_\Delta \langle \mathtt{skip},\sigma\rangle \to_c \sigma$$

$$\mathcal{C}_9: \frac{\rho \vdash_\Delta \langle \mathtt{c},\sigma\rangle \to_c \langle \mathtt{c}',\sigma'\rangle}{\rho \vdash_\Delta \langle \mathtt{c;c}_0,\sigma\rangle \to_c \langle \mathtt{c}';\mathtt{c}_0,\sigma'\rangle}$$

$$\mathcal{C}_{10}: \frac{\rho \vdash_\Delta \langle \mathtt{c},\sigma\rangle \to_c \sigma'}{\rho \vdash_\Delta \langle \mathtt{c;c}_0,\sigma\rangle \to_c \langle \mathtt{c}_0,\sigma'\rangle}$$

$$\mathcal{C}_{11}: \frac{\rho \vdash_\Delta \langle \mathtt{d},\sigma\rangle \to_d \langle \mathtt{d}',\sigma'\rangle}{\rho \vdash_\Delta \langle \mathtt{d;c},\sigma\rangle \to_c \langle \mathtt{d}';\mathtt{c},\sigma'\rangle}$$

$$\mathcal{C}_{12}: \frac{\rho[\rho'] \vdash_{\Delta[\Delta']} \langle \mathtt{c},\sigma\rangle \to_c \langle \mathtt{c}',\sigma'\rangle}{\rho \vdash_\Delta \langle \rho';\mathtt{c},\sigma\rangle \to_c \langle \rho';\mathtt{c}',\sigma'\rangle}\ \rho' \vdash_{\Delta'}$$

$$\mathcal{C}_{13}: \frac{\rho[\rho'] \vdash_{\Delta[\Delta']} \langle \mathtt{c},\sigma\rangle \to_c \sigma'}{\rho \vdash_\Delta \langle \rho';\mathtt{c},\sigma\rangle \to_c \sigma'}\ \rho' \vdash_{\Delta'}$$

7.7.3 Esecuzione ed equivalenza

La funzione di *esecuzione* $Exec : \mathcal{C}^V \times Mem \to Mem$ che descrive il comportamento dinamico dei comandi restituendo la memoria risultante dalle trasformazioni effettuate dai comandi eseguiti, è definita come:

$$Exec(\langle c, \sigma\rangle) = \sigma' \iff \langle c, \sigma\rangle \to^* \sigma'$$

La funzione di *equivalenza* di comandi $\equiv \subseteq \mathcal{C}^V \times \mathcal{C}^V$ è definita come:

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

Capitolo 8

Procedure

L'uso delle *procedure* permette la cosiddetta astrazione del controllo. L'astrazione chiede di identificare le proprietà importanti di ciò che si vuole descrivere al fine di concentrarsi sulle questioni rilevanti ed ignorare le altre.

8.1 Astrazione del controllo

Le procedure sono lo strumento sintattico per l'*astrazione del controllo*. Tale forma di astrazione è quella che ci interessa, in quanto il nostro obiettivo è quello di abbreviare la scrittura di programmi che contengono sequenze ripetute di comandi uguali che possono differire nei dati su cui operano. Ciò che viene fatto nel processo di astrazione del controllo consiste essenzialmente nel legare una sequenza di comandi (detta *corpo* della procedura) ad un identificatore (detto *nome* della procedura). Mantenendo solo nome e corpo potremmo però associare solo esecuzioni che fanno sempre la stessa cosa ogni volta che vengono richiamate. Per rendere l'esecuzione dipendente da altri elementi validi al momento della chiamata, abbiamo bisogno di un'altro aspetto da considerare, ovvero i *parametri*. Nome e parametri formano quindi l'*interfaccia*. Il corpo, invece, è ciò che viene valutato/elaborato/eseguito ogni qualvolta il nome è richiamato, con appropriati parametri, in qualche punto di programma.

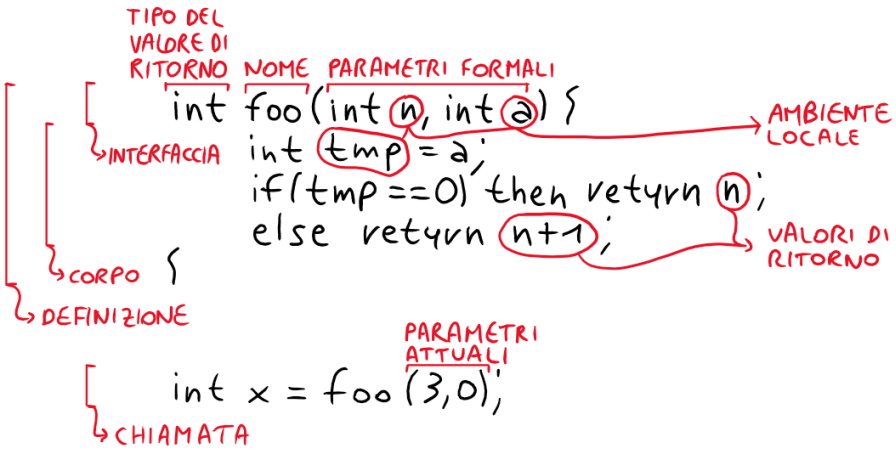
8.2 Sottoprogramma

Quando si parla di astrazione del controllo nei linguaggi di programmazione, ci sono in realtà due categorie di sottoprogrammi che possiamo definire:

- Le *procedure*, ovvero collezioni di comandi che definiscono una computazione parametrizzata senza necessariamente restituire un risultato, ma solo una modifica di stato.
- Le *funzioni*, che restituiscono un valore dopo aver eseguito modifiche di stato.

Un *sottoprogramma* è quindi una porzione di codice identificato da un nome, dotato di un ambiente locale proprio e capace di scambiare informazioni con il resto del codice mediante fissati canali (parametri e/o valori di ritorno).

Per caratterizzare un sottoprogramma dobbiamo specificarlo e scriverlo (tramite la *definizione*) e poi dobbiamo poterlo usare (tramite la *chiamata*).



8.3 Ambiente di riferimento

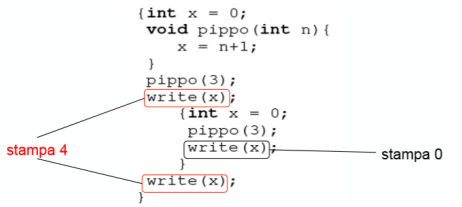
Il concetto di visibilità (scope) può essere esteso in presenza di procedure. Ciò ci permette di definire il concetto di *ambiente di riferimento* per un comando, ovvero la collezione di tutti i nomi che sono visibili al comando. L'ambiente di riferimento di una procedura è determinato da varie tipologie di regole:

- *Regole di scope* (statico o dinamico), che servono quando la procedura contiene un ambiente non locale.
- *Regole specifiche* del linguaggio.
- *Regole per il passaggio dei parametri.*
- *Regole di binding* (shallow o deep), che intervengono quando una procedura è passata come parametro ad un'altra procedura.

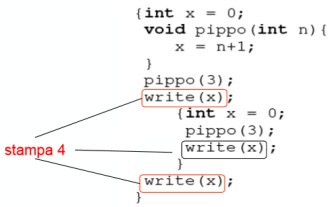
8.3.1 Tipi di scope

In un linguaggio con *scope statico*, l'ambiente di riferimento è costituito dalle variabili locali più tutte le variabili negli scope più esterni. In un linguaggio con *scope dinamico*, l'ambiente di riferimento è l'insieme delle variabili locali più tutte le variabili visibili nei sottoprogrammi attivi (cioè quelli la cui esecuzione è iniziata e non è ancora terminata).

Nello *scoping statico* (a tempo di compilazione), un nome non locale è risolto nel blocco che testualmente lo racchiude. Nello *scoping dinamico* (a tempo di esecuzione), invece, un nome non locale è risolto nella chiamata attivata più di recente e non ancora terminata.

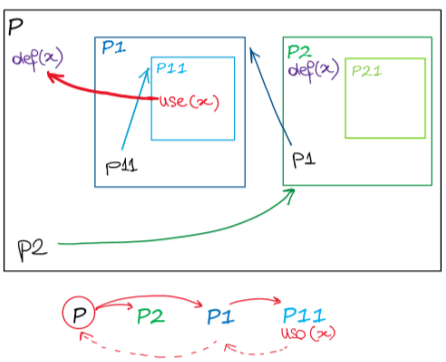


(a) Scoping statico

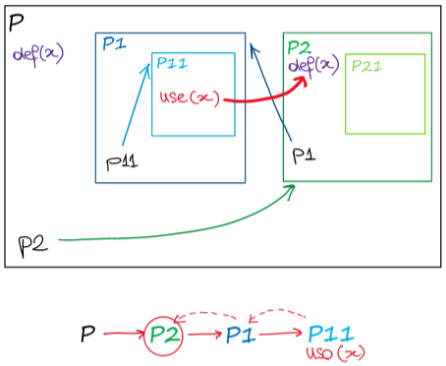


(b) Scoping dinamico

Con scoping statico, si risale la catena di annidamento dei blocchi seguendo quella che viene chiamata *catena statica* (nota a tempo di compilazione). Mentre con scoping dinamico, si risale la catena di chiamate dei blocchi seguendo quella che viene chiamata *catena dinamica* (nota a tempo di esecuzione).



(a) Catena statica



(b) Catena dinamica

In definitiva, lo scope statico è più complesso da implementare (lento nella costruzione della catena), ma è più efficiente (veloce nei riferimenti). Lo scope dinamico, invece, è più semplice da implementare (veloce nella costruzione della catena), ma è meno efficiente (lento nei riferimenti) perché non sfrutta le informazioni date dal compilatore.

8.4 Allocazione della memoria

Risolvere un riferimento vuol dire tradurre/sostituire il nome con l'oggetto a cui si riferisce attraverso la risoluzione del suo legame/binding. L'identificazione del legame corretto dipende dal tipo di *scope*: nel caso *statico*, dobbiamo mantenere la catena statica (tempo di compilazione) che ci permette di riferire gli oggetti mediante indirizzi relativi ad un base address (indirizzo dell'RdA allocato a tempo di esecuzione) e ad un offset (dentro l'RdA); nel caso *dinamico*, invece, basta seguire la catena dinamica (tempo di esecuzione) e cercare il riferimento in ogni RdA. Un concetto fortemente legato all'uso delle procedure è l'*allocazione della memoria*, ovvero come vengono allocati gli oggetti e dove vengono risolti i riferimenti.

Anche in questo caso esistono due meccanismi di allocazione della memoria:

- *Allocazione statica*, in cui la memoria è allocata a tempo di compilazione e prima dell'inizio dell'esecuzione tutti i dati del programma vengono disposti in opportune zone di memoria dove rimangono fino alla fine.
- *Allocazione dinamica*, in cui la memoria è allocata a tempo di esecuzione e si usano delle strutture dinamiche che durante l'esecuzione vengono continuamente allocate e deallocate (pila/stack con politica LIFO, heap).

8.4.1 Record di attivazione (RdA)

Nell'allocazione dinamica, ogni blocco ha un suo record di attivazione. La *pila* è la struttura dati naturale per gestire i RdA, in quanto permette proprio di implementare la struttura a blocchi dei programmi. La pila segue una *politica LIFO*, ovvero se si entra in A e poi in B si deve prima uscire da B e poi da A. L'uso della pila, con allocazione dinamica per la gestione della memoria in presenza di procedure e chiamate a procedure, è compiuto mediante una serie di operazioni:

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

Siccome l'indirizzo di un RdA non è noto a tempo di compilazione, abbiamo bisogno di un puntatore chiamato *Stack Pointer* (SP) che punta al RdA del blocco attivo, ovvero quello in cima allo stack. Le informazioni contenute in un RdA sono accessibili mediante l'uso di un offset rispetto allo SP. Tali informazioni sono:

- *Link di controllo*, che puntano agli RdA precedenti nella catena (statico e dinamico).
- *Indirizzo di ritorno*, ovvero l'indirizzo dell'istruzione da eseguire al termine della procedura.
- *Indirizzo del risultato*, che consiste nell'indirizzo dove depositare il risultato della procedura (se presente).
- *Parametri attuali* della funzione, ovvero i valori passati come input alla procedura.
- *Risultati intermedi*, che vengono calcolati durante l'esecuzione della procedura chiamata.

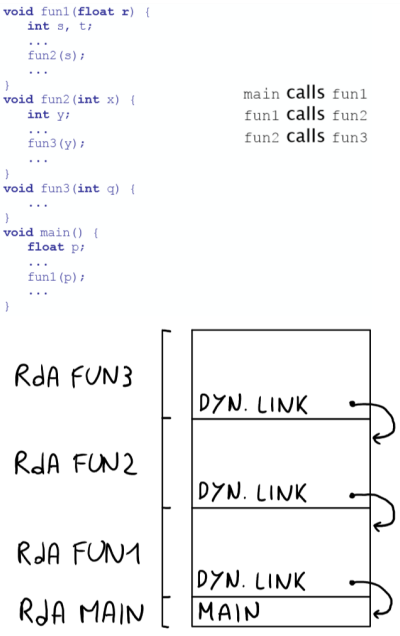
8.4.2 Semantica della chiamata e del ritorno

La semantica della *chiamata* ad un sottoprogramma si basa su vari passi che vanno implementati:

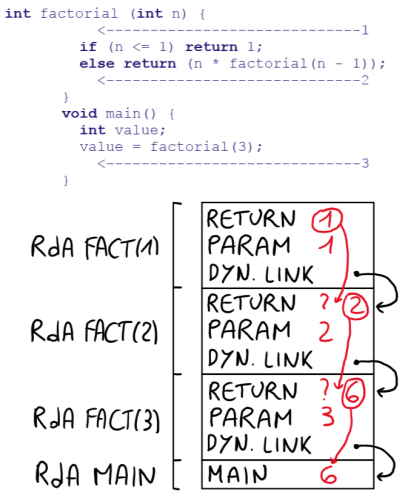
- Metodi di *passaggio dei parametri*.
- *Allocazione* dinamica sullo stack *delle variabili locali*.
- *Salvataggio dello stato di esecuzione* del programma chiamante.
- *Trasferimento del controllo* e preparazione del ritorno.

Analogamente, anche la semantica del *ritorno* da un sottoprogramma si basa sull'implementazione di varie fasi:

- Metodi di *passaggio dei parametri*.
- *Deallocazione dello stack* per le variabili non locali.
- *Recupero dello stato di esecuzione* del chiamante.
- *Ritorno del controllo* al chiamante.



(a) Esempio senza ricorsione



(b) Esempio con ricorsione

8.5 Implementazione dello scope statico

Lo *scope statico* soddisfa il principio di indipendenza, ovvero ogni chiamata ad un identificatore deve accedere sempre allo stesso ambiente per quell'identificatore (determinato staticamente). Per questo abbiamo bisogno di tenere traccia dei link statici e quindi della catena statica.

Gli elementi di cui abbiamo bisogno per l'*implementazione* sono:

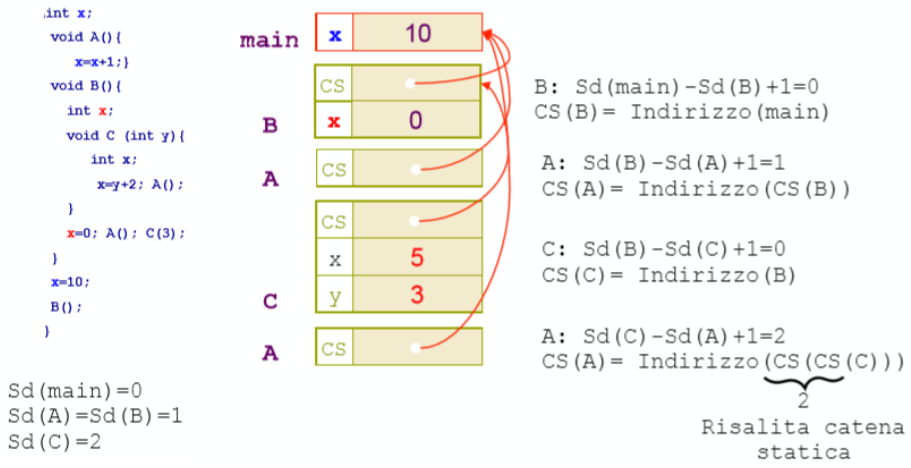
- La *catena statica* (CS), ovvero la catena di link statici che collegano istanze di RdA (ogni RdA è collegato a tutti gli antenati statici).
- Il *link statico*, che dipende dall'annidamento statico delle dichiarazioni delle procedure (il RdA per un sottoprogramma punta al RdA del genitore statico, ovvero dell'antenato del sottoprogramma che sintatticamente lo contiene).
- La *profondità statica* (Sd), ovvero un intero associato ad ogni RdA il cui valore è la profondità di annidamento della definizione della procedura corrispondente al RdA.

8.5.1 Determinare la catena statica

È il chiamante (Ch) a determinare il *link statico* del chiamato, in quanto bisogna risalire la catena statica del chiamante un numero di volte pari alla profondità di annidamento calcolata come differenza tra la profondità statica del chiamante e la profondità statica del sottoprogramma nel quale il chiamato viene definito. Quindi, quando una procedura chiamante *Ch* chiama un'altra procedura *P*, *Ch* può determinare se la definizione di *P* è immediatamente inclusa in *Ch* ($k = 0$), oppure se è inclusa in un blocco che si trova k passi prima di *Ch*. Il calcolo di k avviene nel seguente modo:

$$k = Sd(Ch) - Sd(P) + 1$$

A questo punto, se $k = 0$ allora *Ch* passa a *P* il proprio indirizzo come link statico, mentre se $k > 0$ allora *Ch* risale la propria catena statica di k passi e passa il corrispondente indirizzo come link statico di *P*. Risulta evidente che tutte queste computazioni sono basate su informazioni note a tempo di compilazione e quindi statiche.



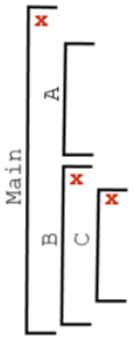
8.5.2 Risolvere i riferimenti

Una volta determinati correttamente i link statici, possiamo usarli per risolvere i riferimenti degli identificatori non locali usati nelle procedure. Infatti, non vogliamo cercare l'ambiente per un identificatore risalendo la catena statica e cercandolo in ogni RdA che incontriamo, bensì vogliamo usare le *informazioni statiche* a disposizione per calcolare il numero preciso di volte in cui bisogna risalire la catena statica per arrivare all'RdA che contiene l'ambiente corretto per l'identificatore. Staticamente conosciamo dove viene definita ogni variabile. Quindi, per ogni variabile, sappiamo staticamente quali sono i sottoprogrammi che la definiscono. A questo punto, quando usiamo una variabile in un sottoprogramma P, possiamo determinare quale è il sottoprogramma D più vicino (nella catena di annidamento) che definisce quella variabile e quindi possiamo calcolare il numero di volte N in cui risalire la catena statica come:

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

- Sequenza chiamate:
  - main B A C A
- main: x locale nel main (x=10)
- B: x locale in B (x=0)
- C: x locale in C (x=5)
- A: x non locale,
  - Antenato statico di A per x = main: Sd(A)-Sd(main)=1
  - Risaliamo la catena statica di 1 da A e troviamo l'RdA di main

```
int x;
void A(){
   x=x+1;}
void B(){
  int x;
  void C (int y){
     int x;
      x=y+2; A();
  }
  x=0; A(); C(3);
}
x=10;
B();
}
```



8.6 Implementazione dello scope dinamico

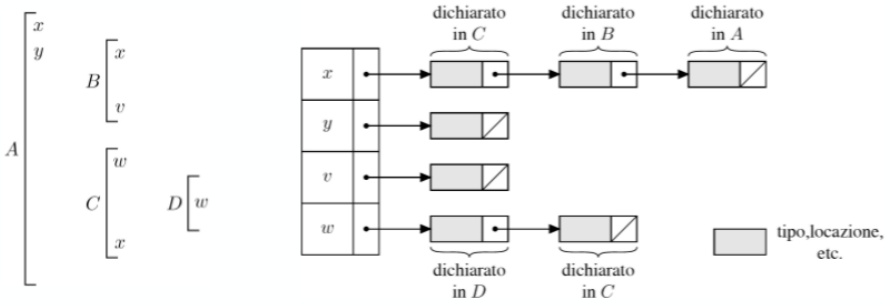
Nello *scope dinamico*, i riferimenti vanno risolti nell'ultimo blocco aperto (nella sequenza delle chiamate) e non ancora chiuso. Al contrario dello scope statico non possiamo usare informazioni statiche per sapere di quanto risalire la catena di chiamate, quindi l'implementazione può seguire due strategie:

- *Deep Access*, dove le variabili non locali vengono trovate cercando negli RdA lungo la catena dinamica.
- *Shallow Access*, dove si pongono le variabili locali in strutture dati centrali.

8.6.1 Tabella centrale dei riferimenti

Una delle possibili implementazioni che usano la strategia Shallow Access consiste nell'uso della *tabella centrale dei riferimenti* (CRT). La CRT è una tabella in cui ogni entry (una per ogni variabile presente nel programma) punta ad una lista di elementi che contengono le informazioni necessarie per accedere all'ambiente di riferimento per l'identificatore corrispondente alla entry. Il *funzionamento* della CRT prevede che, ogni volta che viene aggiunto sullo stack un RdA, per ogni identificatore definito nella procedura chiamata si aggiunga in cima alla sua lista nella CRT un elemento che rappresenta l'ambiente di riferimento della procedura chiamata. Questo significa che, per ogni identificatore del programma, l'ambiente di riferimento attivo è sempre quello in *cima* alla lista dell'identificatore nella CRT. Perché tutto funzioni, all'uscita da una procedura vanno eliminati dalla CRT gli elementi in testa che si riferiscono alla procedura che ha terminato.

Esempio: chiamate A,B,C,D



8.7 Semantica delle procedure

Le *procedure* sono un processo di astrazione del controllo che consistono nell'uso di un identificatore per far riferimento ad una porzione di codice da eseguire ogni volta che si usa il riferimento. Le procedure vanno dichiarate, ovvero va definita una associazione (analoga a quella degli identificatori costanti) tra l'identificatore e il codice che rappresenta. Dobbiamo perciò modificare la categoria sintattica delle *dichiarazioni* aggiungendo la dichiarazione di procedura (senza parametri) `proc P() C` dove `P` è il nome, mentre `C` è il corpo della procedura. Anche la categoria sintattica dei *comandi* va modificata aggiungendo il comando `P()` per chiamare la procedura di nome `P` durante l'esecuzione.

```
<Dec>  D → nil | const I:τ = e | var I:τ = e
               D in D | D ; D | ρ
               proc P() C
```

(a) Nuova grammatica delle dichiarazioni

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

(b) Nuova grammatica dei comandi

8.7.1 Regole (semantica statica)

Per elaborare una dichiarazione dobbiamo costruire l'ambiente statico corrispondente alla dichiarazione. Serve quindi introdurre il *nuovo tipo* `proc` che ci permetterà di identificare gli identificatori che si riferiscono a procedure.

$$DTyp = \{\texttt{int, intloc, bool, boolloc, proc}\}$$

$$\mathcal{D}s_7: \frac{\Delta \vdash_V C}{\Delta \vdash_V \texttt{proc P()C:[P=proc]}}$$

$$\mathcal{C}s_5: \Delta \vdash_V \texttt{P()} \quad \Delta(\texttt{P}) = \texttt{proc}$$

8.7.2 Regole (semantica dinamica)

Il tipo di scoping influisce semanticamente nella determinazione dell'ambiente. In caso di *scoping statico* dobbiamo in qualche modo congelare e portarci dietro l'ambiente valido al momento della definizione (per poterlo poi usare quando la procedura viene chiamata). In caso di *scoping dinamico*, invece, la procedura verrà eseguita nell'ambiente valido al momento della chiamata.

$$\mathcal{D}_{13}: \rho \vdash_\Delta \langle \textbf{proc P()C}, \sigma\rangle \to_d \langle [\texttt{P} \leftarrow \lambda\varepsilon.\texttt{C}'], \sigma\rangle$$

$$\mathcal{C}_{14}: \rho \vdash_\Delta \langle \texttt{P()}, \sigma\rangle \to_c \langle \texttt{C}', \sigma\rangle \qquad \rho(\texttt{P}) = \lambda\varepsilon.\texttt{C}'$$

In base a quanto detto sopra, C' è determinato (in base allo scoping) come:

$$C' = \begin{cases} \rho|_{FI(C)}\,; C & \text{per scoping statico} \\ C & \text{per scoping dinamico} \end{cases}$$

Osserviamo che nello scoping statico prendiamo l'ambiente valido al momento della definizione (ristretto agli identificatori liberi del corpo della procedura) e lo fissiamo come ambiente in cui eseguire il corpo componendolo sequenzialmente con il corpo stesso.

8.8 Procedure con parametri

Grazie ai parametri possiamo usare la stessa computazione in contesti differenti. Al momento della definizione della procedura si parlerà di parametri formali.

Un *parametro formale* è una variabile listata nell'interfaccia e usata nel corpo che definisce il sottoprogramma. Al momento della chiamata della procedura, invece, si parlerà di parametri attuali. Un *parametro attuale* rappresenta un valore o un indirizzo usato nel comando di chiamata per istanziare i parametri formali al valore da utilizzare quando il sottoprogramma viene eseguito. Il binding tra parametri attuali e formali può avvenire in due modi:

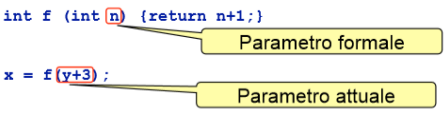
- *Posizionale*, dove il binding è dato dalla posizione (il primo parametro attuale è legato al primo formale e così via).
- *Per keyword*, dove il nome del parametro formale a cui un attuale si riferisce è specificato insieme all'attuale al momento della chiamata.

La direzione in cui si sposta l'informazione tra chiamante e chiamato è descritta dai seguenti *modelli di comunicazione*:

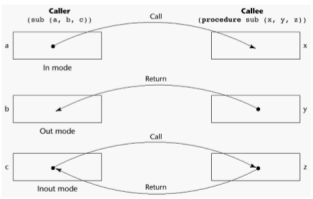
- *In-mode* ($main \Rightarrow proc$), dove l'informazione viaggia in un'unica direzione dal chiamante al chiamato mediante l'uso dei parametri.
- *Out-mode* ($main \Leftarrow proc$), dove l'informazione viaggia in un'unica direzione dal chiamato al chiamante mediante l'uso dei parametri.
- *Inout-mode* ($main \Leftrightarrow proc$), dove l'informazione viaggia in entrambe le direzioni sia mediante i parametri che mediante l'ambiente non locale e il valore di ritorno.

Il modo in cui il valore si sposta fisicamente tra chiamante e chiamato è descritto dai seguenti *modelli concettuali*:

- Spostare fisicamente il valore mediante una copia del valore nel parametro formale.
- Spostare un accesso al valore (riferimento).



(a) Parametri formali e attuali



(b) Passaggio dei parametri

8.8.1 Passaggio per valore

Il passaggio per valore implementa un passaggio *in-mode* che richiede la valutazione completa del parametro prima del passaggio. Questo implica che il parametro attuale può essere esclusivamente una espressione (r-value), mentre il parametro formale deve essere un riferimento (l-value). Il passaggio consiste quindi in una semplice *inizializzazione*, ovvero i parametri formali vengono inizializzati con il valore degli attuali. Inoltre, le modifiche dei formali non si riflettono negli attuali e questo significa che non c'è un ritorno di informazione al chiamante attraverso i parametri. L'implementazione avviene mediante la creazione di una *copia* del valore.

8.8.2 Passaggio per risultato

Quando un parametro è passato per risultato, nessun valore viene trasmesso al sottoprogramma. Questo significa che la comunicazione tra chiamante e chiamato è di tipo *out-mode*, ovvero di sola uscita dal chiamato. Il valore del parametro formale viene *copiato* nell'attuale. In tal caso quindi il parametro formale deve essere un valore (r-value), mentre il parametro attuale deve essere un riferimento (l-value).

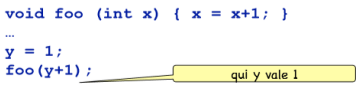
8.8.3 Passaggio per valore-risultato

Il passaggio per valore-risultato implementa un metodo *inout-mode*, ovvero permette una comunicazione bidirezionale mediante i parametri. In pratica, combina i due metodi visti per valore e per risultato, cioè il valore dell'attuale viene usato per *inizializzare* il formale e, al termine dell'esecuzione del sottoprogramma, il valore del formale viene *copiato* nell'attuale. Questo significa che sia attuali che formali devono essere sia l-value che r-value, ovvero variabili.

8.8.4 Passaggio per riferimento

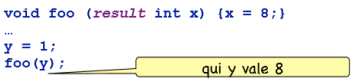
Nel passaggio per riferimento viene passato un *cammino di accesso* al parametro attuale. Viene anche chiamato passaggio per condivisione e implementa una metodologia *inout-mode*. Durante l'esecuzione tutti i riferimenti al valore del formale sono riferimenti al valore dell'attuale e tutti i cambiamenti del formale sono cambiamenti anche dell'attuale. I parametri attuali sono quindi dei riferimenti (l-value), mentre ai parametri formali viene assegnato il riferimento dell'attuale.

8.8.5 Esempi



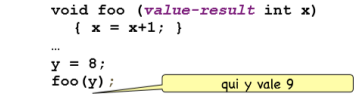
Il formale x è una var locale (sulla pila), alla chiamata, l'attuale y+1 è valutato ed il valore è assegnato al formale, non viene creato nessun legame tra x nel corpo di foo e y nel chiamante, al ritorno da foo, x viene distrutto (tolto dalla pila). Come detto, questo implica che non è possibile trasmettere info da foo al chiamante mediante il parametro.

(a) Passaggio per valore



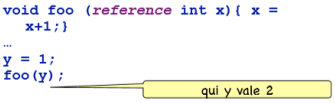
Qui il formale è una var locale (sulla pila), al ritorno da foo, il valore di x è assegnato all'attuale y, non c'è nessun legame tra x nel corpo di foo e y nel chiamante, al ritorno da foo, x viene distrutto (tolto dalla pila). Non è possibile trasmettere informazioni dal chiamante a foo mediante il parametro. L'implementazione si basa sulla copia e quindi è costoso per dati di grandi dimensioni.

(b) Passaggio per risultato



Nel seguente esempio, il formale x è a tutti gli effetti una variabile locale (sulla pila), alla chiamata, il valore dell'attuale è assegnato al formale , al ritorno, il valore del formale è assegnato all'attuale. Quindi non viene stabilito comunque nessun legame tra x nel corpo di foo e y nel chiamante. Al ritorno da foo, x viene distrutto (tolto dalla pila).

(c) Passaggio per valore-risultato



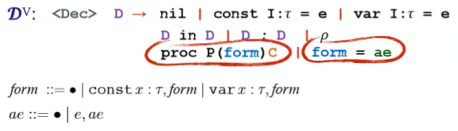
In tal caso viene passato un riferimento (indirizzo; puntatore) al parametro attuale, per questo l'attuale deve essere un l-valore (un riferimento), il formale x diventa quindi un alias di y. Al ritorno da foo, viene distrutto il (solo) legame tra x e l'indirizzo di y. Quindi la trasmissione è bidirezionale tra chiamante e chiamato.

(d) Passaggio per riferimento

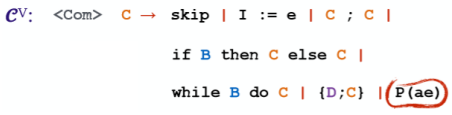
8.9 Semantica delle procedure con parametri

Abbiamo già visto che per creare l'associazione tra nome e procedura, le procedure vanno dichiarate. Ora modifichiamo leggermente la grammatica delle *dichiarazioni* e quella dei *comandi* per permetterci di dichiarare, e successivamente di chiamare, procedure che ricevono parametri.

Sostituiamo quindi la dichiarazione di procedura senza parametri con quella con parametri `proc P(form) C` dove `P` è il nome della procedura, `C` è il corpo della procedura e `form` è la lista dei parametri. Dobbiamo anche aggiungere la dichiarazione `form = ae` che permette di creare le associazioni tra i parametri formali e i parametri attuali descritti dalla lista `ae`. Inoltre, modifichiamo la grammatica dei comandi aggiungendo il comando `P(ae)` per chiamare la procedura di nome `P` con la lista di parametri attuali `ae`.



(a) Grammatica delle dichiarazioni definitiva



(b) Grammatica dei comandi definitiva

8.9.1 Identificatori e definizioni ausiliarie

Di seguito definiamo gli *identificatori liberi e definiti* delle nuove dichiarazioni e tutta una serie di *definizioni ausiliarie* che serviranno per definire le regole della semantica statica delle dichiarazioni che abbiamo aggiunto e modificato. Inoltre, definiamo gli identificatori liberi e definiti dei nuovi comandi.

$$
\begin{aligned}
&FI(\textbf{proc } P(form)C) = FI(C) \setminus DI(form)\\
&FI(form = ae) = FI(ae)\\
&FI(e, ae) = FI(e) \cup FI(ae)\\
&FI(\bullet) = \varnothing\\
&\\
&DI(\textbf{proc } P(form)C) = \{P\} \cup DI(form) \cup DI(C)\\
&DI(form = ae) = DI(form)\\
&DI(\textbf{const } x : \tau, form) = \{x\} \cup DI(form)\\
&DI(\textbf{var } x : \tau, form) = \{x\} \cup DI(form)\\
&DI(\bullet) = \varnothing\\
&\\
&DI(P(ae)) = \varnothing\\
&FI(P(ae)) = FI(ae) \cup \{P\}
\end{aligned}
$$

(a) Identificatori liberi e definiti

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

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

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

(b) Definizioni ausiliarie

8.9.2 Regole (semantica statica)

Per elaborare le dichiarazioni che abbiamo modificato dobbiamo aggiungere delle condizioni aggiuntive alle regole; questo per verificare la corrispondenza tra parametri formali e attuali e permettere quindi una corretta esecuzione della procedura. Inoltre, aggiorniamo anche la regola della chiamata.

$$\mathcal{D}s_{7'}: \frac{\texttt{form}:\Delta_0 \qquad \Delta[\Delta_0]\vdash_{VUV'} \texttt{C}}{\Delta\vdash_V \textbf{proc}\ \texttt{P(form)C}:[\texttt{P}=\mathcal{T}(\texttt{form})\texttt{proc}]}$$

$$\mathcal{D}s_8: \frac{\texttt{form}:\Delta_0 \qquad \Delta\vdash_V \texttt{ae}:\mathcal{T}(\texttt{form})}{\Delta\vdash_V \texttt{form} = \texttt{ae}: \Delta_0}$$

$$\mathcal{C}s_{5'}: \frac{\Delta\vdash_V \texttt{ae}:\texttt{aet}}{\Delta\vdash_V \texttt{P(ae)}} \qquad \Delta(\texttt{P})=\texttt{aetproc}$$

8.9.3 Regole (semantica dinamica)

Il sistema di regole della semantica dinamica delle dichiarazioni va aggiornato considerando anche i parametri. Quindi, nel valore associato al nome della procedura inseriamo anche i parametri formali. Rimane invece inalterata la costruzione di C', che dipende dal tipo di scoping. Infine, dobbiamo aggiungere la regola per l'associazione tra parametri attuali e formali, che però richiede tutta una serie di nuove regole per la valutazione della lista dei parametri attuali. Inoltre, aggiorniamo anche la regola della chiamata.

$$\mathcal{D}_{13'}: \rho\vdash_\Delta \langle \textbf{proc}\ \texttt{P(form)C},\sigma\rangle \to_d \langle[\texttt{P}\leftarrow\lambda\texttt{form.C'}],\sigma\rangle$$

$$\mathcal{D}_{14}: \frac{\rho\vdash_\Delta \langle\texttt{e},\sigma\rangle \to_e \langle\texttt{e'},\sigma\rangle}{\rho\vdash_\Delta \langle(\texttt{e},\texttt{ae}),\sigma\rangle \to_{ae} \langle(\texttt{e'},\texttt{ae}),\sigma\rangle}$$

$$\mathcal{D}_{15}: \frac{\rho\vdash_\Delta \langle\texttt{ae},\sigma\rangle \to_{ae} \langle\texttt{ae'},\sigma\rangle}{\rho\vdash_\Delta \langle(\texttt{k},\texttt{ae}),\sigma\rangle \to_{ae} \langle(\texttt{k},\texttt{ae'}),\sigma\rangle}$$

$$\mathcal{D}_{16}: \frac{\rho\vdash_\Delta \langle\texttt{ae},\sigma\rangle \to_{ae} \langle\texttt{ae'},\sigma\rangle}{\rho\vdash_\Delta \langle\texttt{form=ae},\sigma\rangle \to_d \langle\texttt{form=ae'},\sigma\rangle}$$

$$\mathcal{D}_{17}: \frac{\texttt{ak,L}\vdash \texttt{form}:\rho_0,\sigma_0}{\rho\vdash_\Delta \langle\texttt{form=ak},\sigma\rangle \to_d \langle\rho_0,\sigma[\sigma_0]\rangle}$$

$$\mathcal{C}_{14'}: \rho\vdash_\Delta \langle\texttt{P(ae)},\sigma\rangle \to_c \langle\texttt{form=ae;C'},\sigma\rangle \quad \rho(\texttt{P})=\lambda\texttt{form.C'}$$

$$\bullet, L\vdash \bullet:\emptyset,\emptyset \qquad ak ::= \bullet \mid k, ak$$

$$\frac{ak, L\vdash form:\rho_0,\sigma_0}{(k,ak), L\vdash \texttt{const}\ x:\tau, form:\rho_0[x=k],\sigma_0}$$

$$\frac{ak, L\cup\{l\}\vdash form:\rho_0,\sigma_0}{(k,ak), L\vdash \texttt{var}\ x:\tau, form:\rho_0[x=l],\sigma_0[l=k]}, l\in New_\tau(L_0), \sigma_0: L_0$$

8.10 Ricorsione

La *ricorsione* è un metodo alternativo all'iterazione per ottenere il potere espressivo delle MdT. In generale, una funzione si dice ricorsiva se viene definita in termini si sé stessa. Va osservato che ricorsione e induzione sono due concetti diversi, in quanto una definizione induttiva è sempre ben fondata (cioè arriva sempre ad un caso base), mentre possiamo definire funzioni ricorsive che divergono o che non definiscono nulla. Inoltre, ogni programma ricorsivo può essere tradotto in uno equivalente iterativo e viceversa.

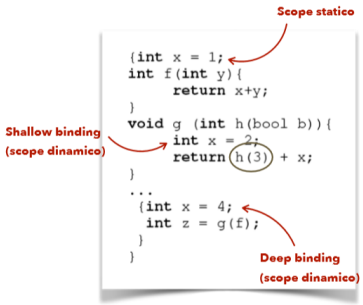
Una chiamata di g in f si dice *in coda* se f restituisce il valore restituito da g senza ulteriore computazione. Quindi, una funzione ricorsiva si dice *ricorsiva in coda* se contiene solo chiamate ricorsive in coda. Una funzione ricorsiva in coda permette una gestione statica della memoria, in quanto necessita di allocare un solo RdA perché tutte le chiamate possono essere soprascritte non avendo bisogno di ritornare valori che devono essere ulteriormente elaborati dal chiamante. In generale, ogni funzione ricorsiva può essere trasformata in una ricorsiva in coda perché basta aggiungere un parametro che raccoglie ad ogni passo il risultato parziale.

8.11 Funzioni di ordine superiore

Alcuni linguaggi permettono di passare funzioni come argomenti di procedure e di restituire funzioni come risultato di procedure. In entrambi i casi ci dobbiamo porre il problema di come gestire l'*ambiente non locale* della funzione. Quindi, per capire quale ambiente usare dobbiamo stabilire quale ambiente non locale si applica al momento dell'esecuzione.

In caso di *scope statico* non ci sono dubbi, in quanto l'ambiente da usare è sempre quello valido al momento della definizione. Quando invece abbiamo *scope dinamico*, allora di ambienti validi al momento dell'esecuzione ne abbiamo due, a seconda di:

- *Deep binding*, in cui l'ambiente valido è quello al momento della creazione del legame tra il parametro attuale e il parametro formale (ovvero quando la funzione viene passata come parametro).
- *Shallow binding*, in cui l'ambiente valido è quello al momento della chiamata della funzione (parametro attuale) attraverso il parametro formale.



Appendice A

Domande sulla teoria

A.1 Domande sul capitolo 1

Definire cosa è una macchina astratta e da quali componenti è costituita. Dare definizione del corrispondente linguaggio macchina.

Una *macchina astratta* M_L per il linguaggio L è un insieme di strutture dati ed algoritmi che permettono di memorizzare ed eseguire programmi scritti in L. La macchina astratta è la combinazione di una *memoria* che immagazzina i programmi e di un *interprete* che esegue le istruzioni dei programmi. Il linguaggio L_M è l'insieme di tutte le stringhe interpretabili dalla macchina astratta M e viene anche chiamato *linguaggio macchina*.

Cosa significa implementare un linguaggio? Definire esplicitamente e dettagliatamente tutti i concetti utilizzati.

Un linguaggio di programmazione è un insieme di costrutti e regole per descrivere algoritmi e dati. *Implementare* un linguaggio significa realizzare la macchina astratta che interpreta il linguaggio. Tale macchina astratta dovrà girare su una macchina ospite M_{L_O} avente un linguaggio L_O. Realizzare M_L consiste quindi nel realizzare una macchina che traduce L in L_O. I possibili approcci sono i seguenti:

- *Soluzione interpretativa*, cioè tramite traduzione implicita realizzata dalla simulazione dei costrutti di M_L mediante programmi scritti in L_O.

$$[int^{L_O,L}](P^L, in) = [P^L](in)$$

- *Soluzione compilativa*, cioè tramite traduzione esplicita dei programmi di L in corrispondenti programmi di L_O.

$$[comp^{L_O,L}](P^L) = P^{L_O} \text{ tale che } [P^{L_O}](in) = [P^L](in)$$

- *Soluzione ibrida*, cioè quando si passa dal linguaggio L al linguaggio intermedio L_{M_I} e i programmi scritti in L_{M_I} sono interpretati su M_O.

Definire cosa è un interprete. Dare definizione semantica e descriverne la struttura.

Un *interprete* è un programma $int^{L_O,L}$ che esegue, sulla macchina astratta per L_O, il programma P^L con input *in*. Formalmente, l'interprete $int^{L_O,L}$ da L a L_O è un programma tale che:

$$[int^{L_O,L}](P^L, in) = [P^L](in)$$

Un interprete è di fatto un *ciclo* di operazioni eseguite per simulare le istruzioni del linguaggio. Tali operazioni sono: elaborazione dei dati primitivi, controllo di sequenza delle esecuzioni, controllo dei dati, controllo della memoria.

Definire intuitivamente e formalmente il concetto di specializzazione e specializzatore.

La *specializzazione* è la valutazione parziale di un programma su un input fissato. Formalmente, uno *specializzatore* $spec^L$ per L è un programma tale che:

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

Specializzando un interprete rispetto al programma otteniamo un compilatore.

A.2 Domande sul capitolo 5

Descrivere intuitivamente cosa è una dichiarazione. Definire formalmente le dichiarazioni di IMP e darne semantica operazionale statica e dinamica.

Le *dichiarazioni* sono una categoria sintattica nella quale gli elementi vengono elaborati per creare o modificare ambienti (ovvero legami tra identificatori e oggetti denotati). Le modifiche sono reversibili in quanto le dichiarazioni non producono valori né modificano la memoria. Le *dichiarazioni di IMP* sono:

REGOLA	DESCRIZIONE	SEMANTICA
`nil`	Dichiarazione vuota	$\mathcal{D}_{S_1}$ e $\mathcal{D}_1$
`const I:`τ`=e`	Dichiarazione di costante	$\mathcal{D}_{S_3}$ e $\mathcal{D}_2$, $\mathcal{D}_3$
`var I:`τ`=e`	Dichiarazione di variabile	$\mathcal{D}_{S_6}$ e $\mathcal{D}_{10}$, $\mathcal{D}_{11}$
ρ	Ambiente	$\mathcal{D}_{S_2}$
D_1`;`D_2	Composizione sequenziale	$\mathcal{D}_{S_5}$ e $\mathcal{D}_4$, $\mathcal{D}_5$, $\mathcal{D}_6$
D_1 `in` D_2	Composizione privata	$\mathcal{D}_{S_4}$ e $\mathcal{D}_7$, $\mathcal{D}_8$, $\mathcal{D}_9$
`proc P(form) C`	Procedura con parametri	$\mathcal{D}_{S_7}$ e $\mathcal{D}_{13}$
`form = ae`	Associazione tra parametri	$\mathcal{D}_{S_8}$ e $\mathcal{D}_{14}$-$\mathcal{D}_{17}$

Nella *composizione sequenziale* di dichiarazioni, tutto ciò che è definito da D_1 è visibile da D_2 e anche all'esterno. Nella *composizione privata* di dichiarazioni, invece, i legami creati da D_1 sono visibili e utilizzabili da D_2 ma non sono visibili all'esterno.

A.3 Domande sul capitolo 7

Descrivere intuitivamente cosa è un comando. Definire formalmente i comandi di IMP e darne semantica operazionale statica e dinamica.

I *comandi* denotano richieste di modifica della memoria e devono essere eseguiti per ottenere tali richieste di trasformazione. Le modifiche della memoria sono irreversibili, in quanto non producono valori né modificano ambienti. I *comandi di IMP* sono:

REGOLA	DESCRIZIONE	SEMANTICA
`skip`	Comando nullo	$\mathcal{C}_{S_4}$ e $\mathcal{C}_8$
`I := e`	Assegnamento	$\mathcal{C}_{S_1}$ e $\mathcal{C}_1$, $\mathcal{C}_2$
`C ; C`	Composizione sequenziale	$\mathcal{C}_{S_3}$ e $\mathcal{C}_9$, $\mathcal{C}_{10}$
`if b then C else C`	Comando condizionale	$\mathcal{C}_{S_2}$ e $\mathcal{C}_3$, $\mathcal{C}_4$, $\mathcal{C}_5$
`while b do C`	Comando iterativo	$\mathcal{C}_{S_5}$ e $\mathcal{C}_6$, $\mathcal{C}_7$
`D ; C`	Blocco	$\mathcal{C}_{S_6}$ e $\mathcal{C}_{11}$, $\mathcal{C}_{12}$, $\mathcal{C}_{13}$
`P(ae)`	Chiamata di procedura	$\mathcal{C}_{S_{5'}}$ e $\mathcal{C}_{14}$

Il *comando condizionale* sceglie tra l'esecuzione di due possibili comandi in funzione del valore booleano di una espressione detta guardia. Il *comando iterativo* permette di ripetere l'esecuzione di un comando finché il valore della guardia rimane vero. L'iterazione (insieme alla ricorsione) permette di ottenere formalismi di calcolo Turing completi. Inoltre, l'iterazione può essere: indeterminata, quando i cicli sono controllati logicamente su una espressione booleana (ad es. while); determinata, quando i cicli sono controllati numericamente con un numero di ripetizioni determinate all'inizio del ciclo (ad es. for).

A.4 Domande sul capitolo 8

Si spieghi il concetto di scope (statico e dinamico).

Lo *scope* di una variabile è il range di comandi ai quali è visibile. Le regole di scope determinano come i riferimenti ad un nome sono associati alle variabili. Se lo scoping è *statico* (tempo di compilazione), un nome non locale è risolto nel blocco che testualmente lo racchiude. Se lo scoping è *dinamico* (tempo di esecuzione), un nome non locale è risolto nella chiamata attivata più di recente e non ancora terminata.

Descrivere cosa viene calcolato durante l'assegnamento in caso di scoping statico e in caso di scoping dinamico.

Nell'allocazione dinamica della memoria, ogni blocco ha un suo record di attivazione. Lo stack (pila LIFO) è la struttura dati naturale per gestire i RdA, in quanto permette proprio di implementare la struttura a blocchi dei programmi. Nello *scoping statico* ogni chiamata ad un identificatore deve accedere sempre allo stesso ambiente per quell'identificatore. Per questo abbiamo bisogno di tenere traccia dei link statici e quindi della catena statica, ovvero della catena di link statici che collegano istanze di RdA. In particolare, è il chiamante *Ch* a determinare il link statico del chiamato *P*, in quanto bisogna risalire la catena

statica del chiamante un numero di volte pari alla profondità di annidamento calcolata come differenza tra la profondità statica del chiamante e la profondità statica del sottoprogramma nel quale il chiamato viene definito:

$$k = Sd(Ch) - Sd(P) + 1$$

A questo punto, se $k = 0$ allora Ch passa a P il proprio indirizzo come link statico, mentre se $k > 0$ allora Ch risale la propria catena statica di k passi e passa il corrispondente indirizzo come link statico di P. Una volta determinati correttamente i link statici, possiamo usarli per risolvere i riferimenti degli identificatori non locali usati nelle procedure. Cioè vogliamo usare le informazioni statiche a disposizione per calcolare il numero preciso di volte in cui bisogna risalire la catena statica per arrivare al RdA che contiene l'ambiente corretto per l'identificatore. Infatti, quando usiamo una variabile in un sottoprogramma P, possiamo determinare quale è il sottoprogramma D più vicino (nella catena di annidamento) che definisce quella variabile e quindi possiamo calcolare il numero di volte N in cui risalire la catena statica come:

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

Nello *scoping dinamico*, invece, i riferimenti vanno risolti nell'ultimo blocco aperto e non ancora chiuso. Una delle possibili implementazioni consiste nell'uso della tabella centrale dei riferimenti (CRT), ovvero di una tabella in cui ogni entry punta ad una lista di elementi che contengono le informazioni necessarie per accedere all'ambiente di riferimento per l'identificatore corrispondente alla entry. Quindi, ogni volta che viene aggiunto sullo stack un RdA, per ogni identificatore definito nella procedura chiamata si aggiunge in cima alla sua lista nella CRT un elemento che rappresenta l'ambiente di riferimento della procedura chiamata. Questo significa che, per ogni identificatore del programma, l'ambiente di riferimento attivo è sempre quello in cima alla lista dell'identificatore nella CRT. Perché tutto funzioni, all'uscita da una procedura vanno eliminati dalla CRT gli elementi in testa che si riferiscono alla procedura che ha terminato.

Definire cosa è il passaggio di parametri, dando in particolare la definizione di tutti i modelli di comunicazione.

Grazie ai parametri possiamo usare la stessa computazione in contesti differenti. Al momento della definizione della procedura si parlerà di parametri formali, mentre al momento della chiamata della procedura si parlerà di parametri attuali. Il *passaggio di parametri* è quindi il binding tra parametri attuali e formali. Analizziamo ora nel dettaglio i quattro tipi di passaggio di parametri:

- Nel *passaggio per valore* (in-mode, $main \Rightarrow proc$) i parametri formali vengono inizializzati con il valore degli attuali, ma le modifiche del formale non si riflettono sull'attuale.
- Nel *passaggio per risultato* (out-mode $main \Leftarrow proc$) nessun valore è trasmesso al sottoprogramma, bensì il valore del formale viene copiato nell'attuale alla fine della procedura.
- Nel *passaggio per valore-risultato* (inout-mode $main \Leftrightarrow proc$) il parametro attuale viene usato per inizializzare il formale e, alla fine della procedura, il valore del formale viene copiato nell'attuale.

- Nel *passaggio per riferimento* (inout-mode *main* $\Leftrightarrow$ *proc*) viene passato un cammino di accesso al parametro attuale, e quindi durante l'esecuzione tutti i riferimenti al valore del formale sono riferimenti al valore dell'attuale e tutti i cambiamenti del formale sono cambiamenti anche dell'attuale.

Definire il concetto di ricorsione.

La *ricorsione* è un metodo alternativo all'iterazione per ottenere il potere espressivo delle MdT. Una funzione è ricorsiva se viene definita in termini si sé stessa. Ogni programma ricorsivo può essere tradotto in uno equivalente iterativo e viceversa. Inoltre, ogni funzione ricorsiva può essere trasformata in una funzione ricorsiva in coda aggiungendo un parametro che raccoglie ad ogni passo il risultato parziale.

Definire cosa sono le politiche di binding. Spiegare in particolare la differenza tra shallow binding e deep binding, facendo attenzione a come si combinano con lo scoping statico e/o dinamico.

Le *politiche di binding* intervengono quando una procedura è passata come parametro ad una altra procedura. In tal caso ci dobbiamo porre il problema di come gestire l'ambiente non locale della funzione, cioè dobbiamo stabilire quale ambiente non locale si applica al momento dell'esecuzione. In caso di *scoping statico* l'ambiente da usare è quello valido al momento della definizione. In caso di *scoping dinamico*, invece, dobbiamo distinguere due casi:

- *Deep binding*, in cui l'ambiente valido è quello al momento della creazione del legame tra il parametro attuale e il parametro formale (ovvero quando la funzione viene passata come parametro).
- *Shallow binding*, in cui l'ambiente valido è quello al momento della chiamata della funzione (parametro attuale) attraverso il parametro formale.

A.5 Altre domande

Descrivere almeno un aspetto caratterizzante dei paradigmi imperativo, logico e funzionale per i linguaggi ad alto livello.

Nel paradigma *imperativo* si descrive la variabile come astrazione della cella di memoria, la quale viene modificata tramite assegnamenti. Nel paradigma *funzionale*, invece, si descrivono i passi di calcolo come funzioni matematiche. Nel paradigma *logico*, infine, si usa la sostituzione come passo di calcolo primitivo.

Credits

Basato sulle dispense fornite dalla *prof.ssa Isabella Mastroeni*

Repository github: `https://github.com/zampierida98/UniVR-informatica`
Indirizzo e-mail: `zampieri.davide@outlook.com`