

Linguaggi di programmazione 1

Effeo

2021/2022

Indice

1	Informazioni del corso	6
1.1	Argomenti trattati	6
1.1.1	Obiettivi formativi	6
1.1.2	Contenuti	6
1.2	Informazioni	7
✓ 2	Introduzione	8
2.1	Parte precedente	8
2.2	Analisi del compilatore	9
2.2.1	Analisi lessicale	9
2.2.2	Analisi sintattica	10
2.2.3	Analisi semantica	10
2.2.4	Programma astratto	10
2.2.5	Generazione codice	10
2.2.6	Possibili errori	11
2.2.7	Visione front-end e back-end	11
2.3	Componenti di un Linguaggi di Programmazione	11
2.4	Proprietà dei linguaggi	12
2.5	Criteri di scelta di un linguaggio	12
2.6	Paradigmi	13
2.6.1	Differenza tra paradigma funzionale e imperativo	13
2.6.2	Esempi	13
2.6.3	Conclusioni	14
✓ 3	Modello imperativo	15
3.1	Memoria	15
3.2	Grammatica in forma Backus-Naur	15
3.2.1	Assegnazione	16
3.3	Ambiente	16
3.3.1	Ambiente nell'imperativo	16
3.3.2	Ambiente nell'funzionale	17
3.4	Data Object	17
3.4.1	Rappresentazione grafica di un data object	18
3.4.2	Modifiche di legami	18

3.5	Diagramma di Memory Layout	19
3.6	Legami di tipo	20
3.6.1	Type checking	20
3.6.2	Il linguaggio perfetto	21
3.6.3	Altre approssimazioni del controllo di tipi	22
3.7	Blocchi di istruzioni	22
3.7.1	Scope	22
3.7.2	Definizione di un nostro pseudo-linguaggio	23
3.7.3	Scoping statico e scoping dinamico	23
3.7.4	Mascheramento	24
3.8	Legami di locazione	24
3.8.1	Tecniche di allocazione	24
3.8.2	Record di attivazione di un blocco anonimo	25
3.9	Implementazione efficiente dell'ambiente non locale con scoping statico	26
3.9.1	Annidamento	26
3.9.2	Rappresentazione variabili non locali	27
3.9.3	Mantenimento del vettore degli ambienti non locali	28
3.9.4	Proprietà di questa implementazione	30
 4	Procedure	31
4.1	Astrazione procedurale	31
4.2	Dichiarazione di procedura	32
4.3	Invocazione di una procedura	33
4.4	Record di attivazione	33
4.5	Implementazione delle regole di scoping	34
4.5.1	Esempio scoping statico con procedure annidate	35
4.5.2	Esempio scoping dinamico	37
 5	Python	38
5.1	Sistema dei tipi	38
5.1.1	Mutabilità e immutabilità	39
 6	Parametrizzazione di procedure	40
6.1	Tipi dei parametri	40
6.2	Associazione dei parametri	41
6.2.1	Associazione di default	41
6.3	Parametri IN	41
6.4	Parametri OUT	42
6.5	Parametri IN-OUT	42
6.6	Aliasing	43
6.7	Procedure come parametri di procedure	44
6.8	Macro	44
6.9	Funzioni	46

✓ 7 Paradigma ad oggetti	47
7.1 Il modificatore static	47
7.1.1 Attributi	47
7.1.2 Blocchi di inizializzazione	48
7.1.3 Metodi statici	49
7.2 Scoping in Java	49
7.2.1 Memory Layout e mascheramento con ereditarietà	50
7.3 Regole di visibilità	51
7.4 Il sistema dei tipi	52
7.4.1 Tipi primitivi	52
7.4.2 Relazione di sottotipo	52
7.4.3 Assegnabilità	53
7.4.4 Operatore instanceof	53
7.4.5 Array e controllo dei tipi	53
7.4.6 Cast	54
7.4.7 Wrapper	55
7.4.8 Autoboxing	57
7.4.9 Auto-unboxing	58
7.5 Risoluzione overloading e overriding	59
7.5.1 Le fasi del binding	59
7.5.2 Binding dinamico e auto-(un)boxing	62
7.6 String e StringBuffer	64
7.6.1 String pool	65
7.6.2 StringBuffer e StringBuilder	66
7.7 Garbage collector	66
7.7.1 Algoritmo mark-and-sweep	66
7.8 Eccezioni	68
7.8.1 Catturare le eccezioni	68
7.8.2 Gerarchia delle eccezioni	69
7.8.3 Scelta del catch	70
7.8.4 Eccezioni verificate e non verificate	71
7.9 Regole dell'overriding	73
7.10 Classi interne	74
7.10.1 Classi interne statiche	76
7.11 Sguardo generale agli altri paradigmi	77
7.11.1 Sistemi di tipi	77
7.11.2 type equivalence	77
7.11.3 Compatibilità di tipi	77
7.11.4 Evoluzione dei sistemi di tipi	77
7.11.5 Caratteristiche utili alla classificazione dei linguaggi	78
7.12 Polimorfismo	79
7.12.1 Polimorfismo parametrico	80
7.12.2 Astrarre un'API tramite interfaccia	82

✓ 8 Paradigma funzionale	83
8.1 Il sistema di tipi	84
8.1.1 Tipi primitivi	84
8.2 Dichiarazioni e scoping	84
8.2.1 Funzioni	84
8.2.2 Dichiarazione di identificatori	85
8.2.3 Scoping	86
8.3 Costrutti	87
8.4 Tipi strutturati	87
8.4.1 Record	87
8.4.2 Dichiarazioni di tipo	88
8.4.3 Datatypes e costruttori	88
8.4.4 Costruttori con argomenti	89
8.5 Patterns e matching	90
8.5.1 Abbreviazioni per i pattern	91
8.5.2 Espressioni per casi	91
8.6 Liste	91
8.7 Currying	92
8.8 Funzioni di ordine superiore	93
8.8.1 Filter	93
8.8.2 Map	93
8.8.3 Reduce	94
8.9 Funzioni anonime	95
8.9.1 Val vs fun	95
8.9.2 Ulteriori dettagli su currying	96
8.10 Polimorfismo parametrico	97
8.10.1 Tipi parametrici	97
8.11 Encapsulazione e interfacce	97
8.11.1 Functors, strutture parametriche	98
8.12 Eccezioni	98
8.13 Esempio compilazione di espressioni	100
✓ 9 Paradigma logico	101
9.1 Costrutti base	102
9.1.1 Facts	102
9.1.2 Queries	103
9.1.3 Variabili logiche nelle query	103
9.1.4 Variabili logiche in generale	104
9.1.5 Termini	104
9.1.6 Termini ground e nonground	105
9.2 Come si risponde alle query	105
9.2.1 Sostituzione	105
9.2.2 Istanze	106
9.2.3 Unificazione	106
9.2.4 Costruzione delle risposte da soli fatti	107

9.2.5	Rappresentazione grafica del procedimento e conjunctive queries	107
9.3	Interpretazione logica	109
9.4	Regole	109
9.5	Ragionare con le regole	110
9.5.1	Overloading	111
9.5.2	Wildcards	112
9.5.3	Esempio analisi di circuiti	112
9.5.4	Regole ricorsive	112
9.5.5	Query non terminanti	112
9.5.6	Prolog e algebra relazionale	114
9.6	Liste	116
9.6.1	Evanescenza di parametri di input e output	116
9.7	Calcolo simbolico in Prolog	117
9.8	Cammini su grafi	118
9.9	Programmazione nondeterministica	118
9.9.1	Applicazione ai giochi	120
9.9.2	Analisi del gioco	123
9.9.3	Interfaccia utente testuale	124
9.10	Unicità di Prolog	124

Capitolo 1

Informazioni del corso

1.1 Argomenti trattati

1.1.1 Obiettivi formativi

Fornire gli elementi tecnici per classificare i numerosissimi linguaggi di programmazione esistenti, rispetto a paradigma di computazione, caratteristiche del sistema di tipi, modalità di gestione della memoria, controllo di flusso e supporto del parallelismo. Cominciare a rendere gli studenti "utenti intelligenti" dei linguaggi di programmazione, cioè capaci di scegliere il paradigma più adatto al contesto applicativo dato, di sfruttare efficacemente le funzionalità offerte dai linguaggi e di apprendere rapidamente nuovi linguaggi. Il corso fornisce un trattamento approfondito del core di Java ed elementi di linguaggi funzionali.

1.1.2 Contenuti

Introduzione ai linguaggi di programmazione. Cenni storici. Richiami degli elementi informatica teorica rilevanti per il corso. Cenni ai paradigmi di programmazione. Compilazione e interpretazione dei linguaggi. Supporto a run-time e gestione della memoria. Modalità di passaggio dei parametri. Strutturazione dei dati e controllo dei tipi. Tipi elementari e user defined. Encapsulation: tipi di dato astratti, moduli, classi. Sistemi di tipo nei linguaggi ad oggetti: sottotipi ed ereditarietà; compatibilità tra tipi. Java: Costrutti di controllo e sistema di tipi in dettaglio. Tipi parametrici (programmazione generica). Strutturazione della computazione: gestione delle eccezioni. Gestione della memoria in Java (inclusi costruttori, stringhe, garbage collection e gestione dell'ambiente non locale in presenza di classi interne). Parallelismo in Java. Costrutti funzionali di base, con esempi in ML e/o in Python

1.2 Informazioni

- **Crediti corso:** 6 cfu.
- **Insegnante:** Faella Marco.
- **Codice corso:** .
- **Libro di testo:** Linguaggi di programmazione, principi e paradigmi, Gabbrielli-Martini. Slide
- **Modalità esame:** Scritto e orale, il risultato dello scritto è non ammesso, esonerato dall'orale e ammesso all'orale.

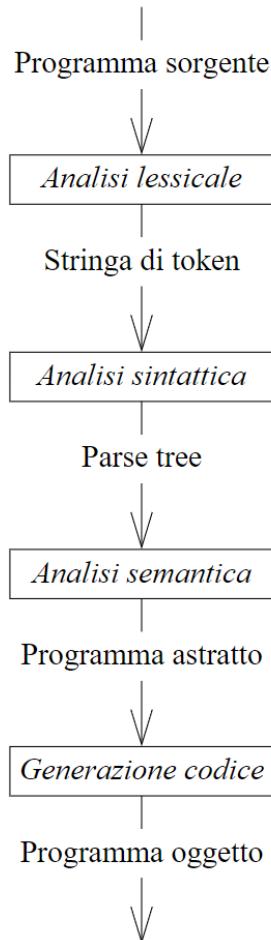
Capitolo 2

Introduzione

2.1 Parte precedente

Vedere dalle slide la parte precedente dal pacchetto di slide 1 dalla pagina 1 a 54.

2.2 Analisi del compilatore



2.2.1 Analisi lessicale

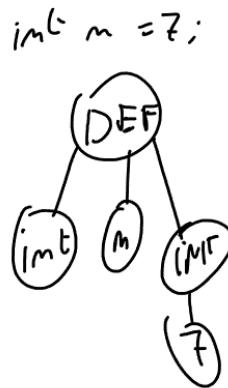
Divide la stringa in parti che si chiamano token, ad esempio: int n = 7; divide la stringa in 5 parti e ognuna di queste è un token. Controlla quindi se i token usati sono corretti.

2.2.2 Analisi sintattica

Distingue i token in tipologie sintattiche, dall'esempio di prima:

- **int**: tipo.
- **n**: identificativo.
- **=**: operatore.
- **7**: costante intera.
- **;**: terminatore.

Poi i token vengono inseriti in un parse tree:



2.2.3 Analisi semantica

Viene fatta una visita sul parse tree vengono fatti i controlli del tipo, se una variabile che viene utilizzata è stata dichiarata, che non sto facendo operazioni tra variabili diversi o altri tipi di controlli del compilatori.

2.2.4 Programma astratto

È come un parse tree che è stato arricchito con ulteriori informazioni di tipo semantico per esempio la symbol table che associa ad ogni variabile il suo tipo.

2.2.5 Generazione codice

Dal programma astratto viene generato il codice oggetto.

2.2.6 Possibili errori

- int n = 'ciao'; : errore di analisi semantica, sintatticamente è tutto corretto.
- Int c = 4; : errore sintattico, nella fase lessicale considera Int come un identificatore.
- int 1c = 3; : errore lessicale, non si può avere un identificatore che inizia con un numero, almeno in c. In ogni caso è raro l'errore lessicale.

2.2.7 Visione front-end e back-end

Si può notare che le fasi si possono distinguere in due tipi, le prime tre sono di tipo front-end, non dipendono dall'architettura della macchina, la quarta è di tipo back-end perché la creazione del programma oggetto dipende dall'architettura della macchina.

2.3 Componenti di un Linguaggi di Programmazione

- **Grammatica:** raggruppiamo la sintassi e il lessico.
- **Semantica:** il significato di ciascun costrutto.
- **Pragmatica:** le linee guida per un uso efficace del linguaggio.
- **Implementazione:** come si realizza una macchina astratta per un linguaggio o anche come si crea un compilatore o interprete di un linguaggio.

Nel corso parleremo molto di grammatica e di semantica con degli accenni sull'implementazione.

2.4 Proprietà dei linguaggi

- Semplicità – (concisione) VS (leggibilità)
 - ◆ Sintattica: unica rappresentabilità di ogni concetto.
 - ◆ Semantica: minimo numero di concetti e strutture.
- Astrazione – (incapsulare e dividere)
 - ◆ Dati: nascondere i dettagli (incapsulamento, information hiding)
 - ◆ Funzioni e procedure: dividere il programma
 - ◆ Moduli, pacchetti, etc.: favorire la modularità
- Espressività – (facilità di rappresentazione di oggetti) VS (semplicità)
- Ortogonalità – (poche eccezioni alle regole del linguaggio)
- Portabilità

2.5 Criteri di scelta di un linguaggio

- Disponibilità di interpreti/compilatori
- Esistenza di standard di portabilità
- Interoperabilità con altri sistemi e linguaggi
- Esistenza di librerie specifiche
- Maggiore conoscenza da parte del programmatore
- Comodità dell'ambiente di programmazione (IDE)
- Sintassi aderente al problema

2.6 Paradigmi

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).

Funzionale: Il programma e le sue componenti sono *funzioni*.
Esecuzione come valutazione di funzioni.

Logico: Programma come descrizione logica di un problema.
Esecuzione analoga a processi di dimostrazione di teoremi.

Orientato ad oggetti: Programma costituito da oggetti che scambiano messaggi.

Parallelo: Programmi che descrivono entità distribuite che sono eseguite contemporaneamente ed in modo asincrono.

Gli ultimi due sono ortogonali rispetto ai primi tre...

2.6.1 Differenza tra paradigma funzionale e imperativo

La differenza tra una funzione matematica e imperativa è che dati gli stessi input ritorna gli stessi output, mentre una funzione imperativa dati gli stessi input può ritornare output diversi, ad esempio la prima chiamata può modificare una variabile globale.

2.6.2 Esempi

Vedere dal gruppo di slide 1 da pagina 77 a 97.

2.6.3 Conclusioni

- Il paradigma di appartenenza può influenzare *radicalmente* il modo in cui si risolve il problema
 - ◆ In linguaggi dello stesso paradigma, lo stesso problema ha soluzioni strutturalmente identiche
 - ◆ Imparato a risolvere un problema in un linguaggio, lo si sa risolvere in tutti i linguaggi dello stesso paradigma
- Il paradigma non è l'unico aspetto determinante. Altri esempi di aspetti importanti:
 - ◆ Il sistema di tipi supportato
 - ◆ Eventuale supporto alle eccezioni
 - ◆ Modello di concorrenza e sincronizzazione
 - ◆ Presenza di garbage collection
 - ◆ ...

Capitolo 3

Modello imperativo

Il modello imperativo è il più vicino agli elaboratori standard, i programmi sono descrizioni di sequenze di modifica della memoria del calcolatore (infatti noi modifichiamo sempre delle variabili).

Ogni istruzione consiste in 4 passi:

1. Ottenere indirizzi delle locazioni di operandi e risultato.
2. Ottenere dati di operandi da locazioni di operandi.
3. Valutare il risultato.
4. Memorizzare il risultato in una locazione risultato.

Essenzialmente sono le stesse operazioni dell'assembly ma più ad alto livello. Le **variabili** sono dei nomi che rappresentano degli indirizzi, rispetto all'assembly invece dove lavoriamo direttamente sugli indirizzi.

3.1 Memoria

Consiste in un insieme di contenitori di dati, cioè le celle, ad ogni cella è associato un valore, cioè i valori delle variabili. Concettualmente quindi la memoria è una **funzione** da uno spazio di locazioni ad uno spazio di valori:

$$mem : Indirizzi \rightarrow Valori$$

mem(loc) = valore contenuto nella locazione loc

3.2 Grammatica in forma Backus-Naur

Sono delle semplici regole di sintassi per scrivere una grammatica CF. Prendiamo ad esempio la seguente regola: $exp \rightarrow exp + exp \dots$ nella grammatica Backus-Naur scriveremo: $<exp> ::= <exp> + <exp> \dots$

Vediamo gli elementi:

- **exp**: è un simbolo non terminale.
- **+**: è un simbolo terminale.
- $< exp > ::= < exp > + < exp > \dots$: è una regola.

3.2.1 Assegnazione

Definizione grammaticale:

$< assegnazione > ::= < nome > < operatore_di_assegnamento > < espressione >$

nome rappresenta la locazione dove viene posto il risultato, non è per forza una semplice variabile a ma può essere a.b o a e altro. In **espressione** invece viene specificata una computazione e i riferimenti ai valori necessari alla computazione, ad esempio può essere una variabile, espressioni aritmetiche e/o logiche, chiamate a funzioni ecc. L'**operatore_di_assegnamento** può cambiare in base al linguaggio.

A livello semantico quindi da name viene preso l'indirizzo su cui salvare il risultato, da espressione invece vengono presi i valori.

3.3 Ambiente

Introduciamo una nuova notazione: chiameremo **ambiente di esecuzione** una funzione che ai nomi visibili nel programma associa un indirizzo. Vediamolo però in modo più generale su tutti i paradigmi.

A tutti i paradigmi conviene introdurre una funzione **ambiente di esecuzione** che da l'interpretazione dei nomi visibili nel programma, ha come dominio un **insieme di nomi** (variabili, parametri, funzioni ecc) e il codominio dipende dal paradigma su cui lavoriamo.

$$env(id) : Nomi \rightarrow ???$$

3.3.1 Ambiente nell'imperativo

Nel paradigma imperativo la funzione env associa gli identificatori a locazioni di memoria, le quali a loro volta sono associate (funzione mem) al contenuto di memoria:

$$env : Nomi \rightarrow Indirizzi$$

Il valore di una variabile x è mem(env(x)).

Nel paradigma imperativo gli identificatori nascono e muoiono durante l'esecuzione e la funzione env identifica una associazione immutabile (la locazione di memoria associata a un nome non cambia).

Per esempio nell'assegnazione:

$$x := x + 1$$

Il valore dell'espressione a destra va salvato nella variabile a sinistra, quindi:

- La x di sinistra indica la locazione associata al nome ($\text{env}(x)$).
- La x di destra indica il valore della variabile ($\text{mem}(\text{env}(x))$).

3.3.2 Ambiente nell'funzionale

In questo caso il **VALORE** associato a un nome non cambia nel tempo, quindi la funzione env associa direttamente gli identificatori al contenuto della memoria:

$$\text{env} : \text{Nomi} \rightarrow \text{Valori}$$

Il valore di una variabile x è $\text{env}(x)$, quindi non esiste la funzione mem . Anche in questo caso env è un'associazione immutabile.

3.4 Data Object

Abbiamo definito delle notazioni per specificare la semantica di un certo costrutto, introduciamo un altro formalismo che ci consente di descrivere cosa ha fatto un programma sulla memoria.

Un **data object** è una quadrupla (L, N, V, T) composta da:

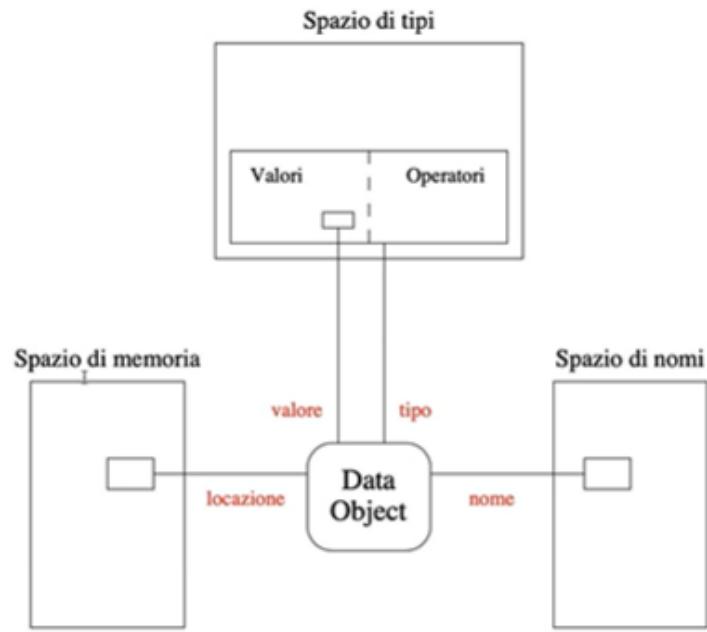
- L : locazione.
- N : nome.
- V : valore.
- T : tipo.

Chiamiamo **legame** la determinazione di una delle componenti. Quindi un data object è un insieme di 4 legami.

Nel paradigma imperativo:

- Il legame di locazione corrisponde alla funzione env . In modo più generico in un qualsiasi linguaggio il legame di locazione è uguale a $\text{env}(N)$.
- Il legame di valore di una variabile di nome x corrisponde a $\text{mem}(\text{env}(x))$, quindi $V = \text{mem}(\text{env}(N))$ in più si aggiunge il tipo.

3.4.1 Rappresentazione grafica di un data object



Dall'immagine possiamo vedere che un data object è un legame simultaneo tra 4 cose che appartengono a spazi diversi:

- Una **locazione** è un valore nello spazio di memoria cioè un indirizzo.
- Un **nome** è un valore nello spazio dei nomi.
- Un **valore** e un **tipo** invece vanno a braccetto perché un valore deve essere di uno specifico tipo quindi entrambi sono elementi di uno spazio dei tipi. Per completezza devono essere associati anche degli operatori, per ogni tipo abbiamo degli operatori specifici.

3.4.2 Modifiche di legami

Vediamo la dinamica dei legami, cioè quando nascono, quanto durano e quando finiscono. Individuiamo prima diverse fasi in cui abbiamo delle variazioni di legami:

1. Durante la **compilazione**.
2. Durante il **caricamento in memoria**.
3. Durante l'**esecuzione**.

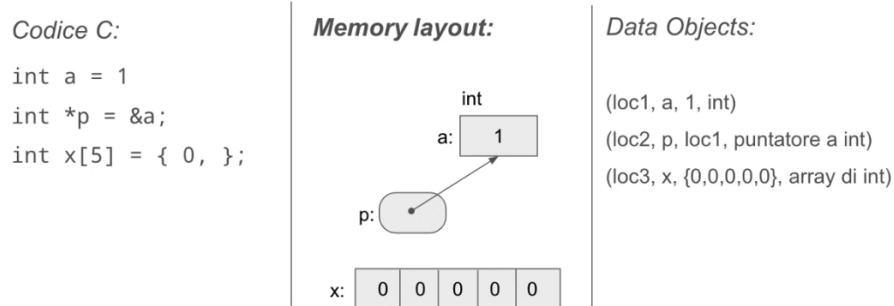
Ad esempio il **location binding** cioè il momento in cui si stabilisce l'indirizzo di ciascuna variabile viene effettuato al momento di **esecuzione** oppure al momento di **caricamento**. Dipende dalle variabili, ad esempio per le variabili locali il legame di locazione avviene al momento dell'esecuzione, cioè quando chiamo la funzione, per le variabili globali il legame viene stabilito al momento di caricamento in memoria (non al momento di compilazione perché posso compilare un programma e eseguirlo in un altro momento).

Il **name binding** cioè stabilire il nome di qualcosa avviene durante la compilazione, cioè i nomi sono già noti al compilatore rispetto agli indirizzi.

Il **type binding** avviene di solito durante la compilazione, ed è definito dal sottospazio di valori (e i relativi operatori) che un data object può assumere. Quindi gli elementi della quadrupla vengono riempiti in momenti diversi, ma alla fine prima o poi saranno stabiliti tutti i valori.

Vedere dalle slide degli esempi.

3.5 Diagramma di Memory Layout



I principi di questo diagramma sono:

- Mostra le allocazioni di memoria contigue, che non sono altro che i data object.
- Il valore di puntatori e riferimenti è rappresentato con frecce.
- Non distingue dove si trovano gli oggetti in memoria, quindi non distingue memoria heap e stack.
- È una sorta di visione grafica di data object.

Altri esempi nelle slide lezione02-addendum da pagina 4 a pagina 11

3.6 Legami di tipo

Il legame di tipo è correlato al legame di valore, cioè il tipo di una variabile e il suo valore devono corrispondere. Un tipo stabilisce due cose:

- Un insieme di valori che una variabile può assumere.
- Un insieme di operazioni che possono essere eseguite sulla variabile.

I tipi sono un costrutto linguistico, cioè inventato nei linguaggi di programmazione ad alto livello che evita errori di programmazione limitando quello che posso fare con determinate locazioni di memoria. Ogni volta che il legame di valore viene modificato occorre controllare la consistenza con il legame di tipo. Definiamo un linguaggio **dinamicamente tipizzato** se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza avvengono durante l'esecuzione, per esempio python.

Definiamo un linguaggio **staticamente tipizzato** se il legame avviene durante la compilazione, in questo caso il controllo di consistenza può avvenire in entrambe le fasi

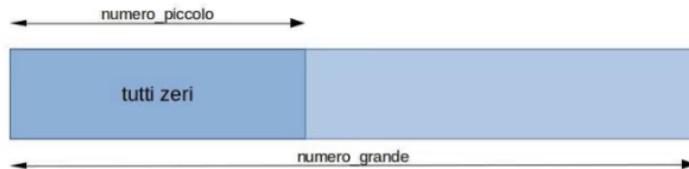
3.6.1 Type checking

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo. Può avvenire durante la compilazione, durante l'esecuzione o anche non avvenire. Un linguaggio è **fortemente tipizzato** se il controllo di consistenza avviene sempre, Java viene considerato fortemente tipizzato mentre pascal è quasi fortemente tipizzato, C è debolmente tipizzato perché esistono i casting, esistono i casting anche i Java ma in C sono più permissivi, inoltre esistono puntatori a void che possono poi essere convertiti in diversi tipi. In C poi esistono le **union** che sovrappongono la locazione di variabili di tipo diverso, senza controllare se il valore corrisponde al tipo con cui viene usato.

Vediamo un esempio di union:

```
union numero
{
    int numero_piccolo;
    double numero_grande;
} numeri;

void main() {
    numeri.numero_grande = 3.0;
    printf("%d\n", numeri.numero_piccolo);
}
```



- si inserisce un double ma si legge come fosse un int
- viene stampato '0'
- il compilatore non segnala l'errore

3.6.2 Il linguaggio perfetto

Sarebbe bello se esistesse un linguaggio Turing completo in cui il type checking avvenisse completamente durante la compilazione e in cui il compilatore non generasse più errori del necessario. Questo linguaggio sarebbe staticamente e fortemente tipizzato. Questo linguaggio non può esistere perché:

```
int x;
P;
x = "pippo";
```

Consideriamo P come un generico programma, il linguaggio perfetto dovrebbe

darci errore se l'ultima riga verrà mai eseguita, ma per sapere se l'ultima riga verrà eseguita dobbiamo sapere se P termina (IL MONDO SAREBBE UN POSTO MIGLIORE SE HALT FOSSE CALCOLABILE) e per questo motivo il linguaggio perfetto non può esistere e un linguaggio normale darebbe errore a quella riga.

3.6.3 Altre approssimazioni del controllo di tipi

Alcuni controlli di tipo sono impossibili a tempo di compilazione, ad esempio la correttezza dei down cast in Java cioè quando facciamo un cast da una classe a un'altra. Quindi la strategia per i linguaggi fortemente e staticamente tipati è fare quanti più controlli possibili a tempo di compilazione e eseguire i rimanenti a tempo di esecuzione.

3.7 Blocchi di istruzioni

I blocchi di istruzioni possono essere raccolti da:

- Una procedura o una funzione.
- Da file diversi.
- Strutture di controllo.

In generale la funzionalità principale dei blocchi è quella di limitare la validità di un legame di nome, cioè dire dove è visibile un certo nome

3.7.1 Scope

Si definisce scope o ambito di validità di un legame, l'insieme degli enunciati del programma in cui quel legame è valido. I linguaggi moderni limitano lo scope usando blocchi di codice.

3.7.2 Definizione di un nostro pseudo-linguaggio

Introduciamo un semplice pseudo-linguaggio di blocchi di codice.

Un blocco contiene due parti:

- Una sezione di dichiarazione di nomi.
- Una sezione che comprende gli enunciati sui quali hanno validità quei legami di nome.

```
...
BLOCK A;
DECLARE I;
BEGIN A
...
END A;
...
```

Il senso di un blocco è che la variabile I è locale al blocco A.

3.7.3 Scoping statico e scoping dinamico

Abbiamo due tipi di scoping:

- **Scoping statico** o lessicale: blocchi annidati vedono e usano i legami dei blocchi più esterni e possono aggiungere legami locali o sovrapporli a nuovi. Ad esempio una funzione in c vede le sue variabili locali e anche le variabili globali.
- **Scoping dinamico:** quello che vedo ad un certo punto nell'esecuzione del programma non dipende dalla sintassi del programma, cioè a quale scope è annidato il blocco, ma dipende dallo stack delle chiamate, quindi se io sono in g e g è stata chiamata da f vedo lo scope di f, non perché g è stata definita in f ma perché è stata chiamata da f.

3.7.4 Mascheramento

Parliamo di mascheramento o shadowing quando un blocco interno ridefinisce un nome che era già definito in un blocco esterno, quindi la nuova definizione nasconde la vecchia finché non si esce dal blocco interno. Vediamo un esempio:

```
void f(int n) {
    int i = 0;
    {
        int n = 0; // maschera n
        n++;       // incrementa la nuova n
    }
    n--;         // decrementa la vecchia n (il par. formale)
}
```

3.8 Legami di locazione

Stabilire un legame di locazione si chiama **allocare memoria**, valgono le seguenti regole:

- Se una variabile è in scope, cioè accessibile in un certo punto del programma, in quel momento deve avere un legame di locazione.
- Non vale il viceversa, cioè un legame di locazione può essere attivo mentre la variabile non è in scope (ad esempio il mascheramento).

Quindi lo scope di una variabile è un sottoinsieme dell'ambito di validità del suo legame di locazione:(Mo ti spiego) cioè dobbiamo distinguere la **vita** di una variabile, quindi quando ha un legame di locazione e quando non ce l'ha più, poi all'interno di questa regione abbiamo lo scope, quindi durante la sua *vita* talvolta la variabile è utilizzabile e talvolta no.

3.8.1 Tecniche di allocazione

Abbiamo tre tipi di legami di locazione:

- **Allocazione statica:** l'indirizzo viene fissato a load-time e tale allocazione vale per tutta l'esecuzione.
- **Allocazione dinamica:** gli altri due tipi li chiamiamo dinamica ma vediamo che abbiamo due tipologie di allocazione dinamica (di solito per allocazione dinamica intendiamo la seconda che ora vediamo):
 - **Su stack:** l'indirizzo viene fissato a runtime su richiesta (ad esempio l'inizio di un blocco, tipicamente all'ingresso di una funzione) e viene rilasciato al termine del blocco.

- **Su heap:** l'indirizzo viene fissato a runtime su richiesta e rilasciato su richiesta oppure automaticamente quando non più utilizzato (garbage collection).
Quindi se abbiamo una variabile che ci serve solo in un blocco non dobbiamo preoccuparci della deallocazione, se invece la variabile ci serve in più blocchi allora dobbiamo allocarla dinamicamente sull'heap e preoccuparci della deallocazione.

Allocazione statica di memoria

Si dice allocazione statica di memoria quando il legame di locazione è fissato e costante al tempo di caricamento. Vediamo un esempio in C:

```
int g; // Allocazione statica e scope globale

void f(int n) {
    static int a; // Allocazione statica e scope locale
    int i;         // Allocazione dinamica su stack e scope locale
    ...
}
```

Quindi vediamo che una variabile che ha scope locale può essere allocata sia staticamente che dinamicamente sulle stack, quindi in questo esempio abbiamo a e g che sono state allocate staticamente mentre i è allocata dinamicamente

Allocazione dinamica su stack

Il legame di locazione è creato all'inizio dell'esecuzione di un blocco e viene rilasciato automaticamente a fine blocco, essa è realizzata attraverso il **record di attivazione** di un blocco. Un record di attivazione contiene tutte le informazioni sull'esecuzione del blocco necessarie per riprendere l'esecuzione dopo che essa è stata sospesa. In ogni momento dell'esecuzione lo stack di attivazione contiene i record attivi, il top dello stack contiene sempre il record del blocco corrente in esecuzione, questo serve per i blocchi annidati.

Vedere da pagina 112 degli esempi e degli esercizi.

3.8.2 Record di attivazione di un blocco anonimo

Il contenuto di un RDA in questo caso è:

- Puntatore di catena dinamica (link all'RDA precedente).
- Ambiente locale (variabili locali e spazio per risultati intermedi).

Nel caso di un blocco più complesso l'RDA è più complesso, lo vedremo in futuro.

3.9 Implementazione efficiente dell'ambiente non locale con scoping statico

Vediamo adesso un implementazione più efficiente dell'ambiente non locale con scoping statico in presenza di procedure annidate. Nell'implementazione che abbiamo visto quando una variabile non si trova nell'ambiente locale il compilatore deve visitare l'intero stack finché non trova la variabile, quindi l'accesso ad una variabile non locale non avviene in tempo costante. Questa implementazione sostituisce la catena statica e permetterà di accedere alle variabili non locali in tempo costante.

3.9.1 Annidamento

```
program p;
var a,b,c int;

procedure q;
var a,c int;

procedure r;
var a int;
...
...

procedure s;
var b int;
...
...
...
```

- Una procedura q è **annidata** in un blocco b se è definita dentro b, ad esempio:
 - q e s sono annidate in p
 - r è annidata in q
- Il **livello di nesting di una procedura** è il numero di blocchi che la contengono
 - il livello di nesting di q e s è 1
 - quello di r è 2
- **L'ambiente statico non locale** di una procedura è dato dall'ambiente delle procedure in cui è innestata

```

program p;
  var a,b,c int;

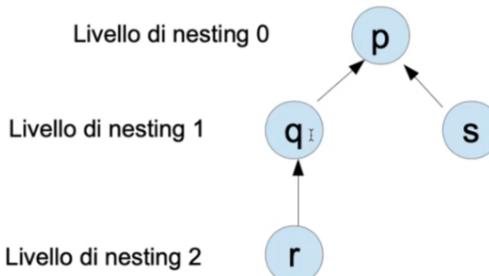
  procedure q;
  var a,c int;

    procedure r;
    var a int;
    ...
  ...

  procedure s;
  var b int;
  ...
  ...

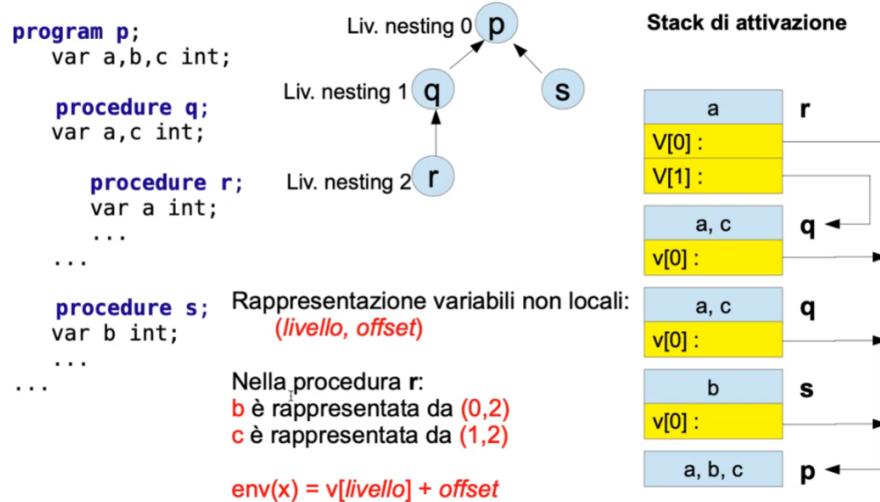
```

- L'annidamento si può rappresentare come un albero
- I livelli dell'albero di nesting corrispondono ai livelli di nesting



Le frecce indicano l'ambiente non locale

3.9.2 Rappresentazione variabili non locali



Nella procedura r b è rappresentata da (0, 2) significa che si trova nel livello 0 ed è la seconda variabile che appare. Quindi l'indirizzo di una variabile non locale x si può ricostruire con:

$$v[\text{livello}] + \text{offset}$$

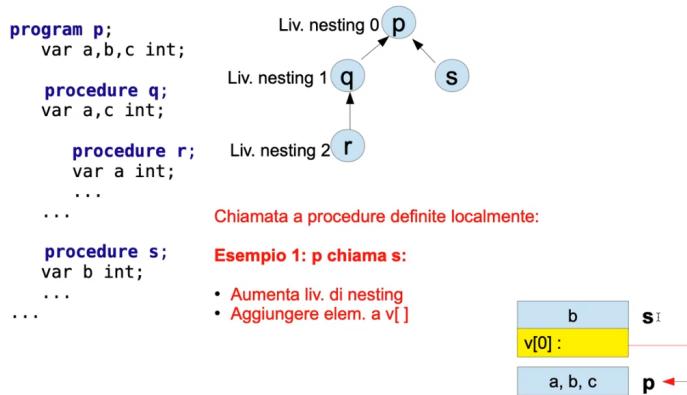
Dove l'array v indica per ogni livello l'indirizzo del record di attivazione più recente della procedura che si trova in quel livello, cioè $v[\text{livello}]$ è l'inizio di un record di attivazione, quindi il più offset lo fa spostare nella variabile giusta. Vediamo come è fatto v, quindi guardiamo lo stack:

- P è il programma e non ha variabili non locale.
- P chiama s, in questo caso s si trova a livello 1, allora ha bisogno solo di $v[0]$, quindi s ha per accedere ad a la coppia (0, 1) e per accedere a c avrà (0, 2).
- Vediamo adesso r, supponiamo che venga nominato c, il compilatore sa che c corrisponde alla posizione (1, 2), quindi dove la procedura r vuole accedere a c viene applicato $v[1] + 2$.

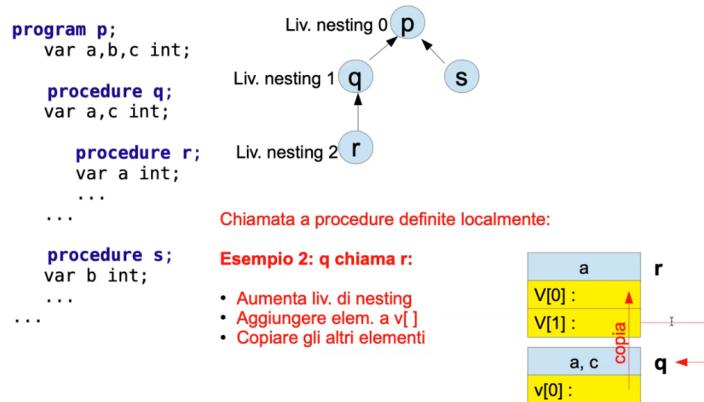
Il compilatore sa che in s se viene chiamata a questa si trova in posizione (0, 1), quello che non sa è $v[0]$ questo viene riempito a run time e poi viene applicata l'operazione quindi al posto di a viene messo $v[0] + 1$. Questi link sostituiscono la catena statica, notiamo che nella catena statica abbiamo un link per record di attivazione, qui invece possiamo avere più puntatori come nel caso di r, però l'accesso alle variabili è costante.

3.9.3 Mantenimento del vettore degli ambienti non locali

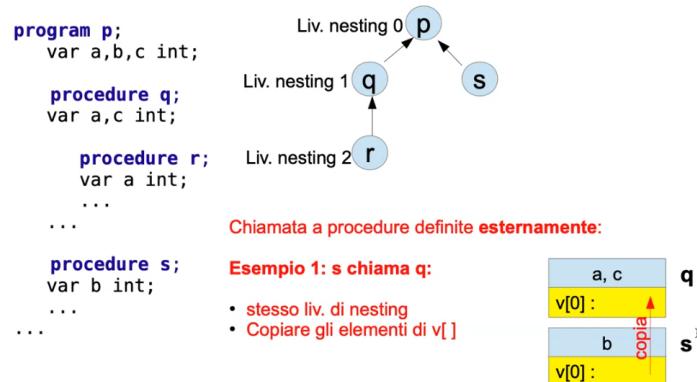
Primo esempio:



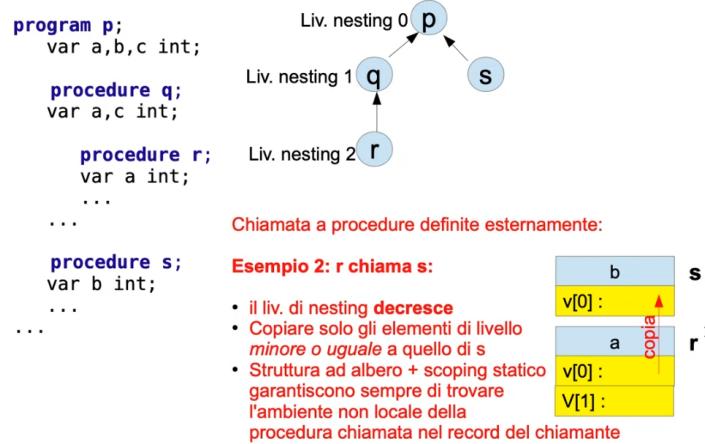
Secondo esempio:



Terzo esempio:



Quarto esempio:



3.9.4 Proprietà di questa implementazione

- Accesso alle variabili in tempo costante
 - Accesso a vettore + somma del puntatore ivi contenuto e dell'offset
 - Indipendente dai livelli di nesting
 - Calcolo supportato direttamente dalle istruzioni macchina
- Creazione record di attivazione lineare nel livello di nesting (copia degli elementi di v[]) (Creation of linear activation record at the nesting level (copy of elements of v[]))
 - Indipendente dall'esecuzione
 - Dipende solo dal testo del sorgente
- **Tempo costante**
- L'implementazione naïve richiederebbe a run time di percorrere le liste di puntatori all'ambiente non locale
 - Accesso alle variabili in tempo **lineare** nel livello di nesting
 - Creazione dei record di attivazione in tempo **costante** nel livello di nesting

Capitolo 4

Procedure

Le **procedure** sono modi di astrarre parti di un programma in unità più piccole (invocazioni), in modo da nascondere i dettagli irrilevanti ai fini del loro riuso. I **vantaggi** sono:

- Programmi più semplici da scrivere, leggere o modificare.
- Suddivisione dei compiti di un programma.
- Progettazione top-down.
- Unità di programmi indipendenti o con dipendenza ben specificate a livello più alto. Cioè un modo di tenere sotto controllo le dipendenze tra diversi pezzi di un programma. Ad esempio se ho una parte di un programma A che non la chiamo più di una volta, c'è ancora motivo per fare una funzione perché magari una parte B ha bisogno dei risultati della parte A per essere eseguita.
- Riusabilità di brani di programmi.
- Riduzione degli errori.

4.1 Astrazione procedurale

In un programma imperativo possiamo distinguere le seguenti unità di esecuzione:

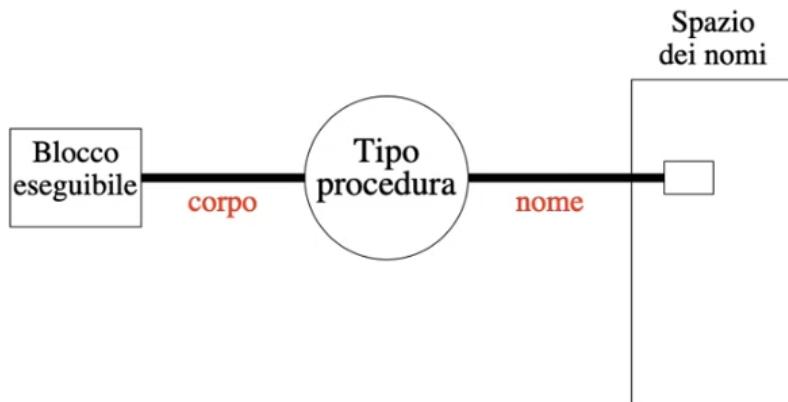
- **Espressioni:** parti di un programma che poi in run time hanno un valore.
- **Enunciato (Statement):** parti di un programma che non hanno un valore, ad esempio la condizione di un if è un'espressione, l'if in se è uno statement.
- **Blocchi:** sequenze di istruzioni.

- **Programmi.**

Un'**astrazione procedurale** è la rappresentazione di una unità di esecuzione, che può essere di varie estensioni (espressioni, statement, ecc), attraverso un'altra unità più semplice, cioè attraverso l'**invocazione** ovvero la chiamata a quella procedura, ogni chiamata alla procedura rappresenta il blocco di codice. In pratica è la rappresentazione di un blocco attraverso un enunciato o una espressione.

4.2 Dichiarazione di procedura

La dichiarazione di una procedura crea un **tipo procedura**, cioè un legame tra il nome della procedura e il corpo della procedura (oltre a parametri, tipi di ritorno, ecc). Questo legame viene stabilito durante la compilazione:



4.3 Invocazione di una procedura

Ogni invocazione diversa della stessa procedura causa la generazione di un nuovo **oggetto procedura** con lo stesso legame di tipo, ma con diverso record di attivazione:



Quindi possiamo notare che lo spazio di tipi procedura è stato costruito a tempo di compilazione ed è statico, mentre ogni chiamata ad una procedura crea un legame tra uno dei tipi di procedura e un nuovo record di attivazione, la stessa procedura può avere diversi record di attivazione se per esempio ha chiamate ricorsive.

4.4 Record di attivazione

Il **record di attivazione** rappresenta l'intero ambiente di esecuzione di una procedura. Vediamo cosa contiene:

- **Variabili locali:** tutte le variabili dichiarate localmente, la dimensione delle variabili è conosciuta a priori dal compilatore, ma dato che è possibile avere variabili o array allocati dinamicamente il compilatore si riserva una parte di memoria libera.
- **Risultati intermedi:** memoria temporanea necessaria alla valutazione delle espressioni contenute nella procedura. Cioè se ho un'espressione del tipo:

$$x = a * b + c$$

Allora il compilatore avrà bisogno di memoria temporanea per eseguire le seguenti istruzioni:

$$tmp = a * b$$

$$x = tmp + c$$

- **Puntatore di catena dinamica:** questo campo serve per memorizzare il puntatore al precedente record di attivazione sulla pila, viene anche detto link dinamico o link di controllo. L'insieme dei collegamenti realizzati da questi puntatori è detto catena dinamica. Quindi serve per implementare i linguaggi con scope dinamico.
- **Puntatore di catena statica:** serve per gestire le informazioni necessarie a realizzare le regole di scope statico. Serve quindi per implementare l'ambiente non locale per quanto riguarda i linguaggi con scope statico.
- **Indirizzo di ritorno:** contiene l'indirizzo della prima istruzione da eseguire dopo che la chiamata alla procedura/funzione è terminata.
- **Indirizzo del risultato:** è presente solo nel caso delle funzioni che devono ritornare un valore, quindi contiene l'indirizzo della locazione di memoria nella quale il sottoprogramma deposita il valore del risultato della funzione. **Attenzione:** è una locazione di memoria all'interno del RDA del chiamante.
- **Parametri:** sono i valori dei parametri attuali usati nella chiamata della procedura o funzione.

4.5 Implementazione delle regole di scoping

Le regole di scoping servono ad implementare la propagazione dei nomi da uno scope più esterno ad uno più interno. Ricordiamo i tre tipi di scope:

- **Propagazione in ambito statico:** l'ambiente non locale di una procedura è propagato dal programma o dalla procedura che la contiene sintatticamente.
- **Propagazione in ambito dinamico:** l'ambiente non locale di una procedura è propagato dal programma o dalla procedura chiamante.
- **Nessuna propagazione:** non vi è ambiente non locale, viene scoraggiato perché produce **effetti collaterali**.

Per implementare le regole di scoping viene aggiunto al record di attivazione un puntatore al record di attivazione della procedura. Se viene richiesto l'accesso ad un dato che non è definito localmente esso viene ricercato in modo ricorsivo nei record di attivazione precedenti.

4.5.1 Esempio scoping statico con procedure annidate

```
program p;
  var a, b, c: integer;

procedure q;
  var a, c: integer;
  procedure r;
    var a: integer;
    begin r      {variabili: a da r; b da p; c da q;
                  ...          procedure: q da p; r da q}
    ...
  end r;
  begin q      {variabili: a da q; b da p; c da q;
                ...          procedure: q ed s da p; r da q}
  ...
end q;

procedure s;
  var b: integer;
  begin s      {variabili: a da p; b da s; c da p;
                ...          procedure: q ed s da p}
  ...
end s;

begin p      {variabili: a, b, c da p;
  ...
end p.
```

ATTENZIONE:

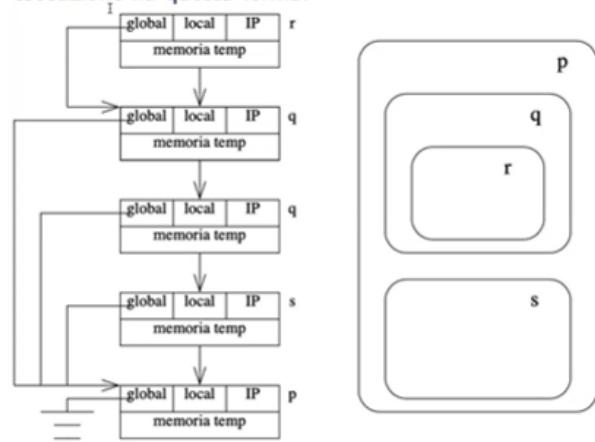
Se viene chiamata q non vuol dire che viene eseguita r.

Aggiungere che r può chiamare s da p.

Generalizzando: il record di attivazione del generico blocco B è collegato dal puntatore di catena statica al record del blocco immediatamente esterno a B. Nel caso in cui B sia il blocco di una chiamata procedura, il blocco immediatamente esterno a B è quello che contiene la dichiarazione della procedura stessa. Inoltre se B è attivo, ossia il suo RDA è sulla pila, allora anche i blocchi esterni a B che lo contengono devono essere attivi e quindi si trovano sulla pila.

Quindi oltre ad una catena dinamica esiste anche una catena statica usata per rappresentare la struttura statica di annidamento dei blocchi nel programma.

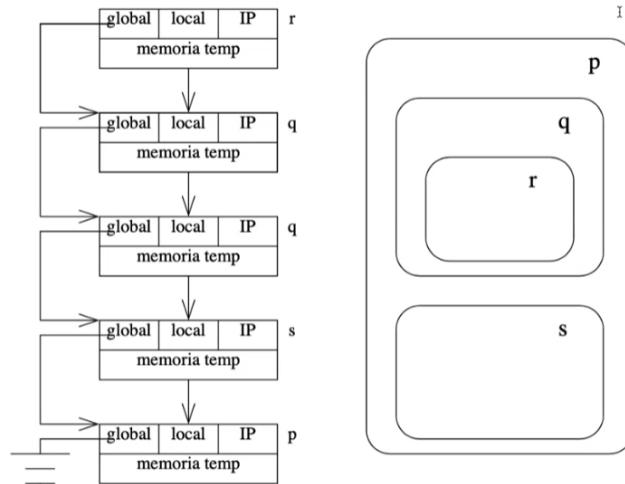
Supponendo una sequenza di attivazione (p, s, q, q, r) , lo stack di esecuzione ha questa forma:



ATTENZIONE: invece di global chiamiamolo non local. Notiamo che r può accedere alle variabili di q ma anche a quelle di p per via del puntatore da q a p .

4.5.2 Esempio scoping dinamico

La stessa sequenza di attivazione precedente (p, s, q, q, r), con dynamic scoping genera allora lo stack di esecuzione:



Notiamo che quindi i collegamenti al centro non servono. Il dynamic scoping è molto più semplice ma è difficile da programmare perché cambia l'ambiente non locale di ogni funzione. Il vantaggio è che se q chiama r allora q non deve passare parametri perché r ha le variabili di q, diventa vantaggioso quando un parametro deve essere passato di funzione in funzione per arrivare ad r, con il dynamic scoping questo è più semplice.

Lo svantaggio è che è impossibile la determinazione dell'ambiente di esecuzione di una procedura durante la scrittura del codice sorgente.

```

program p;
  var a: integer;
  procedure q;
    begin q           {vars: a da p o da r; procs: q da p}
    ...
  end q;
  procedure r;
    var a: integer;
    begin r           {vars: a da r; procs: q, r da p}
    ...
  end r;
begin p            {vars: a da p; procs: q, r, da p}
...
end p.

```

ATTENZIONE: q può chiamare anche r da p.

Capitolo 5

Python

È un linguaggio imperativo orientato agli oggetti, interpretato, dinamicamente tipizzato e offre garbage collection. Ha una sintassi snella, non ci sono dichiarazioni di tipi (cosa tipica dei linguaggi di scripting) e l'indentazione è rilevante. Tutti i valori sono oggetti, tutte le variabili sono riferimenti, è come se in java non esistessero i tipi primitivi. Due funzioni built in sono strettamente legate a questa semantica, cioè la funzione **id(<exp>)** che restituisce l'indirizzo dell'oggetto puntato da <exp>, e la funzione **type(<exp>)** che restituisce il tipo dell'oggetto puntato da <exp>. Quindi ovviamente i tipi in python esistono e vengono controllati, possono cambiare dinamicamente ma l'interprete tiene traccia dei tipi di tutti gli oggetti e effettua controlli di tipo. Ad esempio se scrivessi

```
a = 'ciao' + 1
```

L'interprete da errore, perché esegue l'istruzione vede che una stringa non si può sommare con un intero e da errori, invece:

```
if(1 == 2):
    a = 'ciao' + 1
```

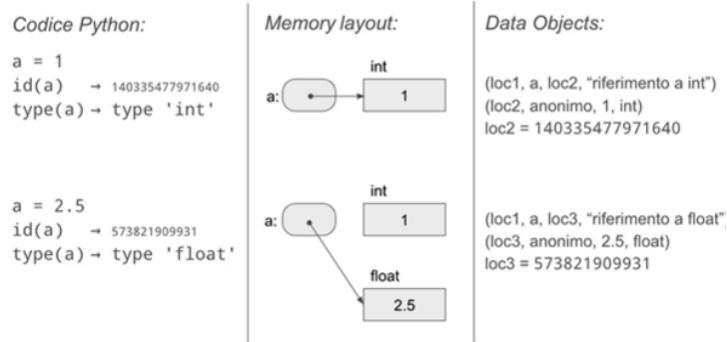
In questo caso l'interprete non arriva alla riga dell'assegnazione e quindi non da errore. Vediamo un data object di Python:

5.1 Sistema dei tipi

Abbiamo a disposizione diversi tipi nativi:

- boolean.
- Numerici: int, float, complex.
- Stringhe: str, bytes, bytearray.
- Collezioni: list, tuple, dict, set, frozenset.

(Locazione, Nome, Valore, Tipo)



L'utente può creare nuove classi e nuove funzioni. le funzioni sono **first-class objects** cioè le funzioni accettano altre funzioni come argomenti e restituire delle funzioni. Lo vedremo meglio in futuro e soprattutto nei linguaggi funzionali.

Gli interi hanno dimensione arbitraria, cioè un intero non ha un limite superiore sul valore che può assumere, possiamo dire che è una scelta di astrazione maggiore rispetto ad altri linguaggi quali C e Java, ovviamente nei calcoli questo è meno efficiente.

5.1.1 Mutabilità e immutabilità

Mutable significa che è possibile cambiare il valore di una variabile senza cambiare l'indirizzo. Il tipo int è immutable, quindi cambiando valore viene cambiato anche l'indirizzo. Attenzione non viene cambiato l'indirizzo della variabile, ma come abbiamo detto prima i valori sono oggetti quindi cambia l'indirizzo a cui punta la variabile che possiamo vedere come un riferimento.

Capitolo 6

Parametrizzazione di procedure

I **parametri** costituiscono il mezzo attraverso il quale le informazioni transitano tra un'unità chiamante e quella chiamata. Alcuni linguaggi distinguono tre categorie:

- **Parametri di ingresso (IN):** hanno lo scopo di passare informazioni dal chiamante al chiamato.
- **Parametri di uscita (OUT):** hanno lo scopo di passare informazioni dal chiamato al chiamante.
- **Bidirezionali (IN-OUT):** servono a far transitare le informazioni in entrambe le direzioni.

I parametri devono essere specificati in due punti:

- Nella definizione della procedura: **parametri formali**.
- Nelle invocazioni della procedura: **parametri attuali**.

6.1 Tipi dei parametri

Nei **linguaggi staticamente tipati**, nella definizione della procedura bisogna specificare il tipo, poi il compilatore controllerà che il tipo dei parametri attuali coincida con quello dei parametri formali.

Nei **linguaggi dinamicamente tipati** invece i parametri formali non hanno alcun vincolo di tipo, a run time viene poi stabilito il tipo del parametro formale in coincidenza del tipo del parametro attuale che viene passato.

6.2 Associazione dei parametri

Abbiamo due **metodi di associazione** tra parametri attuali e formali:

- **Posizione:** a seconda della posizione relativa nella sequenza dei parametri.
- **Nome:** il nome del parametro formale è aggiunto come prefisso al parametro attuale.

Vediamo un esempio:

```
procedure TEST (A: in Atype; b: in out Btype; C: out Ctype)
```

allora una invocazione che usa associazione per posizione è:

```
TEST(X, Y, Z);
```

mentre una che usa associazione per nome può essere:

```
TEST(A=>X, C=>Z, b=>Y);
```

6.2.1 Associazione di default

Permette di associare dei valori di default a dei parametri formali che non sono stati legati a valori da parametri attuali. Quindi se ho due parametri formali e ad uno di loro è stato definito un valore di default, la procedura può essere chiamata o con un solo parametro attuale o con due.

6.3 Parametri IN

Possono essere realizzati in due modi:

1. **Per riferimento:** in questo caso viene passato l'indirizzo del parametro attuale, poiché il parametro formale è di tipo IN, allora si deve impedire la modifica all'interno della procedura.
2. **Per copia:** il valore del parametro attuale viene copiato nel parametro formale, in questo caso la procedura non può cambiare il valore del parametro attuale, quindi la modifica all'interno della procedura è permessa.

Il secondo modo è meno efficiente del primo, sia rispetto allo spazio sia al tempo, ma garantisce automaticamente che il parametro attuale non sia modificato.

6.4 Parametri OUT

Nell'accezione precisa di parametro out alla funzione dovrebbe essere impedito di leggere il parametro passato, altrimenti ci troveremo nel caso di un parametro IN-OUT. In ogni caso la realizzazione può essere fatta per:

- **Copia:** non abbiamo una copia in ingresso ma in uscita, cioè il chiamante chiama la procedura e viene allocato lo spazio del parametro passato nel suo ambiente locale (quello del chiamato), non viene però fatta la copia di nessun valore, questo spazio allocato viene poi riempito e poi in uscita avviene la copia dall'ambiente locale del chiamato a quel parametro che viene nominato nella chiamata. Per la realizzazione per copia non è possibile passare da una procedura un parametro che è stato passato col metodo OUT ad una procedura che riceve un parametro IN o IN OUT.
- **Riferimento.**

6.5 Parametri IN-OUT

Sono la combinazione dei due precedenti. Anche loro possono essere passati per riferimento, in questo caso non ci sono limitazioni, o per copia, in questo caso avvengono due processi di copia, in ingresso ed in uscita.

Vediamo adesso degli esempi in vari linguaggi:

Linguaggio	IN	OUT	IN-OUT
Ada	Sì	Sì	Sì
Pascal	Sì	No	Sì (keyword var)
C	Sì	No	No. Emulato con puntatori
C++	Sì	No	Sì (con riferimenti)
Java ¹	Sì	No	No. Emulato con oggetti mutabili
Python ²	Sì	No	No. Emulato con oggetti mutabili

Nota 1: in Java, tutti i parametri, primitivi e non, vengono passati in modalità “IN per valore”, ma le variabili di tipo non primitivo sono riferimenti.

Nota 2: in Python, tutte le variabili sono riferimenti, e vengono passati in modalità “IN per valore”.

In Java se ad esempio vogliamo passare un valore intero in-out, il valore non

viene modificato, perché la classe Integer non è mutabile (in generale le classi Wrapper), per farlo dovremmo creare una classe MyInteger.

6.6 Aliasing

È la possibilità di riferirsi alla stessa locazione con nomi diversi. Capita spesso con linguaggi che offrono puntatori o riferimenti, ma anche se non li offrissero può capitare se il linguaggio supporta alcuni tipi di passaggi di parametri, e spesso può causare problemi:

```
program MAIN;
var
  A: integer;
procedure TEST (var X, Y: integer);
begin
  X := A + Y;
  writeln(A, X, Y)
end;
begin
  A:= 1;
  TEST(A, A)
end.
```

Esercizio: determinare l'uscita del programma nel caso in cui i parametri VAR siano realizzati *per riferimento* e nel caso in cui siano realizzati *per copia*.

Nel caso di in-out per riferimento il risultato è 2, 2, 2 e A nel chiamante diventa 2.

Nel caso di in-out per copia il risultato è 1, 2, 1 e A nel chiamante rimane 1.

6.7 Procedure come parametri di procedure

Alcuni linguaggi permettono l'uso di procedure come argomento di altre procedure. Esempio:

```
program MAIN;
  VAR a: real;
  procedure TESTPOS (X: real; procedure ERROR (MSG: string));
    begin
      if X <= 0 then ERROR ('Negative X in TESTPOS')
    end;
  procedure E1 (M: string);
    begin
      writeln('E1 error: ', M)
    end;
  procedure E2 (M: string);
    begin
      writeln('E2 error: ', M)
    end;
begin
  readln (A);
  TESTPOS(A, E1);
  TESTPOS(A, E2)
end.
```

6.8 Macro

Una macro è la generazione di un nuovo brano di codice sorgente in cui i nomi dei parametri attuali sostituiscono i nomi dei parametri formali. Per esempio:

```
procedure swap (a, b: integer);
  var temp: integer;
begin
  temp:= a;
  a:= b;
  b:= temp
end;
```

allora la chiamata `swap(x, y)` esegue il seguente brano di codice:

```
temp:= x;
x:= y;
y:= temp;
```

Le macro possono creare alcuni problemi:

```
program main;
var
  i: integer;
  m: array[1..100] of integer;
  ...
begin
  ...
  swap(i, m[i]); i
  ...
end.
```

```
program main;
var
  i, temp: integer;
  ...
begin
  ...
  swap(i, temp);
  ...
end.
```

Il primo esempio fa questo:

```
temp = i;
i = m[ i ];
m[ i ] = temp;
```

Lo capisci non te lo devo spiegare.

Il secondo esempio invece:

```
temp = i;
i = temp;
temp = temp;
```

Te lo devo spiegare ?

6.9 Funzioni

Sono procedure che restituiscono un valore alla procedura chiamante, possono essere realizzate:

- O creando una pseudovariabile nell'ambiente locale della procedura chiamata. Tale variabile può essere solo modificata, non è possibile l'accesso in lettura.
- O utilizzando un istruzione di return per restituire esplicitamente il controllo alla procedura chiamante inviandole allo stesso tempo il valore di una espressione.

Capitolo 7

Paradigma ad oggetti

Adesso iniziamo ad analizzare il paradigma ad oggetti, andando nello specifico del linguaggio Java.

7.1 Il modificatore static

Questo **modificatore** si può applicare a vari elementi sintattici del linguaggio:

1. **Attributi.**
2. **Blocchi di inizializzazione.**
3. **Metodi.**
4. **Classi interne.**

Vediamo i primi tre poi in futuro vedremo il quarto caso.

7.1.1 Attributi

Partiamo da un esempio:

```
Class A{  
    private static int n;  
    private String name;  
}
```

La differenza è che ciascuna istanza di A ha l'attributo nome, ma esiste un unico attributo n, poi l'allocazione di n è statica e viene allocata in **load time** e non risiede nel memory layout delle altre istanze. Quindi tutti gli oggetti della classe A fanno riferimento allo stesso attributo n. In Java le classi vengono caricate dinamicamente quindi il load time è intrecciato al run time, quando avviamo il programma viene caricata la prima classe e poi a cascata le altre che vengono invocate, tornando al discorso degli attributi statici, ciascuna variabile statica

di una classe viene allocata quando la classe a cui appartengono viene caricata.
Un esempio di uso di n è un contatore di istanze di A:

```
public A( String name){  
    this.name = name;  
    n++;  
}
```

(Il parametro String name sta mascherando l'attributo name).
Un'altra applicazione è quella di avere delle costanti statiche:

```
private final static int MAX = 100;
```

È possibile accedere ad un'attributo statico anche in questo modo:

```
A.n++;
```

Se n fosse pubblico, potrei accedere ad n ovunque e quindi avere una variabile globale, questo è molto pericoloso (**RIO DOCET**).

È possibile fare:

```
Double d = 3.0;  
d.POSITIVE_INFINITY;
```

e l'ultima riga equivale a:

```
Double.POSITIVE_INFINITY;
```

Cioè sto accedendo ad una costante statica della classe Double istanziando un oggetto Double, questa cosa si può fare ma non è buona norma.

7.1.2 Blocchi di inizializzazione

```
private static int[] a = new int[100];  
static{  
    for (int i = 0; i < 100; i++){  
        a[i] = i;  
    }  
}
```

Abbiamo bisogno di un inizializzazione dell'array e per farlo usiamo un blocco di codice statico che viene eseguito una sola volta al load della classe. Il senso di questo costrutto è quello di inizializzare i campi statici, ma in ogni caso possiamo scrivere quello che vogliamo. Dal blocco statico posso accedere alle variabili locali e solo alle variabili statiche.

7.1.3 Metodi statici

Facciamo direttamente un esempio:

```
Math.sin / cos ...
```

Nella classe Math avremo:

```
public static Double cos(Double x) ...
```

Quindi un metodo statico non è legato ad un'istanza e per invocarlo si usa il nome della classe, nell'ambiente locale del metodo quindi non esiste this dato che non viene invocato da un'istanza.

Vediamo di analizzare un'altra invocazione ad un metodo:

```
System.out.println("ciao");
```

System è una classe, out è un campo statico di System di tipo PrintStream quindi nella classe System avremo:

```
public final static PrintStream out;
```

Mentre println è un metodo non statico perché è legato ad un'istanza.

7.2 Scoping in Java

Lo scoping in Java è statico, vediamo direttamente un esempio:

```
n = 7;
```

Ci troviamo in un metodo, nell'ordine il compilatore cerca n in questo modo:

1. Ambiente locale.
2. Tra gli attributi della classe corrente.
3. Può trovarsi in una superclasse ed è soggetto a visibilità, se nella superclasse n è privata allora i figli non possono accedervi.

In linea generale il compilatore cerca una variabile in quest'ordine:

1. Blocco corrente.
2. Blocchi esterni.
3. Metodo corrente.
4. Classe corrente.
5. Superclassi.

Questo schema vale solo per nomi non qualificati cioè semplici dove ho solo n, se ad esempio avessi:

```
this.n = 7;
```

Il compilatore partirebbe dal caso 4.

Vediamo un altro esempio:

a . b ;

Digressione:

Object x = "Ciao";

Object è il tipo dichiarato o statico, "Ciao" è il tipo dinamico o effettivo.

Tornando all'esempio, il compilatore seguendo i passi da 1 a 5 identifica il tipo dichiarato di a che adesso chiamiamo A, per b invece il compilatore segue i passi da 4 a 5 ma applicati ad A.

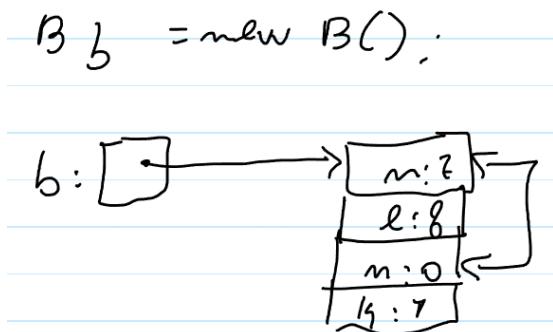
7.2.1 Memory Layout e mascheramento con ereditarietà

Partiamo con un esempio:

```
Public class A{  
    public int n = 0;  
    public int k = 1;  
    public void f(){...}  
}
```

```
Public Class B extends A{  
    public int n = 7;  
    public int l = 8;  
    public void f(){...}  
}
```

Il memory layout per b è:



Quindi ci sono le due n ma da B è mascherata la n di A, ma esiste nel memory layout di B, in pratica b.n sta mascherando a.n. Con super.n posso anche accedere alla n di A.

Se gli attributi di A fossero privati invece il memory layout di B è lo stesso ma non c'è modo per accedere alla n di A, quindi b.n non sta mascherando a.n perché a.n non è in scope.

Per quanto riguarda il metodo invece, viene applicato l'override e viene risolto a tempo di esecuzione.

7.3 Regole di visibilità

Esempio: nella classe A

id1.id2

Chiamiamo:

- **B1:** tipo dichiarato di id1.
- **B2:** classe in cui è dichiarato id2.
- **V:** visibilità di id2 in B2.

B2 o è B1 oppure è super di B1, vediamo le visibilità:

- **V è private:** solo se A = B2.
- **V è default:** A e B2 si trovano nello stesso package.
- **V è protected:** A e B2 si trovano nello stesso pacchetto, oppure A è sottoclasse di B2 e B1 è sottoclasse di A.
- **V è public:** nessun vincolo.

Vediamo un esempio di protected:

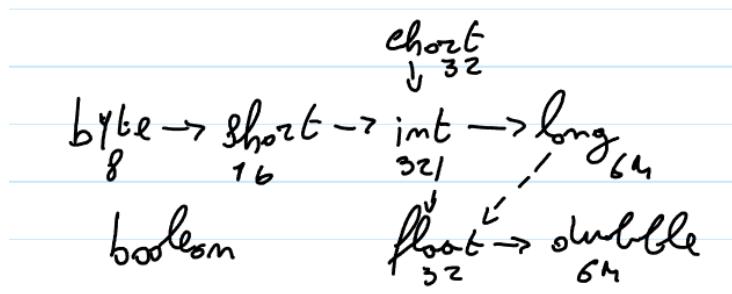
```
package uno;
class X{
    protected int n;
}
package due;
class Y extends X{ \\A = Y
    void f(X x, Y y){
        this.n = 3; \\B1 = Y, B2 = X OK
        x.n = 4;     \\B1 = X, B2 = X ERRORE
        y.n = 5;     \\B1 = Y, B2 = X OK
    }
}
```

7.4 Il sistema dei tipi

Java fornisce i seguenti tipi:

- Tipi base o primitivi.
- Tipi riferimento.
 - Array.
- Tipo nullo.

7.4.1 Tipi primitivi



Le frecce indicano che c'è una **conversione implicita** tra un tipo e l'altro, cioè ad esempio un valore di una variabile **byte** può essere assegnato ad una variabile di tipo **short** e sono transitive. Alcune di queste frecce però possono creare problemi, ad esempio da **long** a **float** vi può essere perdita di informazione perché long ha più bit. Anche da **int** a **float** è pericoloso, perché i float hanno un problema di risoluzione cioè hanno una risoluzione diversa a magnitudine diversa, vicino allo 0 hanno una risoluzione molto fine ma per numeri grandi è come se si sgranassero, per risoluzione intendo la distanza da un numero all'altro, quando il numero è vicino allo 0 la distanza è piccola, quando il numero è molto grande la distanza può essere anche maggiore di 1, quando si arriva a questo punto iniziano ad esserci dei buchi negli interi, cioè alcuni interi non si possono rappresentare in float.

7.4.2 Relazione di sottotipo

Tra tipi primitivi valgono le conversioni implicite, nei tipi non primitivi invece vale la **relazione di sottotipo**.

$\forall S, T$ non primitivi:

1. S è sottotipo di se stesso, (**riflessività**).
2. S è sottotipo di Object.

3. Se S estende o implementa T (anche indirettamente) allora S è sottotipo di T .
4. Il tipo null è sottotipo di S .
5. Se S è sottotipo di T allora, array di S è sottotipo di array di T .

È una relazione antisimmetrica, transitiva e riflessiva e quindi è una relazione d'ordine parziale tra tipi non primitivi.

Le conversioni implicite e la relazione di sottotipo definiscono le compatibilità tra tipi in Java.

7.4.3 Assegnabilità

$\forall S, T : S$ è assegnabile a T sse S, T sono primitivi e $S \rightarrow T$, oppure, S è sottotipo di T .

Applicazione all'assegnabilità

1. **id = exp**: quando il tipo stativo di exp deve essere assegnabile al tipo dichiarati di id.
2. **return exp**: il tipo dichiarato di exp deve essere assegnabile al tipo di ritorno del metodo corrente.
3. **x.f(exp)**: si parte dal tipo dichiarato di x, si trova un metodo f con un parametro, e ciascun tipo dichiarato di ogni parametro attuale deve essere assegnabile al corrispondente parametro formale, (ci sarà poi una lezione su questo).

7.4.4 Operatore instanceof

È un operatore binario infisso, cioè ha due argomenti e l'operatore si scrive tra i due. Vediamo un esempio:

```
f( Object o){
    if(o instanceof String){}
}
```

Quindi a sinistra ho un **espressione (exp, un tipo non primitivo)** e a destra ho una Classe (S). Quindi permette di sapere il tipo effettivo dell'exp. Precisamente restituisce true se il tipo effettivo di exp non è nullo ed è sottotipo di S.

7.4.5 Array e controllo dei tipi

Vediamo se il seguente codice compila e se da errori a run time:

```

String [] arr1 = new String [10];
Object [] arr2 = arr1;
arr2 [0] = new Object ();
String s = arr1 [0];

```

La prima riga non ci sono dubbi, la seconda riga per la regola 2 e la 5 è corretta, nella terza riga il tipo dichiarato di arr2[0] è Object e quindi è giusta, infine nella quarta riga non ci sono problemi. Quindi il codice viene compilato, ma solleva un'eccezione alla terza riga: notiamo che abbiamo un solo array e il suo tipo dichiarato è String e ogni volta che si modifica una cella di un array viene controllato che il tipo effettivo che si vuole assegnare corrisponde al tipo dichiarato del array. L'eccezione che viene sollevata è **ArrayStoreException**.

7.4.6 Cast

Cast tra tipi primitivi:

- Si può utilizzare un cast per effettuare esplicitamente una **promozione**
 - in questo caso il cast è superfluo

- Si può utilizzare un cast per effettuare una **promozione al contrario**
 - ad esempio, da double a int
 - in questi casi, è facile incorrere in *perdite di informazioni*
 - ad esempio, nel passaggio da numeri in virgola mobile a numeri interi, si può perdere **sia in precisione che in magnitudine** (ordine di grandezza)
 - i dettagli sono definiti nella sezione 5.1.3 del JLS: Narrowing Primitive Conversion
 - queste conversioni sono decisamente sconsigliate, al loro posto è opportuno utilizzare i metodi appropriati della classe Math (come Math.round)

Cast tra riferimenti:

- Sono **consentiti** dal compilatore i seguenti cast tra un tipo riferimento (o array) A ad un tipo riferimento (o array) B:
 - 1) se B è **super tipo** di A
 - si chiama "upcast"
 - è superfluo, perché i valori di tipo A sono di per sé assegnabili al tipo B
 - 2) se B è **sottotipo** di A
 - si chiama "downcast"
 - al run-time, la JVM controlla che l'oggetto da convertire appartenga effettivamente ad una sottoclasse di B
 - in caso contrario, viene sollevata l'eccezione ClassCastException
 - si deve cercare di evitare i downcast, perché aggirano il type checking svolto dal compilatore
 - a tale scopo, i tipi parametrici introdotti da Java 1.5 possono aiutare
 - se proprio si deve usare un downcast, esso andrebbe preceduto da un controllo instanceof, che assicuri la correttezza della conversione

- Negli altri casi, il cast porta a un errore di compilazione

Vediamo adesso un esercizio:

Sapendo che sia la classe C sia la classe B estendono A, effettuare il type checking del seguente codice, evidenziando:

- errori di tipo
- cast non validi
- cast validi ma potenzialmente pericolosi al run-time

```
boolean f(A a, B b) {  
    C c = (C) a;  
    A a1 = (A) b;  
    Object o = a;  
    A[] arr = new A[10];  
    arr[5] = (Object) a;  
    arr[6] = b;  
    return a == c;  
}
```

La prima riga compila ed è un downcast, a run time dipende dal tipo effettivo di a se passo come tipo C allora viene eseguito. È un upcast ed è ridondante il compilatore non lo controlla e viene eseguito. Terza riga compila e viene eseguita. Quarta riga va tutto bene. Quinta riga è un upcast ma provoca un errore di compilazione perché non posso assegnare un Object ad arr[5]. Sesta riga compila perché b è sottotipo di A e anche a run time va. Settima riga è vera perché a c assegno a.

7.4.7 Wrapper

- Per ogni tipo base, Java offre una corrispondente classe, che ingloba (*wraps*) un valore di quel tipo in un oggetto
- Queste classi, dette appunto wrapper, servono a trattare i valori base come se fossero oggetti
 - ad esempio, per inserirli nelle strutture dati offerte dalla Java Collection Framework (vedi lezioni seguenti)
- Le classi wrapper sono:
 - Byte, Short, Integer, Long, Float, Double
 - Boolean
 - Character
 - Void
- Come si vede, sono tutte omonime del rispettivo tipo base, tranne Integer, Character, e Void (perché formalmente void non è un tipo base)

- Tutte le classi wrapper sono **immutabili e final**
- Ogni classe wrapper ha un metodo statico **valueOf** che prende come argomento un valore del tipo base corrispondente alla classe e restituisce un oggetto wrapper che lo ingloba
- Esempio:

```
Integer n = Integer.valueOf(3);
Double x = Double.valueOf(3.1415);
```
- A differenza di un costruttore, l'oggetto restituito da `valueOf` non è necessariamente nuovo
 - ovvero, il metodo `valueOf` cerca di riciclare gli oggetti già creati (*caching*)
 - questo non è un problema, perché gli oggetti wrapper sono immutabili

Per una classe essere **final** significa che non può avere sottoclassi. **Immutabile** significa che non può essere modificato il contenuto degli oggetti.

Classi wrapper numeriche

- Le sei classi wrapper relative ai tipi numerici estendono la **classe astratta Number**
- La classe `Number` prevede sei metodi, che estraggono il valore contenuto, convertendolo nel tipo base desiderato

```
public byte     byteValue()
public short    shortValue()
public int      intValue()
public long     longValue()
public float    floatValue()
public double   doubleValue()
```
- Se un oggetto wrapper viene convertito in un valore base verso il quale non c'è una conversione implicita (ad es., da `double` a `int`), l'effetto sarà lo stesso di quello di un cast

Tipi wrapper ed uguaglianza

- L'operatore `"=="` può dare risultati sorprendenti se applicato ai tipi wrapper

- Infatti, è facile dimenticare che si tratta di un **confronto tra riferimenti**, come per tutti gli oggetti

- Ad esempio:

```
Integer a = new Integer(7), b = new Integer(7);
System.out.println(a==b); // false, sono oggetti diversi

Integer a = 7, b = 7;
System.out.println(a==b); // true, perché viene chiamato il metodo statico valueOf,
                        che riutilizza gli oggetti corrispondenti a valori piccoli

Integer a = 700, b = 700;
System.out.println(a==b); // false, perché il metodo valueOf riutilizza soltanto
                        gli interi compresi tra -127 e 127
```

- Morale: confrontare i tipi wrapper con **equals** e non con `"=="`

Il metodo `valueOf` per motivi di efficienza riserva delle celle di memoria ai numeri, in questo modo l'accesso all'oggetto è costante e quindi riutilizza soltanto gli interi compresi tra -127 e 127 perché altrimenti allocherebbe troppa memoria.

7.4.8 Autoboxing

- Fino alla versione 1.4 di Java, era necessario convertire esplicitamente i valori dei tipi base in oggetti e viceversa

- A partire dalla versione 1.5, questo procedimento è stato automatizzato, introducendo l'**autoboxing** e l'**auto-unboxing**

- Grazie a queste funzionalità, il compilatore si occupa di inserire le istruzioni di conversione laddove queste siano necessarie

- Ad esempio, l'istruzione

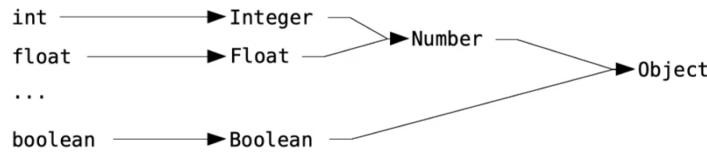
```
Integer n = 7;
viene convertita in:
Integer n = Integer.valueOf(7);
```

- Allo stesso modo, le istruzioni:

```
Integer n = 7;
Integer i = n + 7;
vengono convertite in:
Integer n = Integer.valueOf(7);
Integer i = Integer.valueOf(n.intValue() + 7);
```

L'autoboxing e l'autounboxing sono a carico del compilatore. L'autoboxing può convertire un'espressione di tipo primitivo in un oggetto del tipo wrapper corrispondente, o di un suo supertipo, quindi è possibile effettuare le seguenti conversioni:

L'autoboxing non prende in considerazione le conversioni implicite, ad esempio ciascuna delle seguenti istruzioni provoca un **errore di compilazione**,



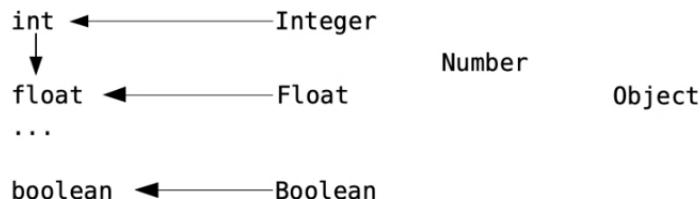
perché oltre all'autoboxing richiederebbe anche una conversione implicita di tipo (promozione):

```

Double x = 1;
Double y = 1.0 f;
Integer n = (byte) 1;
  
```

7.4.9 Auto-unboxing

È possibile effettuare le seguenti conversioni:



Ad esempio le seguenti istruzioni sono corrette:

```

Integer i = 7; //autoboxing
double d = i //auto-unboxing seguito da promozione
  
```

L'auto-unboxing di un'espressione che a runtime risulta **null** comporta il lancio di un'eccezione.

7.5 Risoluzione overloading e overriding

Il binding dei metodi è **dinamico**, per **binding dinamico** si intende il meccanismo per cui non è il compilatore, ma la JVM ad avere l'ultima parola su **quale metodo invocare** in corrispondenza di ciascuna chiamata a metodo. In altri termini si tratta di stabilire il legame di locazione di un nome di metodo nel contesto di una invocazione, per esempio:

x.f(exp)

Nei capitoli precedenti abbiamo visto l'identità di x e in quel caso vi è un binding statico e l'identità viene stabilita a tempo di compilazione, adesso vediamo il binding dinamico di x.f la cui identità viene stabilita a tempo di esecuzione. Il fatto che il binding di x.f sia dinamico è una diretta conseguenza del polimorfismo e dell'overriding, ovvero ciascun riferimento può puntare ad oggetti di tipo effettivo diverso (polimorfismo) e ciascuno di questi tipi effettivi può prevedere una versione diversa dello stesso metodo (overriding). Inoltre il compilatore non può prevedere di che tipo effettivo sarà una variabile nel corso dell'esecuzione del programma (problema indecidibile).

7.5.1 Le fasi del binding

In java il binding dei metodi avviene in due fasi:

1. **Early binding:** in cui il **compilatore** risolve l'overloading (scegliendo la firma più appropriata alla chiamata).
2. **Late binding:** in cui la **JVM** risolve l'overriding (scegliendo il metodo vero e proprio).

Il late binding non è necessario per quei metodi che non ammettono overriding: private, statici o final. Per questi metodi si parla di binding statico perché la scelta del metodo da eseguire viene fatta già dal compilatore.

Fasi dell'early binding

L'early binding si divide a sua volta in due fasi:

1. Individuazione delle firme candidate, l'output di questa fase è una lista di firme.
2. Scelta della firma più specifica tra quelle candidate.

Vediamo la **prima fase**. Consideriamo una generica invocazione:

x.f(a₁, ... , a_n)

E consideriamo una generica firma di f:

f(T₁, ... , T_n)

Una firma è candidata se:

- Si trova nella classe dichiarata di x o in una sua superclasse.
- È visibile dal punto della chiamata, rispetto alle regole di visibilità Java.
- È compatibile con la chiamata, ovvero, per ogni indice i compreso tra 1 ed n, il tipo dichiarato del parametro attuale ai è **assegnabile** al tipo Ti.

Se nessuna firma risulta candidata per una data chiamata, il compilatore segnala un errore.

Vediamo adesso la **seconda fase**. Date due firme con lo stesso nome e numero di argomenti, si dice che la prima è **più specifica** della seconda se, per ogni indice i compreso tra 1 ed n, il tipo Ti della prima firma è assegnabile al tipo Ui della seconda firma. Notiamo che questo confronto tra firme non dipende dal tipo dei parametri attuali passati alla chiamata. È possibile verificare che essere più specifico è una relazione riflessiva, antisimmetrica e transitiva, proprio come la relazione di assegnabilità, quindi è una relazione d'ordine parziale tra firme. Per questo motivo le firme f(int, double) e f(double, int) non sono confrontabili quanto a specificità, int è assegnabile a double ma double non è assegnabile a int.

L'early binding si conclude individuando, tra le firme candidate una che sia **più specifica di tutte le altre**. Nei casi complessi è utile disegnare il **grafo** in cui ci sia un nodo per ciascuna firma candidata ed un arco orientato ad un nodo a ad un nodo b quando la firma a è più specifica della firma b, mentre per le firme non confrontabili si disegna un arco tratteggiato. Se nel grafo c'è un nodo che ha archi uscenti diretti verso tutte le altre firme, quella sarà la firma scelta dal compilatore, se nessuna firma è più specifica di tutte le altre il compilatore segnale una errore e termina. Non è possibile che si trovi più di una firma più specifica perché è antisimmetrica.

Late binding

Il late binding è la fase di risoluzione dell'**overriding**, a carico della **JVM**. Questa fase riceve in input la firma scelta dal compilatore durante l'early binding. Consideriamo una generica invocazione:

x.f(a1, ..., an)

La JVM cerca un metodo da eseguire, con il seguente algoritmo:

- Si parte dalla classe effettiva di x.
- Si cerca un metodo che abbia la firma **identica** a quella scelta dall'early binding se non lo si trova, si passa alla superclasse.
- Così via, fino ad arrivare ad Object.

Questo procedimento può fallire, cioè non trovare alcun metodo, solo in casi molto particolari, come ad esempio se una classe A dipendeva da una classe B e la classe B è cambiata da quando è stata compilata A.

Esercizio

- Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(double n, A x) { return "A1"; }  
    public String f(double n, B x) { return "A2"; }  
    public String f(int n, Object x) { return "A3"; }  
}  
class B extends A {  
    public String f(double n, B x) { return "B1"; }  
    public String f(float n, Object y) { return "B2"; }  
}  
class C extends A {  
    public final String f(int n, Object x) { return "C1"; }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = new B();  
        A alfa = beta;  
        System.out.println(alfa.f(3, beta));  
        System.out.println(alfa.f(3.0, beta));  
        System.out.println(beta.f(3.0, alfa));  
        System.out.println(gamma.f(3, gamma));  
        System.out.println(false ||  
            alfa.equals(beta));  
    }  
}
```

- Indicare l'output del programma
- Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione
- Per ogni chiamata ad un metodo (escluso System.out.println), indicare la lista delle firme candidate

-
- Esaminiamo le chiamate una per volta

1) System.out.println(alfa.f(3, beta));

- alfa è di tipo dichiarato A, quindi le firme candidate vanno cercate nella classe A (o tutt'al più in Object)
- i due parametri attuali della chiamata sono di tipo (dichiarato) int e B, rispettivamente
- la firma f(double, A) è candidata, in quanto visibile e compatibile
 - essa è compatibile perché int è assegnabile a double (conversione implicita) e B è assegnabile ad A (sottotipo)
- la firma f(double, B) è candidata, in quanto visibile e compatibile
- la firma f(int, Object) è candidata, in quanto visibile e compatibile
- non vi sono altre firme candidate
- Delle tre firme candidate, la seconda è più specifica della prima, ma non è confrontabile con la terza
- Quindi, nessuna firma è più specifica di tutte le altre
- Il risultato è un **errore di compilazione**
- ATTENZIONE: ricordate che la scelta della firma più specifica non dipende dal tipo dei parametri attuali della chiamata

- Esaminiamo la seconda chiamata:
- 2) `System.out.println(alfa.f(3.0, beta));`
- alfa è di tipo dichiarato A, quindi le firme candidate vanno cercate nella classe A (o tutt'al più in Object)
 - i due parametri attuali della chiamata sono di tipo (dichiarato) double e B, rispettivamente
 - la firma `f(double, A)` è candidata, in quanto visibile e compatibile
 - la firma `f(double, B)` è candidata, in quanto visibile e compatibile
 - la firma `f(int, Object)` non è candidata, in quanto non compatibile
 - non vi sono altre firme candidate
 - Delle due firme candidate, la seconda è più specifica della prima
 - Quindi, l'early binding si conclude con la selezione della firma `f(double, B)`
 - Per il late binding, cerchiamo il metodo da eseguire a partire dalla classe effettiva di alfa: B
 - Nella classe B, troviamo un metodo visibile con quella firma
 - Quindi, l'output di questa chiamata è

B1

l'ultima istruzione stampa vero.

7.5.2 Binding dinamico e auto-(un)boxing

- Nella risoluzione dell'overloading, l'autoboxing e l'auto-unboxing entrano in gioco *soltanto se necessario*
 - Ovvero, soltanto se altrimenti non ci sarebbero firme candidate
- Quindi, come **primo tentativo**, il compilatore cerca le firme che sono candidate senza prendere in considerazione l'autoboxing e l'auto-unboxing
- **Solo se non ci sono firme candidate**, il compilatore abilita le conversioni da tipo primitivo a tipo wrapper, e viceversa, e riesamina tutte le firme (**secondo tentativo**)
- Questa scelta è stata fatta per mantenere la compatibilità con il codice scritto prima dell'introduzione dell'auto-(un)boxing
- Infatti, le invocazioni a metodo che funzionavano senza auto-(un)boxing continuano a funzionare con l'auto-(un)boxing, e sono risolte nello stesso modo
- Con l'auto-(un)boxing, alcune invocazioni che prima non erano consentite diventano lecite

Una volta ottenuto un insieme non vuoto di firme candidate il compilatore passa alla scelta della più specifica, con le regole descritte prima. Quindi l'auto-(un)boxing **non influenza** in alcun modo la scelta della firma più specifica e analogamente **non influenza** in alcun modo il **late binding**, **influenza** solo la prima sottofase dell'early binding dove aggiunge un secondo tentativo.

Vediamo degli esempi:

- Consideriamo i seguenti metodi:

```
public static int foo(int i, Object o) { return 1; }
public static int foo(long i, String o) { return 2; }
```

- La chiamata

```
foo(new Integer(7), "ciao")
```

provoca un errore di **ambiguità**, perché il compilatore prima ottiene un insieme di firme candidate vuoto; poi, una volta attivato l'auto-(un)boxing, ottiene candidate **entrambe** le firme "foo", delle quali nessuna è più specifica dell'altra

- Consideriamo i seguenti metodi:

```
public static int foo(int i, Object o) { return 1; }
public static int foo(long i, String o) { return 2; }
```

- La chiamata `foo(new Float(7), "ciao")` provoca un **errore** di compilazione, in quanto il compilatore non trova firme candidate neanche al secondo tentativo
- La chiamata `foo(new Long(7), "ciao")` ottiene output 2, in quanto quella è l'unica firma candidata, una volta attivato l'auto-(un)boxing

- Consideriamo i seguenti metodi:

```
public static int bar(double a, Integer b) { return 3; }
public static int bar(Double a, Integer b) { return 4; }
```

- La chiamata `bar(1.0, 7)` provoca un errore di ambiguità, perché entrambe le firme saranno candidate (al secondo tentativo)
- La chiamata `bar(1, 7)` ottiene il risultato 3, perché avrà un'unica firma candidata (al secondo tentativo)
 - la seconda firma non è candidata perché un *int* non può trasformarsi in *Double* tramite autoboxing

7.6 String e StringBuffer

Le stringhe in Java sono oggetti, la classe String è una classe a tutti gli effetti ma Java la tratta in modo speciale rispetto alle altre classi, ad esempio abbiamo due modi per poter creare delle stringhe:

```
String s = new String("abc"); // primo metodo  
String s = "abc"; // secondo metodo
```

Vedremo poi le differenze tra questi due metodi, se String fosse una classe normale l'unico modo per poter creare un oggetto sarebbe il primo. Nel secondo metodo "abc" viene chiamato **letterale String**, analogamente 1.0 si chiama **letterale double**.

Le stringhe sono oggetti **immutabili**, attenzione ciò che è immutabile è il contenuto di un tipo String non i riferimenti quindi un riferimento può puntare ad una stringa e poi ad un'altra. Vediamo degli esempi:

```
String s = "Walter";  
String s2 = s;  
s = s.concat(" White");  
System.out.println(s);
```

s prima punta ad un oggetto "Walter", poi s2 punta a "Walter" poi s punterà ad una stringa "Walter White", quindi l'output è "Walter White". Se ci fosse una printf di s2 questa stamperà Walter. Se alla riga 3 non ci fosse stato s = ... e avessimo stampato s stamperà solo Walter, perché la funzione concat ritorna la stringa concatenata con un'altra stringa.

```
String s1 = "A";  
String s2 = s1 + "B";  
s1.concat("C");  
s2.concat(s1);  
s1 += "D";  
System.out.println(s1 + s2);
```

Vediamo prima quanti oggetti vengono creati:

1. s1.
2. s2.

3. "B".
4. "C".
5. concat di s1 e C. Anche se non viene assegnato l'oggetto viene comunque creato.
6. concat s2 e s1.
7. s1 + "D".
8. "D".
9. s1 + s2.

Viene poi stampato "ADAB".

```
String s1 = "abc";
String s2 = s1 + "";
String s3 = "abc";
System.out.println(s1 == s2);
System.out.println(s1 == s3);
```

Qui la prima printf è false perché abbiamo due oggetti diversi, mentre la seconda stampa vero perché s1 e s3 puntano allo stesso oggetto (come per gli Integer).

7.6.1 String pool

Quando il compilatore trasforma i letterali in oggetti li conserva in una cache così che se un altro riferimento a String vuole puntare ad una stringa già creata gli viene dato lo stesso indirizzo allocato precedentemente, la differenza con i tipi Wrapper è che la dimensione di questa memoria è potenzialmente illimitata. Questo meccanismo si chiama **String constant pool**, è stata fatta questa scelta di implementazione per motivi di efficienza di memoria. Questo meccanismo di condivisione può funzionare perché gli oggetti String sono immutabili. Se lo stesso letterale compare in punti diversi del codice, l'eventuale modifica di un letterale modificherebbe anche l'altro.

```
String s = new String("abc"); // primo metodo
String s = "abc"; // secondo metodo
```

Da qui possiamo capire che il primo metodo crea una stringa a prescindere se è già stata creata o no, il secondo invece controlla prima se è già stata creata.

7.6.2 StringBuffer e StringBuilder

Esistono anche due classi String che sono mutabili, **StringBuilder** e **StringBuffer**, la differenza tra i due è che **StringBuffer** è **thread-safe**, in pratica se abbiamo diversi thread che lavorano sulla stessa StringBuffer le operazioni che vengono fatte su di esse sono atomiche, lo svantaggio è che le operazioni sono un pò più lente, vediamo un esempio:

```
StringBuffer s = new StringBuffer("Walter");
s.append(" -White");
System.out.println(s);
```

append modifica this quindi s e può farlo perché la classe è mutabile e quindi il risultato è "Walter White". Facciamo attenzione ad una cosa:

```
StringBuffer s = "abc";
```

Questo non si può fare, StringBuffer e StringBuilder non sono auto-convertibili, nemmeno con cast esplicito.

Per quanto riguarda l'**uguaglianza**, la classe String sovrascrive il metodo equals in modo da controllare l'uguaglianza del contenuto dei due oggetti, le classi StringBuffer e StringBuilder non lo sovrascrivono ed usano quello ereditato da Object, che funziona come l'operatore ==, quindi compara i riferimenti.

Queste due classi e anche la classe String sono classi **final**.

7.7 Garbage collector

Consiste nel rilascio automatico della memoria non più accessibile dal programma. Un data object, o più semplicemente un oggetto si dice **eleggibile** per la GC se non è più accessibile dal programma, facciamo un esempio:

```
public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Ciao");
    System.out.println(sb);
    sb = null;
    // ora l'oggetto StringBuffer e' eleggibile per la GC
}
```

Tranne in casi eccezionali, le stringhe nello string pool non sono mai eleggibili per la GC, ovviamente perché altrimenti si perderebbe il senso dello string pool. Quando partirà la GC la memoria degli oggetti eleggibili verrà rilasciata, l'avvio della GC è volutamente incontrollabile ed è trasparente al programmatore, l'unica cosa che possiamo sapere è quando un oggetto è eleggibile.

7.7.1 Algoritmo mark-and-sweep

Esistono diversi algoritmi della GC, questa è la principale famiglia. Questa famiglia della ha due fasi:

- **Mark:** a partire dallo stato attuale dello stack di attivazione e dalla regione statica (in Java consiste nella regione dove vengono allocate tutte le variabili statiche delle classi), visita tutti gli oggetti accessibili e li marca come tali. Possiamo immaginarla come una visita di un grafo dove i nodi sono gli oggetti e gli archi sono i riferimenti agli oggetti.
- **Sweep:** rilascia tutti gli oggetto che non sono stati marcati come accessibili.

La JVM permette di utilizzare diversi algoritmi di GC. Gli altri algoritmi possono differenziarsi per il livello di parallelizzazione, cioè l'algoritmo del GC viene eseguito in parallelo al programma. Vediamo un esempio:

```
public static void main(String[] args) {
    StringBuffer s1 = new StringBuffer("Ciao");
    StringBuffer s2 = new StringBuffer("Addio");
    System.out.println(s1);
    // l'oggetto riferito da s1 non e' ancora
    // eleggibile per GC           I
    s1 = s2;
    // ora e' eleggibile.
}
```

7.8 Eccezioni

Le eccezioni sono un meccanismo offerto da molti linguaggi per gestire situazioni anomale, lo troviamo anche nei linguaggi funzionali, quindi non è dipendente dal tipo di linguaggio. Un linguaggio che non ha questo meccanismo è il C, ma ci sono dei modi per poter gestire situazioni anomale. Prendiamo come esempio la funzione `sqrt` a cui passiamo un valore negativo, per poter gestire l'eccezione possiamo o terminare il programma o ritornare un valore speciale. Il vantaggio di terminare il programma è che abbiamo tutti i valori di ritorno disponibili, però bloccherebbe l'intero programma, mentre i vantaggi e gli svantaggi del `return` sono gli opposti dell'`exit`.

Vediamo come funzionano le eccezioni: supponiamo che il main chiama una funzione `f` che chiama una funzione `g(x)`, viene passato un parametro che però non rispetta una condizione, indipendentemente dal suo valore di ritorno `g` può lanciare un eccezione. In Java `g` chiamerà `throw new IllegalArgumentEx-ception` quindi invece di restituire un valore segnala una situazione anomala. A run time succede che l'esecuzione di `g` viene terminata, il chiamante riceve l'eccezione e ora `f` usa un costrutto specifico per catturarla e quindi l'eccezione si ferma nella `f` e viene eseguito il codice di una delle catch, se `f` non la cattura l'eccezione termina l'esecuzione di `f` e il controllo passa a `main` e si ripete il discorso di prima, se neanche il `main` riesce a catturare l'eccezione il programma termina. Quando viene lanciata un'eccezione, l'oggetto che ne fa riferimento contiene le informazioni dello stack al momento del lancio. Vediamo che l'uso delle eccezioni prende i vantaggi di entrambe le soluzioni che potevamo usare in C, il grosso vantaggio di questo meccanismo è che è il chiamante a decidere cosa fare nel caso di un eccezione.

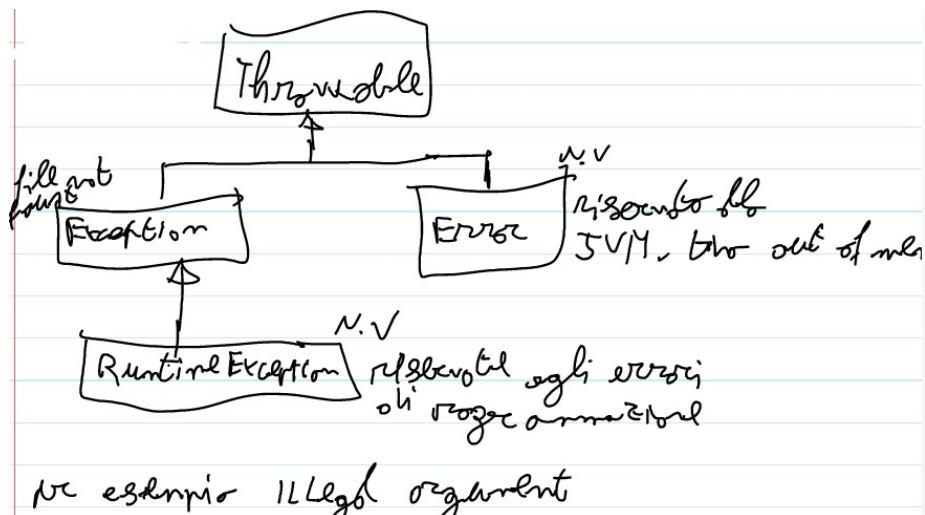
7.8.1 Catturare le eccezioni

Per catturare le eccezioni usiamo il seguente costrutto:

```
try
{
    ...
}
catch(E1 e)
{
    ...
}
catch(E2 e)
{
    ...
}
finally
{
    ...
}
```

Nella parte try si **prova** ad eseguire delle operazioni, se durante la prova viene sollevata un'eccezione di tipo E1 viene eseguito il codice nel catch E1, altrimenti il secondo e infine in ogni caso viene eseguito il codice finally, sia se si sono verificate le eccezioni oppure no, il finally viene eseguito anche se nel catch vi è un return e anche quando l'eccezione non viene catturata e cioè quando viene propagata al chiamante. Se viene sollevata un'eccezione la JVM cerca il primo catch in grado di catturare l'eccezione, quindi l'ordine è rilevante, **in grado di catturare l'eccezione** vuol dire che viene eseguito il codice del primo catch il cui tipo di eccezione è supertipo del tipo effettivo dell'eccezione, ovvero il primo catch di cui l'eccezione è sottotipo.

7.8.2 Gerarchia delle eccezioni



In Java esistono già centinaia di classi di eccezioni, ma quello che bisogna sapere è che un'eccezione deve estendere la classe **Throwable**. Possiamo anche crearne altre noi estendendo una qualsiasi classe e poi per buona prassi scrivere i due costruttori, quello con argomento e quello senza. La nuova classe che creo eredita quindi tutte le caratteristiche del padre, anche se è verificata o no (vedremo poi cosa significa).

7.8.3 Scelta del catch

Ora che sappiamo qual'è la gerarchia delle eccezioni vediamo più nello specifico la scelta del catch. Vediamo un esempio:

```
try
{
    throw new IllegalAE();
}
catch(IAE e)
{
    ...
}
catch(RuntimeExce e)
{
    ...
}
finally
{
    ...
}
```

In questo caso viene eseguito il primo catch, se lanciamo qualcosa di più generale di IAE viene eseguito il secondo catch.

```
try
{
    throw new NullPointerException();
}
catch(IAE e)
{
    ...
}
catch(RuntimeException e)
{
    ...
}
finally
{
    ...
}
```

In questo caso viene eseguito il secondo catch perché NPE non è sottotipo di IAE ma di RE sì.

```

try
{
    throw new IllegalAE ();
}
catch(RuntimeException e)
{
    ...
}
catch(IAE e)
{
    ...
}
finally
{
    ...
}

```

In quest'ultimo caso invece vi è un errore di compilazione, perché RE è stato scritto prima e IAE è sottotipo di RE, l'errore di compilazione avviene quando vi è ridondanza, se l'eccezione E1 è supertipo dell'eccezione E2 vuol dire che tutte le eccezioni che vengono sollevate e che sono sottotipi di E2 sono anche sottotipi di E1 quindi dovrà essere eseguita la catch di E1, per ridondanza intendiamo questo.

7.8.4 Eccezioni verificate e non verificate

Di default le eccezioni sono verificate, quindi le classi **Throwable** sono verificate, poi le classi **Error** e **RunTimeException** non sono verificate. Un **eccezione verificata** vuol dire che se un metodo lancia direttamente o indirettamente un'eccezione verificata il compilatore verifica che l'eccezione venga o catturata oppure sia dichiarata nell'intestazione del metodo con la parola chiave **throws**, vediamo un esempio:

```

void g() throws FileNotFoundException , ...
{
    return ;
}

void f()
{
    g();
}

```

Vediamo che g dichiara che può lanciare un'eccezione però non la lancia, poi f chiama g, FNFE è un'eccezione verificata, quindi scatta il controllo da parte del compilatore ma f non fa niente per l'eccezione che potrebbe essere lanciata e quindi vi è un errore di compilazione. Per non avere un errore di compilazione

dobbiamo mettere in f un try catch oppure nel suo prototipo deve avere **throws** con le classi delle eccezioni verificate che può lanciare g o i suoi supertipi. Questo controllo non viene fatto per le classi di eccezioni non verificate, grazie a dio, perché se ad esempio RE fosse verificata ogni volta che accediamo ad un array bisogna mettere try catch o throws, quindi per questo motivo abbiamo diverse famiglie di eccezioni. Alcune eccezioni sono verificate perché il programma non è sicuro di evitarle perché non dipendono da lui ad esempio nelle situazioni di input output.

7.9 Regole dell'overriding

Supponiamo di avere una classe padre A e una classe figlia B, se in A è presente un metodo con:

- **Visibilità** V.
- **Tipo di ritorno** T.
- **Nome** f.
- **Parametri formali** di tipo U₁, ..., U_n.
- **Dichiara di lanciare eccezioni** E₁, ..., E_n.

In B allora:

- V₁ deve essere visibile almeno quanto V (ovviamente V non deve essere private).
- Il tipo di ritorno T₁ deve essere uguale a T o sottotipo di T, questo vuol dire che se T è primitivo allora T₁ deve essere uguale, per motivi di memory layout, perché i riferimenti sono tutti puntatori mentre i tipi primitivi hanno footprint di memoria diversa.
- Il nome deve essere uguale.
- I parametri formali non possono cambiare.
- Può lanciare le stesse eccezioni oppure le nuove eccezioni dichiarate devono essere tutte sottotipo di quelle vecchie, per essere precisi questo si applica solo alle eccezioni nuove e verificate. Il numero di quelle nuove può essere anche minore o maggiore di quelle vecchie ma l'importante è che ciascuna di esse che sia verificata deve essere sottotipo di quelle vecchie. L'ordine non ha importanza. In matematiche:

$$\forall i = 1 \dots k \text{ se } F_i \text{ è checked allora } \exists j = 1 \dots m : F_i \text{ è sottotipo di } E_j$$

Facciamo qualche esempio:

```
in A:  
f() throws FNFE, IAE
```

Vediamo cosa possiamo scrivere in B che estende A:

- **f()**, va bene perché se vedi nell'ultima condizione non vi è alcun vincolo in cui k deve essere maggiore di 0, di conseguenza il per ogni rende vera la condizione.
- **f() throws FNFE**, si va bene perché è presente anche nel super.
- **f() throws NPE**, è **vera** perché NPE fa parte dei RE e FNFE fa parte delle EX.

- **f() throws Exception, non va bene** è super di FNFE e IAE.
- **f() throws F1, F2, NPE**, considerando che F1 e F2 sono due classi che ereditano da FNFE, allora **va bene**.

7.10 Classi interne

Una classe oltre ad avere attributi, metodi e costruttori può avere anche altre classi annidate:

```
class A{
    ...
    class B{
        ...
    }
}
```

A viene detta classe **top-level**, la classe interna può avere una sua visibilità e può essere statica o no.

Vediamo le differenze tra A e B:

1. **Restrizioni di visibilità:** B rispetto ad A può avere come visibilità anche protected o private, mentre A può avere solo default o public. Dire che B è private significa che è visibile solo esclusivamente da A, proprio come per gli attributi. Rendere B private serve per creare un maggiore encapsulamento, decidiamo di creare una classe interna private quando vogliamo specificare un dettaglio implementativo della classe A, quindi esiste solo per supportare A. Ad esempio in una classe lista concatenata, la classe nodo è una classe interna di lista ed è visibile solo da lista. Esiste un principio che fa nascere l'idea di dare meno visibilità possibile e cioè il principio del **minimo privilegio possibile**, un'altra applicazione di questo principio è quello di creare le variabili in modo **final** se è possibile, perché la final ha meno privilegi. Questo principio nasce perché mettendo più vincoli possibili irrobustisco il codice.

2. **Riferimento隐式 (alla classe contenitore):** Il modo migliore per poterlo capire è vedere il Memory Layout:

```
public class S{
    private int n = 2;

    public class T{
        private int m = 7;
        public void f(){
            System.out.println(S.this);
        }
    }
}
```

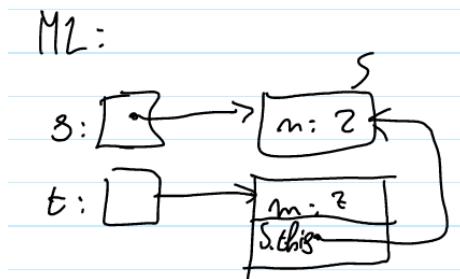
}

Fuori da S:

```
S s = new S();  
S.T t = s.new T();
```

Come si **istanzia un oggetto di T**: a causa della proprietà numero 2 ogni oggetto di tipo T deve essere legato ad un oggetto di tipo S.

Il Memory Layout in questo caso è:



Il nome di questo riferimento è S.this. Questo significa che quando viene chiamato il metodo f() la println fa il toString di s, in particolare della s che ha creato t. In pratica l'interno conosce chi li ha creati ma l'esterno non conosce i suoi interni. Questo riferimento quindi viene usato quando la classe interna ha bisogno di sapere chi è il suo contenitore.

3. **Privilegi di visibilità:** in pratica A e B non hanno segreti, vediamo un esempio:

```
public class A{  
    private int n = 7;  
  
    public void f(B b){  
        System.out.println(b.m); //OK  
        System.out.println(m); //NO  
        B.h(); //OK  
    }  
  
    public class B{  
        private int m = 42;  
  
        private void g(){  
            A.this.n = 0; //OK  
            n = 0; //OK  
        }  
    }  
}
```

```

    private static void h() { ... }
}
}

```

Ovviamente l'unico NO che si vede è perché m non è nello scope. Quindi tutto ciò che è privato in B è visibile ad A e viceversa. Questa proprietà serve per stabilire una collaborazione tra B ed A. Tutte le restrizioni di visibilità iniziano ad avere effetto fuori da A. Questa proprietà vale anche per classi interne sorelle o annidate ancora di più.

7.10.1 Classi interne statiche

Le classi TopLevel non possono essere statiche, quando una classe interna è static gode solo delle proprietà 1 e 3, quindi B non ha più il riferimento al suo contenitore. Prendendo gli esempi di prima, fuori da A possiamo fare:

```

A a = new A();
A.B b = new A.B();

```

Mentre invece il seguente codice non funziona

```

//B e' classe interna di A:
public class B{
    private int m = 42;

    private void g(){
        A.this.n = 0; //NO
        n = 0; //NO
    }

    private static void h() { ... }
}

```

Questo perché B non ha il riferimento ad A. L'unico modo per poter eseguire questo codice è passare a g un A. Quando è possibile creare una classe interna statica è meglio farlo, sempre per il principio del minimo privilegio, non solo perché ha meno privilegi ma anche perché occupa un campo di memoria in meno.

Rispetto ad altri linguaggi le classi interne hanno le seguenti proprietà:

- **C++:** gode della proprietà 1 e 3 solo in una direzione, cioè da A a B. La classe interna statica non esiste perché di già non godono della proprietà 2. C++ in aggiunta alle classi interne esiste il costrutto **friend** che permette a delle classi TopLevel di non avere segreti, quindi la visibilità non conta. Se ho due classi A e B che devono essere amiche devo specificare che A è friend di B e B è friend di A.
- **C#:** Come in C++ ma non ha il costrutto **friend**. C# sconsiglia di creare classi interne pubbliche.

7.11 Sguardo generale agli altri paradigmi

Abbiamo quasi finito con il paradigma ad oggetti, ora diamo uno sguardo all'evoluzione dei diversi paradigmi e linguaggi.

7.11.1 Sistemi di tipi

All'inizio esistevano solo i tipi elementari e nessuna gerarchia di classi. Poi sono stati introdotti i tipi user-defined, ancora nessuna classe e inizialmente erano delle semplici ridenominazioni di tipi elementari oppure nomi di record. Il **type equivalence** stabilisce quando assegnamento $x = y$ è valido.

7.11.2 type equivalence

In questi linguaggi non ancora ad oggetti esistevano due **forme di equivalenza**:

- **Name equivalence:** i tipi di x e y devono avere lo stesso nome, cioè lo stesso tipo.
- **Structural equivalence:** i tipi di x e y sono equivalenti se hanno lo stesso contenuto. Ad esempio:

```
Euro x;  
Dollar y;  
z = x + y;
```

Si può fare se vi è equivalenza strutturale, altrimenti con l'equivalenza di nome è errore.

In Pascal abbiamo sempre name equivalence. in C e in C++ è sempre structural tranne che per le struct. In C e in C++ è stata fatta questa scelta perché è più difficile stabilire equivalenza strutturale nelle struct (immaginiamo struct con 10 miliardi di campi, o struct dotati di puntatori ad altre struct, il compilatore si scoccia e c'ha pure ragione).

7.11.3 Compatibilità di tipi

Con l'avvento dei linguaggi a oggetti l'equivalenza viene rimpiazzata da compatibilità, qualunque sottotipo di T è compatibile con T . Nei linguaggi OO più comuni la compatibilità è basata su nome piuttosto che struttura, due classi con nomi diversi sono diverse, anche se hanno gli stessi attributi e gli stessi metodi.

7.11.4 Evoluzione dei sistemi di tipi

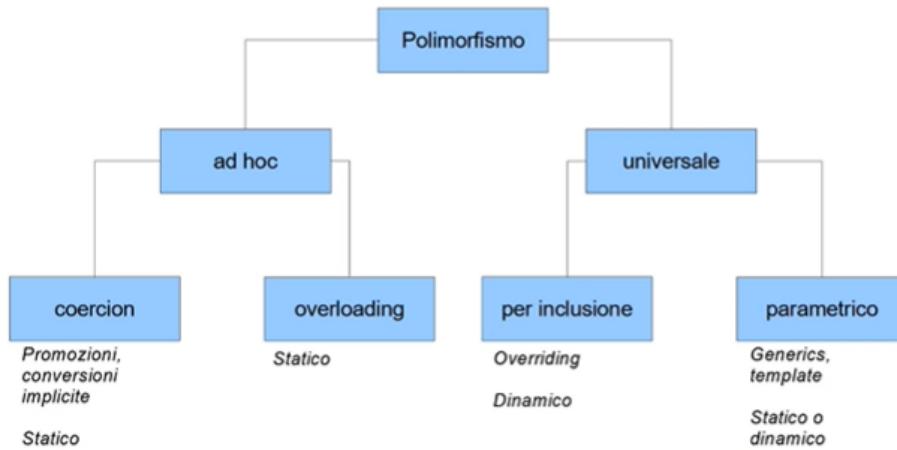
1. Tipi elementari.
2. Tipi user-defined, solo strutture dati.
3. Interfacce e tipi di dato astratti.
4. Gerarchia di tipi.

7.11.5 Caratteristiche utili alla classificazione dei linguaggi

- Paradigma di riferimento
 - imperativo, O.O, funzionale, logico
- Scoping (statico/dinamico)
- Gestione della memoria
 - Allocazione dinamica, garbage collection/esplicita, ...
- Sistema di tipi
 - strong/weak, encapsulation, equivalence/compatibility, polimorfismo (di 4 tipi), *type inference*...
- Supporto alle eccezioni
 - *Eventualmente integrato con type checking... vedi ML* :
- Supporto al parallelismo
 - Memoria condivisa (*synchronized*), scambio di messaggi (nel senso *RMI*), gestione del nondeterminismo, fairness

7.12 Polimorfismo

Definizione: uno stesso oggetto sintattico (funzione, metodo, variabile, ecc.) appartiene a più tipi, significato o comportamento diverso a seconda del contesto. Ad esempio anche l'operatore + è polimorfo, $1 + 2$ ha un tipo e un significato diverso da "a" + "b". Possiamo parlare di polimorfismo in molti casi diversi:



- **Ad hoc:** si chiama ad hoc perché è specifico per particolari tipi di dato. Si aggiungono singoli casi di polimorfismo ogni volta. Ad esempio le conversioni implicite, o l'overloading, rispetto al caso della somma abbiamo: somma tra interi, somma tra double e somma tra stringhe.
 - **Coercizioni di tipo:** sono le promozioni automatiche, per esempio la somma tra double e int, viene convertito int in double e la somma si riduce a somma di double
- **Universale:** si applica ad un numero di casi illimitato a priori. Ad esempio aggiungendo una nuova superclasse con un metodo concreto, l'overriding crea casi di polimorfismo rispetto a tutte le sottoclassi che già definiscono quel metodo.
 - **Parametrico:** riguarda i tipi parametrici o templates, lo spieghiamo dopo.

7.12.1 Polimorfismo parametrico

Facciamo direttamente un esercizio. Dobbiamo definire il tipo di dato Stack con operazioni:

- Push(element).
- Pop().
- Non forzare un tipo specifico per gli elementi.
- Implementazione a vettore.

Una prima implementazione che non usa i **Generics** è la seguente:

■ Sfruttare Object

```
public class ObjectStack {  
  
    private Object[] stack;  
    private int top = -1;  
  
    public ObjectStack( int dim ) { stack = new Object [dim]; }  
  
    public void push( Object el ) { stack[++top] = el; }  
  
    public Object pop() { return stack[top--]; }  
}
```

Una soluzione migliore è quella di usare una **classe parametrica**:

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack( int dim ) { stack = new ElemType[ dim ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Parametro formale di tipo

Ci piacerebbe...
ma non si può fare!

Vantaggi dei due approcci:

- **Pro del polimorfismo parametrico:**

- Non richiede controlli di tipo a run time.
- Anticipa scoperta errori di tipo a tempo di compilazione.

- **Pro del polimorfismo per inclusione:**

- Permette di sfruttare strutture dati eterogenee, ad esempio stack con tipi diversi. Con un generics possiamo farlo, basta passare Object come tipo parametrico.

In molti casi servono collezioni di elementi eterogenei ma vogliamo usarli allo stesso modo, ad esempio una lista di forme eterogenee (rettangoli, ellissi, linee ecc) vogliamo usarla per implementare refresh, che deve solo inviare un messaggio draw() a tutte le forme della lista, forme diverse avranno implementazioni diverse del metodo draw, però posso usare il polimorfismo per inclusione per creare un supertipo comune di tutte queste forme diverse e uso questo supertipo come argomento del generics, ad esempio un arraylist. Quindi all'interno di questo arraylist posso inserire tutti i tipi di forme.

Abbiamo visto nell'ultima immagine che una riga non viene compilata. Questo perché in Java dopo la compilazione la classe che abbiamo creato rimane comunque uno stack di Object. I parametri di tipo vengono usati per il type checking e poi cancellati, per questo un parametro attuale di tipo non può essere primitivo ma dobbiamo usare i wrapper e per questo non si può istanziare un array di un tipo parametrico. Quindi quel problema si risolve così:

```
public class GenericStack<ElemType> {
    private ElemType[] stack;
    private int top = -1;

    public GenericStack( int dim ) { stack = new ElemType[ dim ]; }

    public void push( ElemType el ) { stack[++top] = el; }

    public ElemType pop() { return stack[top--]; }
}
```

The code shows a generic stack class named `GenericStack` with a type parameter `<ElemType>`. It contains three methods: `push` which adds an element to the top of the stack, and `pop` which removes the top element. A `private` variable `stack` is used to store the elements. Annotations are present: a yellow box labeled "Parametro formale di tipo" points to the type parameter `<ElemType>`; another yellow box labeled "Ci piacerebbe... ma non si può fare!" points to the line `return stack[top--];`.

▪ Per riuscire a compilare:

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    @SuppressWarnings("unchecked")  
    public GenericStack( int dim ) { stack = (ElemType[]) new Object[ dim ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Annotazione

- Ora niente warning
- Meglio ancora: usare ArrayList<ElemType>

Se stack fosse un arraylist non avrei nessuno di questi problemi.

Note sull'implementazione in C++

▪ In C++, ogni uso di un template fa compilare una nuova classe

- Sorta di macro
- Il compilatore inserisce nel programma la definizione di classe specializzata e la ricompila per ogni parametro attuale
- Più espressivo ma anche più “costoso”
- Si può usare un parametro di tipo in tutti i modi in cui si potrebbe usare un tipo vero

7.12.2 Astrarre un'API tramite interfaccia

Vedere l'esercizio dalle slide Java-generics a pagina 15.

Capitolo 8

Paradigma funzionale

Programmare in stile funzionale puro significa usare solo **espressioni e funzioni**, eventualmente ricorsive, non c'è una memoria che cambia e quindi di conseguenza non vi sono assegnamenti e non ci sono iterazioni. Gli environment mappano gli identificatori direttamente sul loro valore (immutabile) invece di una locazione di memoria (il cui contenuto può cambiare) perché tutti gli identificatori indicano delle costanti, cioè è come se nei linguaggi funzionale esistesse solo l'inizializzazione e quel valore poi non può essere modificato. Le conseguenze sulla programmazione sono:

- Ricorsione al posto dei cicli.
- Modifiche all'ambiente anziché alla memoria.
- Creazione identificatori con stesso nome mediante chiamate ricorsive.
- Creazione di nuovi identificatori con lo stesso nome che mascherano la versione precedente (come nello scoping statico).

Il linguaggio che vedremo in particolare è ML (Meta Language, nella versione Standard ML of New Jersey <https://www.smlnj.org/>). Diversi linguaggi OO stanno acquistando caratteristiche funzionali come le lambda-espressioni. ML supporta le seguenti implementazioni:

- Interprete interattivo.
- Implementazione mista.
- Compilazione.

I linguaggi funzionali sono ampiamente utilizzati in ambiente finanziario. Scegliamo questo paradigma per diversi motivi: per parallelismo (per via dell'immutabilità), perché teniamo particolarmente alla robustezza e perché vogliamo evitare gli errori a run time, in un linguaggio funzionale è molto improbabile che avvengano errori a run time per via del sistema dei tipi molto robusto.

8.1 Il sistema di tipi

ML è fortemente e staticamente tipato, il controllo dei tipi avviene interamente a tempo di compilazione ma non richiede di dichiarare il tipo degli identificatori, spesso lo capisce da solo (**type inference**) . Usa sia structural che name equivalence, permette di definire tipi ricorsivi e supporta polimorfismo parametrico, ha un garbage collector e supporta encapsulation (tipi di dato astratti) ma non è un linguaggio a oggetti perché mancano la gerarchia di tipi. Il linguaggio OCaml supporta anche gerarchia di tipi.

8.1.1 Tipi primitivi

Detti anche base types:

- **int**: 0, 1, 2, \sim 1, 0xff. Si usa tilde per indicare il meno e alcuni operatori sono: +, -, mod, div, =.
- **word**: interi senza segno. 0w10, 0w2, 0wxff.
- **real**: 3.14, 0.1E6.
- **string**: "abc". Alcuni operatori sono ; size, =.
- **char**: #'a". Alcuni operatori sono ord, chr, =.

it si riferisce all'espressione data, calcola si il valore che il tipo. Non vi è nessuna conversione automatica tra tipi numerici. Real non supporta l'uguaglianza, per via dell'arrondimento tra reali, per confrontare l'uguaglianza tra due reali bisogna usare Real.==. Esiste il valore **NaN (not a number)** e non può essere confrontato con nessun altro numero.

8.2 Dichiarazioni e scoping

8.2.1 Funzioni

Ci sono diversi modi di definire e chiamare funzioni in ML:

```
fun quadr x = x * x;
val quadr = fn : -> int

quadr(3);
val it = 9 : int
```

Con **fun** si dichiara la funzione, fun aggiunge all'ambiente l'identificatore quadr e lo associa alla funzione da interi a interi. Si può vedere cosa è associato a quadr senza chiamare la funzione:

```
quadr;
val it = fn : int -> int
```

E' possibile indicare esplicitamente i tipi:

```
fun quadr (x : int) : int = x * x;
```

Vediamo ora una funzione ricorsiva:

```
fun fatt x = if x = 0 then 1 else x * fatt (x - 1);
```

In questo linguaggio il costrutto if è un'espressione.

8.2.2 Dichiarazione di identificatori

Con **val** si aggiunge un nuovo identificatore all'ambiente e gli si associa un valore:

```
val x = 2 + 2;  
val x = 4 : int  
  
x + 2;  
val it = 6 : int
```

Vediamo adesso la grammatica delle dichiarazioni viste sinora:

```
<declaration> ::=  
  val <id name> = <expression> |  
  fun <func name> <argument>* = <expression>
```

8.2.3 Scoping

L'equivalente dei blocchi in ML è:

```
let
  <dichiarazioni>
in
  <espressione>
end
```

E' necessario utilizzarle quando le espressioni (che rappresentano il corpo di una funzione) diventano più corpose. Lo scoping è statico, ad esempio:

```
- let val x=2 in 3*x end;
val it = 6 : int

-x;
Error: unbound variable or constructor: x

- let val x=2 in
  let val x=3 in (* questa def. maschera la precedente *)
    3*x
  end
end;
val it = 9 : int
```

Vediamo ora l'ambiente non locale delle funzioni:

```
- val x=0;
val x=0 : int

- let val x=1 in
  let fun f(y) = x+y in      (* x è non locale *)
    f(0)
  end
end;
val it = 1 : int
```

Ora vediamo il costrutto **local**:

```
local
  <dichiarazioni>
in
  <dichiarazioni>
  (le dichiarazioni sopra valgono solo qui)
end
```

E' simile a let ma dopo in c'è una dichiarazione invece di una espressione da valutare.

Per esempio:

```
local val x = 1 n val y = x + 1 end;
val y = 2 : int
let val x = 1 in x + 2 end;
val it = 3 : int
```

Qui ho dovuto usare local perché ho messo una dichiarazione che non è una espressione, una dichiarazione è ad esempio val o fun e aggiungono un nuovo binding nome valore all'ambiente, dato che let dentro in vuole un'espressione non potevo usare let. Quindi in sintesi se voglio una dichiarazione che è locale ad un'altra dichiarazione si usa local, se invece voglio una dichiarazione che è locale ad un'espressione si usa let.

8.3 Costrutti

Questa è una piccola nota sui costrutti offerti da ML. Abbiamo due costrutti:

- **Espressioni:** sono frammenti sintattici dotati di valori, cioè una volta eseguite ritornano un valore.
- **Dichiarazioni:** è un costrutto che aggiunge un nuovo binding all'ambiente, al suo interno sicuramente contiene un'espressione, ad esempio abbiamo **val** che crea un binding nome e valore, dove il valore è un'espressione. **Fun** invece assegna un nome ad una funzione l'unica differenza con **val** è che può accettare parametri

8.4 Tipi strutturati

Si possono definire n-uple semplicemente mettendo i valori tra parentesi:

```
val x = (1 + 1, "A");
val x = (2, "A") : int * string
```

Il prodotto cartesiano viene indicato con *. Si estrae l'i-esimo elemento da una n-upla con l'operatore prefisso #i:

```
#1(x);
val it = 2 : int
```

E' importante non chiamarlo array perché questo è un prodotto cartesiano di tipi, un array è una struttura dati omogenea mentre con la n-upla possiamo avere tipi eterogenei.

8.4.1 Record

E' un insieme di espressioni del tipo nome = valore.

```

val r = {nome = "Mario", nato = 1998};
val r = {nato = 1998, nome = "Mario"} : {nato:int, nome:string}

```

Non c'è bisogno di creare il tipo struct ma è il compilatore che lo crea tramite type inference, in questo caso crea una struct con un campo di nome nato di tipo int e un campo di nome nome (e che cazzo) di tipo string. Il valore associato al nome N si estrae con #N:

```

#nome(r);
val it = "Mario" : string

```

L'ordine delle coppie non conta.

8.4.2 Dichiarazioni di tipo

ML permette di definire nuovi tipi, come il typedef:

```

type coord = real * real;
type coord = real * real

```

Se io dichiaro una variabile x di tipo coord il compilatore mi dirà che è di tipo real * real, per far sapere al compilatore che x è proprio di tipo coord dobbiamo scrivere:

```

val x:coord = (3.0, 4.0);
val x = (3.0, 4.0) : coord

```

I tipi coord e real * real sono compatibili tra loro (structural equivalence).

8.4.3 Datatypes e costruttori

Mentre **type** è come typedef quindi definisce un alias per un tipo già esistente, **datatype** crea un nuovo tipo su una base di elenchi di casi, questo elenco di casi assomiglia ad una enum:

```

datatype color = red | green | blue;
datatype color = blue | green | red

```

Questi tre casi che vediamo si chiamano **costruttori**, definiscono i possibili valori del tipo color e permettono di creare dei **data objects**.

```

val c = red;
val c = red : color

```

Notiamo la somiglianza con le enum del C, ma solo apparente. In ML qui è tutto estremamente controllato mentre in C è più flessibile:

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };
printf("%d%d%d",red,green,blue);      (* stampa 012 *)

enum color c = red;
if( c == 0 ) then ... else ...;      (* esegue il then *)

c = 10;                                (* nessun errore!!! *)
```

- Invece i datatypes di ML definiscono tipi genuinamente nuovi: nessuna corrispondenza con gli int

```
- val c:color = 10;
Error: pattern and expression in val dec don't agree

- c = 0;                                (* c è uguale a 0? *)
Error: operator and operand don't agree
```

red, green, blue sono oggetti completamente nuovi

- Ogni tipo definito con datatype è incompatibile con tutti gli altri tipi (*name equivalence*)

8.4.4 Costruttori con argomenti

I valori enumerati possono avere degli argomenti, facciamo direttamente un esempio: dobbiamo definire una lista concatenata di interi, per farlo creiamo un datatype ricorsivo che prevede due casi: lista vuota o un nodo che contiene un intero e una lista di interi. Quindi servono 2 costruttori uno per la lista vuota e uno per i nodi:

```
datatype listaInt = vuota | nodo of int * listaInt;
datatype listaInt = nodo of int * listaInt | vuota
```

Vediamo che abbiamo utilizzato la parola chiave of dopo nodo, in pratica abbiamo il costruttore nodo che accetta dei parametri.

```
val L = nodo(1, nodo(2, vuota));
val L = nodo(1, nodo(2, vuota)) : listaInt
```

Nella slide 31 vi è un altro esempio con un albero binario.

8.5 Patterns e matching

Per scorrere una lista o più in generale per manipolare datatype (ricorsivi o meno) è fondamentale il **pattern matching**. Consideriamo una lista, se non è vuota potrebbe servirci il primo elemento, si estrae con **pattern matching**, vediamo un esempio considerando la lista di prima:

```
val nodo(p, _) = L;  
val p = 1 : int
```

L'istruzione assegna a p il primo elemento di L, nodo(p, _) si chiama **pattern**, il _ si chiama wildcard e vuol dire "qualsiasi cosa". In pratica significa dire all'interprete: vedi se L corrisponde al seguente pattern nodo, p e qualsiasi altra cosa. Per ottenere il resto della lista invece:

```
val nodo(_, r) = L;  
val r = nodo (2, vuota) : listaInt
```

Vediamo un altro esempio:

```
- fun conta x =  
  if x = vuota then 0  
  else  
    let val nodo(_, r) = x in  
      conta(r) + 1  
    end;  
val conta = fn : listaInt -> int  
  
- conta L;  
val it = 2 : int
```

- Notare come il compilatore ha *inferito* il tipo della funzione
 - 1. x viene confrontato con "vuota", che è di tipo listaInt \Rightarrow anche x è di tipo listaInt \Rightarrow l'input di "conta" è un listaInt
 - 2. il "then" restituisce 0, che è un intero; quindi l'output di "conta" è un intero

- Inoltre il compilatore controlla che anche il resto della funzione sia compatibile con questi tipi
 - 1. r corrisponde al 2° argomento del nodo, che è di tipo listaInt \Rightarrow è corretto passarlo a conta che restituisce un intero
 - 2. quindi anche l'else restituisce un intero \Rightarrow tutto torna

8.5.1 Abbreviazioni per i pattern

Qua arriva la parte forte. Si possono definire delle funzioni per casi, vediamo direttamente l'esempio della funzione conta:

```
fun conta(vuota) = 0 |
    conta(nodo(_, r)) = conta(r) + 1;
```

Possiamo mettere dei pattern al posto dei parametri formali.

8.5.2 Espressioni per casi

Abbiamo visto pattern matching in due contesti sintattici diversi val e fun, ora vediamo il terzo e cioè un'espressione fatta da un pattern matching. E' un'espressione definita per casi, vediamo un esempio: vogliamo dare un valore intero al datatype color.

```
fun ordinal c = case c of verde -> 0
                     | giallo(_) -> 1
                     | rosso(_) -> 2;
```

8.6 Liste

Esiste la struttura dati lista built in nel linguaggio, in qualche misura sostituisce il vettore.

```
nil (* lista vuota *)
1 :: 2 :: 3 :: nil
```

nil è il costruttore di lista vuota, :: è il costruttore di lista non vuota in questo caso crea la lista 1, 2, 3. Per nil abbiamo anche un sinonimo che sfrutta le parentesi quadre:

```
[]  
[1, 2, 3]
```

Il costruttore :: è un operatore asimmetrico che vuole a sinistra un valore e a destra una lista, infatti vediamo che:

```
1 :: 2 :: 3 :: nil;
equivale a
1 :: (2 :: (3 :: nil));
```

Le funzioni fornite dal linguaggio sono:

- **length.**
- **null**, restituisce true se la lista è vuota.
- **hd (head)** e **tl (tail)** restituiscono il primo e il resto della lista rispettivamente.

- **Concatenazione @**

- **List.nth(L, i)** è una funzione della libreria e restituisce l'iesimo elemento della lista.
- **List.last(L)** restituisce l'ultimo elemento di L, anche questa appartiene alle funzioni della libreria.

8.7 Curryng

In ML ogni funzione ha un solo argomento:

```
fun f(x, y) = ...
fun f' x y = ...
```

La prima è una coppia, la seconda invece è una funzione che accetta un parametro x e che restituisce una funzione che prende y e calcola l'espressione dopo =, infatti il tipo di f' è int -> int -> int che va inteso come int -> (int -> int). La trasformazione da n-uple (f (x, y)) a funzioni che restituiscono funzioni (f' x y) si chiama **currying** (dal nome Haskell Curry era talmente op che è stato dato il suo nome ad un linguaggio funzionale, il suo secondo nome ad un'altra cosa e il suo cognome al currying, GOD). L'utilizzo è ovviamente diverso:

```
- fun f (x,y) = x+y;
val f = fn : int * int -> int

- fun f' x y = x+y;
val f' = fn : int -> int -> int

- f (3,2);
val it = 5 : int

- f' 3 2;           (* viene interpretato come (f'3)(2) *)
val it = 5 : int

- f 3 2;
Error: operator and operand don't agree

- f' (3,2);
Error: operator and operand don't agree

- val g = f' 3;
val g = fn : int -> int

- g 1;           I
val it = 4 : int
```

8.8 Funzioni di ordine superiore

Sono delle funzioni che accettano altre funzioni, le tipologie di funzioni più comuni sono tre:

- **filter.**
- **map.**
- **reduce.**

Vediamo un esempio, vogliamo scrivere una funzione f che esegue due volte una funzione g su un valore x:

```
fun f g x = g(g(x));
val f = fn : ('a -> 'a) -> 'a -> 'a
```

8.8.1 Filter

la funzione filter prende una funzione booleana f e una lista L e seleziona gli elementi di L per cui f è vera:

```
fun filter f [] = []
| filter f (x :: y) = if f(x) then x :: (filter f y) else fi
```

Vediamo un esempio dove dobbiamo selezionare gli elementi negativi da una lista

```
let fun neg x = x < 0
in filter neg [L] end;
```

8.8.2 Map

La funzione map prende una funzione di qualsiasi tipo f e una lista L e applica f a tutti gli elementi della lista:

```
fun map f [] =
| map f (x :: y) = f(x) :: (map f y)
```

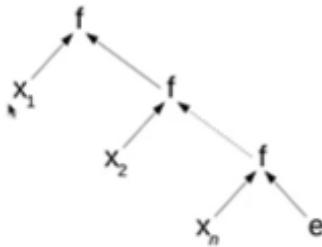
Vediamo un esempio dove convertiamo una lista di interi in reali:

```
map real [1, 2, 3];
val it = [1.0, 2.0, 3.0]
```

8.8.3 Reduce

Serve per calcolare aggregati di una lista: min, max, prodotto, media. Abbiamo visto che filter e map sono due pattern che guardano un elemento alla volta, quindi non possono combinare gli elementi. Prende in input una funzione a 2 argomenti f, un valore finale e una lista L ed effettua questo calcolo:

$$\text{reduce } f \text{ e } [x_1, \dots, x_n] = f(x_1, f(x_2, \dots, f(x_n, e)))$$



Ad esempio se f fosse $+$ ed e = 0 allora reduce calcolerebbe la somma degli elementi della lista. Vediamo la definizione e degli esempi:

```

fun reduce f e [] = e
| reduce f e (x :: y) = f (x, reduce f e y);
Il seguente esempio fa la somma
reduce (op +) 0 [1, 2, 3];
val it = 6 : int
  
```

op + vuol dire che stiamo passando l'operazione somma, avremmo anche potuto creare una nostra funzione che fa la somma, l'importante è che la funzione che passiamo abbiamo una coppia di argomenti e non sfrutti il currying.

In ML abbiamo due funzioni che implementano reduce: **List.reduce** e **List.foldr**. **List.reduce f e l** restituisce:

- e se la lista l è vuota.
- $f(x_1, f(x_2, \dots, f(x_{n-1}, x_n)))$ se la lista non è vuota.
- Se la lista ha un solo elemento a viene ritornato l'elemento a.

Il tipo di **List.reduce** è:

```
fn : ('a * 'a -> 'a) -> 'a -> 'a list -> 'a
```

List.foldr f e l restituisce:

- $f(x_1, f(x_2, \dots, f(x_n, e)))$.

Quindi è come la reduce canonica ma è più generale perché è in grado di accettare una funzione f che accetta due tipi diversi:

```
f : 'a * 'b -> 'b
```

Il tipo di di List.foldr è:

```
fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Cioè foldr ci consente di aggregare i valori presenti nella lista con un aggregato di tipo diverso, mentre reduce permette di aggregare i valori della lista in unico valore dello stesso tipo della lista, vediamo un esempio:

```
- fun f (n:int, s:string) = Int.toString(n) ^ ";" ^ s;
val f = fn : int * string -> string

- f (323, "aaa");
val it = "323;aaa" : string

- List.foldr f "fine" [23,3434,44,0,12];
val it = "23;3434;44;0;12;fine" : string
```

8.9 Funzioni anonime

Esiste una sintassi che permette di definire delle funzioni senza dover assegnare un nome e è molto utile per le funzioni di ordine superiore. La keyword che usiamo è **fn**:

```
fn x => x + 1;
val it = fn : int -> int

map (fn x => x + 1) [1, 2, 3];
val it = [2, 3, 4] : int list
```

L'idea di avere delle funzioni anonime sta prendendo piede anche nei linguaggi imperativi con le **lambda espressioni**. Le funzioni anonime non possono accettare più parametri, bisogna fare manualmente il currying.

8.9.1 Val vs fun

Avendo visto le funzioni anonime adesso possiamo capire perché val è più generale di fun. Le seguenti definizioni sono equivalenti:

```
fun f x = x + 1;
val f = fn : int -> int

val f = fn x => x + 1;
val f = fn : int -> int
```

Fun è **zucchero sintattico**, cioè una utile abbreviazione per qualcosa che si potrebbe fare in altro modo.

8.9.2 Ulteriori dettagli su currying

Adesso possiamo mostrare più esplicitamente la natura del currying, riprendiamo l'esempio:

```
fun f' x y = x + y;
```

f' può essere definita equivalentemente come:

```
fun f' x = fn y => x + y;  
val f' = fn : int -> int -> int
```

A questo punto scrivere:

```
val g = f' 3;  
g 1;  
val it = 4 : int
```

Equivale a:

```
g = fn y => 3 + y;
```

E' possibile scrivere delle funzioni che effettuano il currying e il de-currying di una funzione:

```
val curry_2args = fn f => fn x => fn y => f (x, y);  
val uncurry_2args = fn f => fn (x, y) => f x y;
```

Vediamo degli esempi di utilizzo:

```
fun f(x, y) = x + y;  
val f' = curry_2args f;  
(* equivalente a f' x y = x + y*)  
fun f' x y = x + y;  
val f = uncurry_2args f';  
(* equivalente a f(x, y) = x + y*)
```

8.10 Polimorfismo parametrico

ML supporta l'analogo dei template di C++ e Java, ovvero i **tipi parametrici**, in realtà li stiamo usando da quando usiamo le liste. Il costruttore :: può essere applicato a qualunque tipo, interi stringhe ecc. Per definire i tipi parametrici si usa 'a:

```
(* definizione albero binario *)
datatype 'a bt = emptybt | btnode of ('a * 'a bt * 'a bt);
datatype 'a bt = btnode of 'a * 'a bt * 'a bt | emptybt
```

8.10.1 Tipi parametrici

ML usa l'apice prima del nome per indicare che quella è una variabile di tipo, ad esempio 'a. Se vogliamo che il tipo supporti l'uguaglianza allora si mette un doppio apice "a. Ogni tanto la type inference se ne accorge da sola.

8.11 Encapsulation e interfacce

Le **signature** sono il costrutto ML per definire interfacce, definiscono tipi e funzioni senza specificare come sono implementati. Le abbiamo usate ad esempio per le funzioni List. Vediamo un esempio:

```
signature STACK =
sig
    type 'a stack
    val empty : 'a stack
    val push: ('a * 'a stack) -> 'a stack
    val pop: 'a stack -> ('a * 'a stack)
end;
```

Ci serve ora una struttura che implementa questa segnatura e questa struttura si chiama struttura (XD). Per implementare una interfaccia usiamo le **structure** che come le classi definiscono tipi di dato astratti, per esempio:

```
structure Stack :> STACK =
struct
    type 'a stack = 'a list;
    val empty = [];
    fun push (x, s) = x :: s;
    fun pop (x :: s) = (x, s);
end;
```

:> significa **implements**. Ovviamente non è una classe perché una segnatura è semplicemente una collezione di firme e tipi, una struttura invece è una collezione di tipi, valori e funzioni concreti che rispettano i tipi della segnatura. Vale l'**incapsulamento** cioè come viene implementato il tipo di dato astratto è visibile solo dalla structure e non all'esterno, ogni altra funzione definita nella

structure non è accessibile da fuori, e per questo motivo lenght Stack.empty da errore perché lenght non sa che stack è una lista.

8.11.1 Functors, strutture parametriche

Alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri, per definire una struttura parametrica si usa la keyword: **functor**. Vediamo un esempio, definiamo una struttura che rappresenta un'immagine che ha come parametro un tipo che permette di conoscere quale rappresentazione di colori è stata usata cioè RGB e CMYK, supponiamo che esista una signature COLOR che rappresenta in astratto queste due codifiche e quindi RGB e CMYK sono strutture che implementano COLOR, noi dobbiamo dichiarare una struttura che ha un parametro di tipo COLOR, cioè:

```
functor Image(X : COLOR) =  
  struct  
    (* qui si puo' usare X come un tipo *)  
    (* con le operazioni definite da COLOR *)  
  end;  
  (* col functor si possono generare diversi tipi di dato *)  
  structure Image_RGB = Image(RGB);  
  structure Image_CMYK = Image(CMYK);
```

8.12 Eccezioni

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
  x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. la type inference capisce che l'input di pop è una lista
2. il datatype lista ha due costruttori: `::` e `[]`
3. la definizione per casi di pop ha un caso solo per `::`
4. da cui il warning
5. il compilatore inserisce automaticamente una eccezione `Match` nei casi mancanti

- Il programmatore può definire le proprie eccezioni:

```
exception EmptyStack;      (* dichiara una nuova eccezione *)
fun pop(x::s) = (x,s)
| pop [] = raise EmptyStack; (* come il throw di Java *)
```

Il risultato in caso di errore è più esplicativo dell'eccezione "automatica" Match

```
- pop [];
uncaught exception EmptyStack
```

- Le eccezioni possono essere catturate e gestite con handle:

```
pop x
handle EmptyStack =>
  ( print "messaggio di errore specializzato";
  raise EmptyStack );
```

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

- un singolo handle può gestire diverse eccezioni

```
<expression> handle
  <exception 1> => ...
  | <exception 2> => ...
  | ...
```

Ogni ordine è buono: perchè?

- Due modi di usarlo in ML funzionale puro:

1. fare qualcosa come stampare un messaggio e *rilanciare l'eccezione*
2. "aggiustare" l'errore restituendo un valore *dello stesso tipo* dell'espressione che ha sollevato l'eccezione

Sono le uniche opzioni che passano il type checking senza errori

- Altro esempio (da non seguire acriticamente ...)

```
- fun pos x (y::z) =  
    if (x=y) then 1 else 1 + pos x z;
```

Questa funzione restituisce la posizione di x nella lista, ma se non trova x solleva una eccezione (manca un caso terminale per [])

- Si può usare handle per modificare pos per restituire -1 quando non trova x nella lista:

```
- fun pos2 x y = (pos x y) handle Match => -1;
```

- Esempio didattico un po' artificiale: si potrebbe obiettare che pos è realizzata male...

- Si possono aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni

- Esempio di eccezione con parametri:

```
exception SyntaxError of string
```

- Questa eccezione può essere lanciata in diversi modi...

```
raise SyntaxError "Identifier expected"  
raise SyntaxError "Integer expected"
```

- ... e il parametro "letto" col pattern matching

```
... handle SyntaxError x => L.. (* qui si può usare x *)
```

8.13 Esempio compilazione di espressioni

Vedere l'esempio dalle slide.

Capitolo 9

Paradigma logico

Per scaricare l'ambiente cliccare sul seguente link: <https://www.swi-prolog.org/>.

E' un paradigma **dichiarativo**. I programmi sono **insiemi di assiomi**, la computazione consisterà nello sviluppo di **dimostrazioni** costruttive di una formula logica data detta **query**, mediante gli assiomi del programma. Quindi noi forniremo da una parte l'insieme degli assiomi (che possiamo chiamare programma) e una query, l'interprete o il compilatore poi cerca di dimostrare se la query è vera o no oppure dare ulteriori informazioni. Per esempio:

- **programma:**

- Socrate e' un uomo (assioma 1).
- Tutti gli uomini sono mortali (assioma 2)

- **Query 1:**

- Socrate e' un uomo ?
- risposta: true

- **Query 2:**

- Socrate e' mortale ?
- risposta: true

In prolog:

```
umano(socrate). (fatto)
mortale(X):- umano(X). (regola)

umano(socrate).
risposta: true

mortale(socrate).
risposta: true
```

La regola significa $mortale(X) \Leftarrow umano(X)$.

L'**implementazione** di un programma prolog è mista:

- I programmi Prolog vengono compilati in un bytecode che viene interpretato da una macchina virtuale la Warren Abstract Machine (WAM).
- Si possono anche compilare i programmi in codice stand-alone, direttamente eseguibile.

L'**interpretazione** con Prolog è analoga a quella con ML:

- Si inviano query all'interprete e si ottengono le relative risposte.
- Oppure, se il programma è stand alone, si interagisce mediante la sua UI (user interface).

9.1 Costrutti base

Abbiamo tre tipi di statement:

1. **Fatti (facts)**.
2. **Regole (rules)**: che sono assiomi con l'implicazione
3. **Queries (goals)**.

Abbiamo una sola struttura dati, che dopo vedremo:

1. **termini logici (logical terms)**.

9.1.1 Facts

Lo scopo di un fatto è quello di stabilire una relazione tra oggetti, ad esempio:

```
father(abraham, isaac).
```

Abramo è il padre di Isacco. **father** oltre che **relazione** è chiamato anche **predicato**. I nomi dei predicati devono iniziare per lettera minuscola. Gli argomenti abraham e isaac iniziano con lettera minuscola perché sono costanti, come i numeri, introdurremo le variabili che si distinguono per essere maiuscole più avanti.

Con i fatti possiamo definire un database. Ogni predicato corrisponde a una tabella relazionale.

<code>father(terach,abraham).</code>	<code>male(terach).</code>
<code>father(terach,nachor).</code>	<code>male(abraham).</code>
<code>father(terach,haran).</code>	<code>male(nachor).</code>
<code>father(abraham,isaac).</code>	<code>male(haran).</code>
<code>father(haran,lot).</code>	<code>male(isaac).</code>
<code>father(haran,milcah).</code>	<code>male(lot).</code>
<code>father(haran,yiscah).</code>	<code>female(sarah).</code>
<code>mother(sarah,isaac).</code>	<code>female(milcah)</code>
	<code>female(yiscah)</code>

9.1.2 Queries

I programmi logici sono fatti per rispondere a queries. Le query hanno la stessa forma dei fatti e sono entrambi dei cosiddetti **atomi logici**, nei programmi sono asserzioni (fatti) nelle query sono domande:

```
?— father(abraham, isaac).  
true
```

```
?— father(isaac, abraham).  
false
```

Nell'esempio qui sopra, l'interprete trova il primo atomo nel programma e non trova il secondo perciò le risposte sono diverse.

9.1.3 Variabili logiche nelle query

A volte possiamo chiederci ad esempio, quali sono i figli di abraham ? Per poter fare questa domanda possiamo usare le **variabili logiche** che si riconoscono perché iniziano con una lettera maiuscola:

```
?— father(abraham, X).  
X = isaac.
```

La query che ho scritto significa: esiste X tale che `father(abraham, X)` ? Se ad esempio un padre ha più figlio se scrivo ; l'interprete mi fornisce gli altri figli:

```
?— father(terach, X).  
X = abraham;  
X = nachor;  
X = haran.
```

La macchina virtuale cerca i valori che sostituiti a X rendono la query uguale a uno dei fatti del programma, questo però vale per i programmi di soli fatti, vedremo poi come funzionano gli algoritmi del linguaggio.

9.1.4 Variabili logiche in generale

Notiamo le differenze tra le variabili logiche e le variabili degli altri paradigmi:

- Le variabili logiche rappresentano oggetti qualsiasi, non specificati.
- Le variabili dei linguaggi imperativi sono locazioni di memoria.
- Gli identificatori dei linguaggi funzionali denotano i valori immutabili.

E' possibile usare una variabile anche in un fatto:

```
creato_da(X, dio).
```

Questo fatto significa che ogni cosa è creata da Dio. La differenza tra le variabili nei fatti e nelle query è che:

- Nelle query: esistenzialmente quantificate, esiste un X tale che...?
- Nei fatti: universalmente quantificate, per ogni X vale che...

9.1.5 Termini

I **termini** sono l'unica struttura dati in Prolog, si costruiscono con costanti, variabili (logiche) e funtori (funzione matematica). Hanno un ruolo simile ai costruttori di ML cioè sono caratterizzati da un nome e da un arietà (il numero di argomenti):

```
<term> :: = <constant> | <variabile>
          | <funcotor name> '( '[<term> [ , ' <term>]*] )'
```

Per esempio:

```
successor( Int )
date( 25, april, 2020 )
color( rgb, 0, 0, 1 )
```

Non ci sono dichiarazioni di tipo e le costanti simboliche e i funtori si usano senza dichiararli prima.

Gli argomenti di un predicato possono essere termini qualsiasi:

```
born(john, date(10, october, 2000))
```

Quindi la sintassi generale dei fatti è:

```
<fact> ::= <atomic formula> ','  
<atomic formula> ::=  
    <predicate> '(' [<term> [, , <term>]*] ')'
```

Un fatto è una formula atomica seguita da un punto, una formula atomica è un predicato applicato a 0 o più termini. La stessa variabile X può comparire più volte nello stesso fatto o nella stessa query, significa che i termini nelle posizioni dove si trova X devono essere uguali, ricorda un pò il pattern matching ma non avevamo questa caratteristica, perché in ML il pattern matching serve solo a estrarre informazioni da una struttura.

9.1.6 Termini ground e nonground

Un termine è **ground** se non contiene variabili, altrimenti è nonground:

```
foo (a , b) ground  
bar (X) nonground
```

Gli stessi aggettivi e gli stessi criteri si applicano ai fatti alle query e anche alle regole (che poi vedremo):

```
father (abraham , isaac ) ground  
sum(X , 0 , X) nonground
```

9.2 Come si risponde alle query

Per spiegarlo dobbiamo specificare come avviene il matching tra query e fatti e come si costruiscono le risposte, dovremmo introdurre due nozioni:

- **Sostituzione.**
- **Unificazione.**

9.2.1 Sostituzione

Una sostituzione è un insieme finito di coppie:

$$\Theta = \{ X_1 = t_1, \dots, X_n = t_n \}$$

dove:

- Le X_i sono variabili e i t_i sono termini.
- Le X_i sono tutte diverse tra loro.

- Nessuna delle X_i compare dentro i t_i .

Le sostituzione si applica ad una espressione E (che potrebbe esser un termine, un fatto, una query o una regola) che si denota con $E\Theta$ e sostituisce le occorrenze delle variabili X_i in E con i rispettivi termini t_i , $E\Theta$ è una nuova espressione. Vediamo un esempio:

$$\begin{aligned}\Theta &= \{ X = isaac \} \\ E &= father(abraham, X) \\ E\Theta &= (abraham, isaac)\end{aligned}$$

9.2.2 Istanze

Applicare la sostituzione ad un'espressione E crea un caso particolare di E, dove le variabili di E (che indicano oggetti non specificati) vengono sostituite con valori specifici (termini ground) o parzialmente specificati (termini nonground).

Definizione: E1 è un'istanza di E2 se esiste una sostituzione Θ tale che $E1 = E2\Theta$, cioè se E1 è un caso particolare di E2 dove alcune variabili di E2 sono istanziate, cioè legate a un valore.

9.2.3 Unificazione

L'algoritmo di unificazione è quello che effettua il matching che consente di rispondere ad una query in senso positivo o negativo. Prende due espressioni E1 e E2 e se possibile restituisce una sostituzione Θ tale che $E1\Theta = E2\Theta$ e quindi le ha unificate. L'algoritmo di unificazione va alla ricerca di un particolare unificatore detto **most general unifier (mgu)** cioè va alla ricerca della sostituzione più generale possibile che rende due istanze uguali, quindi l'mgu non sostituisce una variabile con un termine se non è necessario e quindi vincola il meno possibile il risultato $E1\Theta$ lasciando le variabili libere quando può. **In particolare:** Θ è mgu se è un unificatore e inoltre ogni altro unificatore Θ' porta a un'istanza di $E1\Theta$ cioè ogni altro unificatore porta a un caso più particolare di quello dato da Θ . Vediamo degli esempi:

- ◆ $\text{mgu}(\text{sum}(A, B, C), \text{sum}(X, 0, X)) = \{A = X, B = 0, C = X\}$
- ◆ $\text{mgu}(\text{sum}(0, X, 0), \text{sum}(Y, 0, Y)) = \{X = 0, Y = 0\}$
- ◆ $\text{mgu}(\text{sum}(0, X, 0), \text{sum}(Y, Z, 1))$ non esiste a causa del terzo argomento (non si può rendere $0=1$)

Andare alla slide 25 per degli esercizi.

9.2.4 Costruzione delle risposte da soli fatti

Abbiamo un programma che consiste di soli fatti e una query:

$$q(t_1, \dots, t_n)$$

Le risposte vengono costruite così

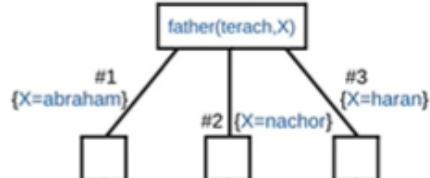
1. Si cerca nel programma il primo fatto $p(u_1, \dots, u_m)$ con lo stesso funtore (cioè $p = q$ e $m = n$) se se ne trova uno:
2. si calcola $\Theta = \text{mgu}(q(t_1, \dots, t_n), p(u_1, \dots, u_m))$, se Θ esiste allora:
 - (a) si restituisce Θ come risposta (se è vuota allora Prolog dice true)
 - (b) Poi se l'utente non vuole altre soluzioni si termina.
3. Altrimenti si cerca il prossimo fatto con lo stesso funtore. Se esiste si ripete da 2 altrimenti si termina

Prolog dice false quando nessun fatto unifica con la query. A questo punto prima di andare avanti consiglio di leggere la sezione Interpretazione logica.

9.2.5 Rappresentazione grafica del procedimento e conjunctive queries

- Un *search tree* per la query `father(terach,X)`.

```
/* programma */
/*1*/ father(terach,abraham).
/*2*/ father(terach,nachor).
/*3*/ father(terach,haran).
/*4*/ father(abraham,isaac).
...
```



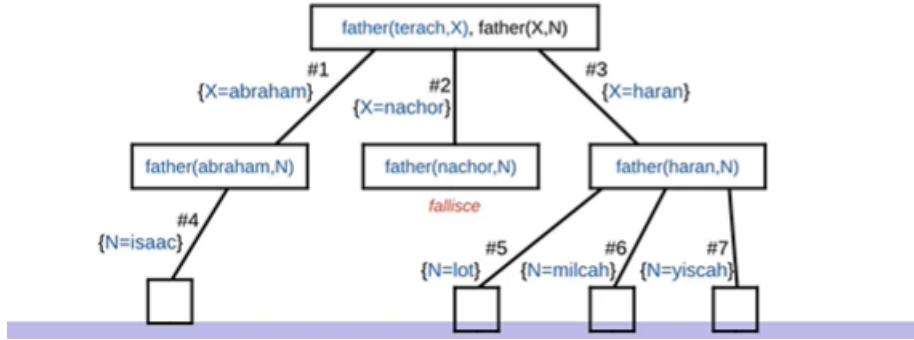
- Gli archi corrispondono ai fatti che unificano con la query
- Sono etichettati con
 - ◆ il numero del fatto utilizzato (nell'ordine in cui compare nel programma)
 - ◆ il mgu della query e del fatto
- Il search tree di una query (rispetto a un programma) comprende tutte le risposte che Prolog genera se l'utente glielo chiede
- Le computazioni di Prolog corrispondono a una visita depth-first da sinistra a destra (ogni ramo di successo una risposta)

La sequenza è rilevante se ci fai caso.

- Le query possono contenere più formule atomiche (*goals*)
- Esempio: chi sono i nipoti di terach?

`!father(terach,X), father(X,Nipote)`
 (la virgola è un *and*) è come un *join*

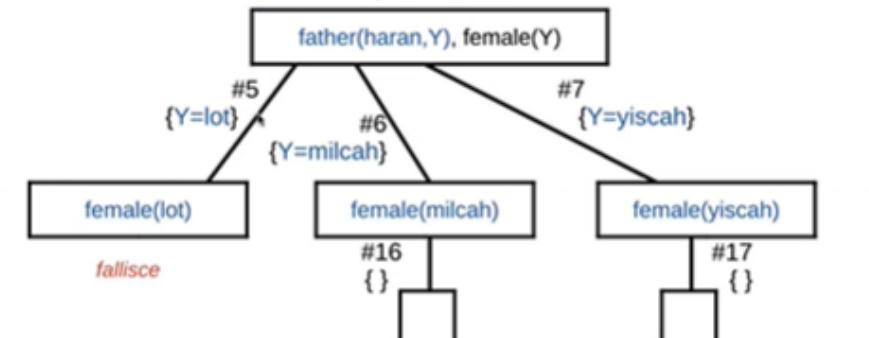
```
/* programma */
/*1*/ father(terach, abraham).
/*2*/ father(terach, nanchor).
/*3*/ father(terach, haran).
/*4*/ father(abraham, isaac).
/*5*/ father(haran, lot).
/*6*/ father(haran, milcah).
/*7*/ father(haran, yiscah).
...
```



- Le figlie di haran le trovo con la query congiuntiva ... ?

`father(haran,Y), female(Y).`

I valori di Y mi danno le figlie di haran



```
Y = milcah;
Y = yiscah;
false.
```

9.3 Interpretazione logica

Per ora abbiamo visto le operazioni di un programma funzionale, vediamo ora il punto di vista dichiarativo del programma, cioè vediamo cosa descrive un programma funzionale. Partiamo da un esempio:

```
umano( socrate ).
```

L'interpretazione logica prevede che esista un dominio di individui Dom ed è come se fosse un unico tipo di dati, cioè i valori appartengono tutti a Dom. Umano di socrate dice che socrate appartiene a Dom e denota un individuo, umano è un sottoinsieme di Dom oppure è una funzione $Dom \rightarrow \{ true, false \}$, poi applicando la funzione umano(socrate) ritorna true. Umano è un predicato e socrate è una costante, le costanti denotano elementi del dominio e i predicati denotano funzioni booleane dal dominio.

Vediamo un altro esempio:

```
hasColor( obj , color( rgb , 0 , 0 , 1 ) )
```

Le costanti sono tipi particolari di termini e i predicati sono tipi particolari di formule. rgb, 0, 0, 1 e obj sono costanti, hasColor è un predicato mentre color è un **funtore** (simbolo funzionale) cioè una funzione da elementi del dominio a elementi del dominio: $color : Dom^4 \rightarrow Dom$. In pratica siccome l'esempio è un fatto per forza la cosa più esterna (hasColor) è un predicato perché un fatto è una cosa affermata vera, il predicato poi si applica a dei termini e quindi se un termine non ha argomenti è una costante, invece se ha argomenti è un funtore.

9.4 Regole

Vediamo un esempio: X è figlia di Y se Y è padre di X e X è femmina:

```
daughter(X, Y) :- father(Y, X), female(X).
```

In generale la sintassi delle regole è: `:-` sta per \Leftarrow , la formula atomica prima

```
<rule> ::= <atomic formula> [:- <goal list>].  
  
<goal list> ::= <atomic formula> [, <atomic formula>]*
```

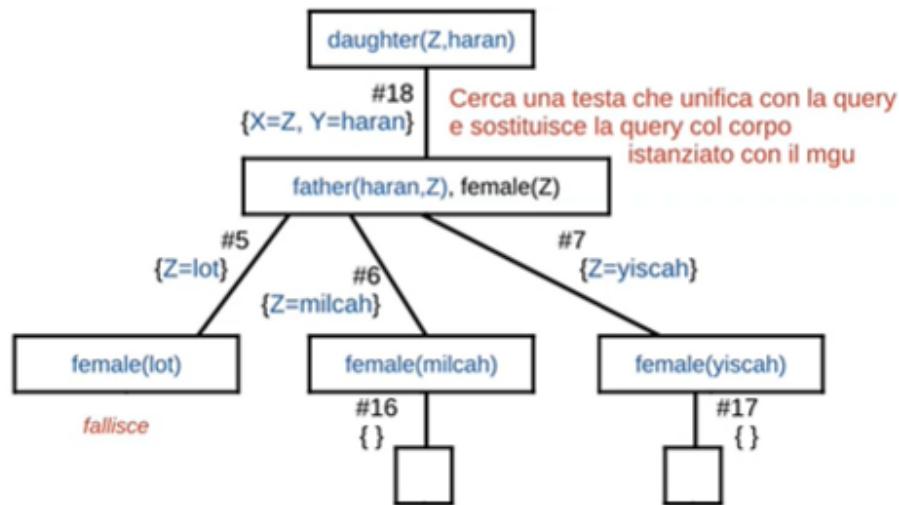
di `:-` è detta **testa**, la parte dopo `:-` è detta **corpo**. Notiamo che i fatti sono regole senza corpo perché abbiamo tutta la parte opzionale che se omettiamo otteniamo un fatto. Quindi un **programma logico** è un insieme di regole, in prolog invece è una **sequenza** di regole.

9.5 Ragionare con le regole

Generalizziamo l'algoritmo della costruzione delle risposte aggiungendo le regole. Aggiungiamo in coda al programma (posizione 18) la regola:

```
daughter(X, Y) :- father(Y, X), female(X).
```

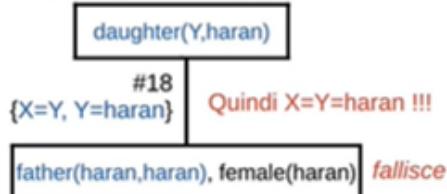
Le risposte alla query `daughter(Z, haran)` si trovano così:



Bisogna fare attenzione con le variabili, quelle della query devono sempre essere diverse da quelle della regola, ad esempio:

```
daughter(X, Y) :- father(Y, X), female(X).
```

Le risposte alla query `daughter(Y, haran)` verrebbero perse

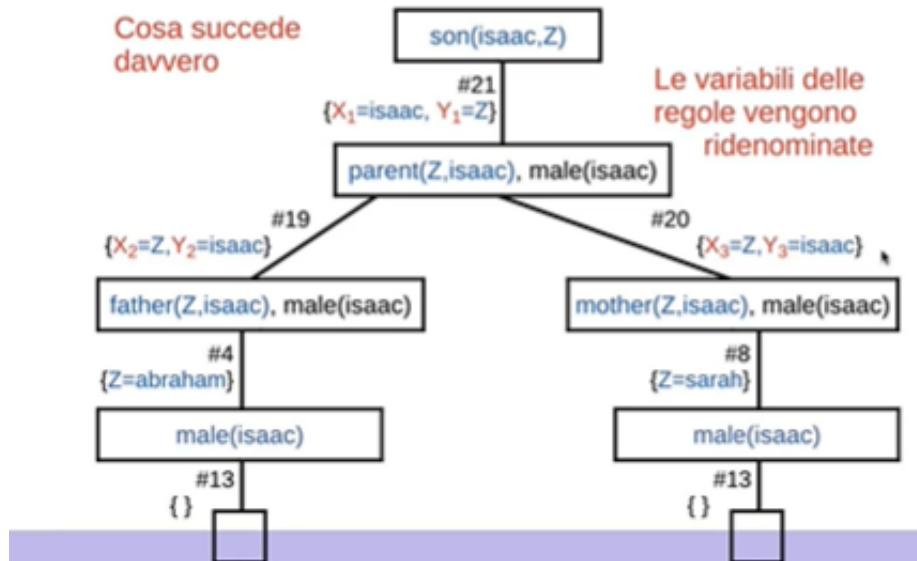


- Per evitare questo problema, ogni volta che una regola viene usata la WAM ridenomina le sue variabili, ad esempio

```
daughter(_101, _102) :- father(_102, _101), female(_101).
```

La tecnica di ridenominazione si chiama **standardization apart**. Vediamo un esempio più completo:

```
/* 4*/ father(abraham,isaac). ...
/* 8*/ mother(sarah,isaac). ...
/*13*/ male(isaac). ...
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```



9.5.1 Overloading

Si può usare lo stesso nome di predicato con numeri diversi di argomenti. Simili prediciati vengono trattati come prediciati diversi:

- Esempio: dall'informazione che "X è madre di Y" derivare il concetto di essere madre

```
mother(Mom) :- mother(Mom,X).
/* Mom è una mamma se esiste un X tale che Mom è madre di X */
/* Notare che abbiamo un mother unario e uno binario */
```

9.5.2 Wildcards

Notiamo che nel goal `mother(Mom, X)` il valore di `X` non interessa in questi casi possiamo usare dei wildcards simili a ML:

```
mother(Mom) :- mother(Mom, _).
```

Attenzione: se usiamo due volte una variabile `X` allora in quei punti devo avere lo stesso valore, mentre ogni wildcard può essere associata a un valore diverso:

```
mother(X, X).  
false (nessuna e madre di se stessa)  
mother( _, _ ).  
true (cerca un fatto mother(X, Y))
```

9.5.3 Esempio analisi di circuiti

Vedere dalle slide 39.

9.5.4 Regole ricorsive

Vedere dalle slide 41.

9.5.5 Query non terminanti

Invece delle regole viste prima:

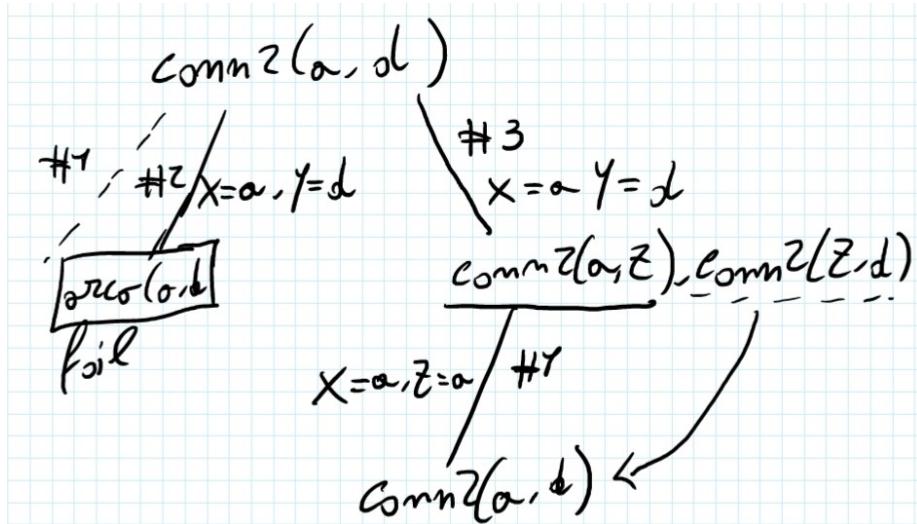
```
connessi(X, X).  
connessi(X, Y) :- arco(X, Z), connessi(Z, Y).
```

Consideriamo le seguenti:

```
connessi2(X,X)  
connessi(X, Y) :- arco(X, Y).  
connessi(X, Y) :- connessi2(X, Z), connessi2(Z, Y).
```

Dal punto di vista logico è corretto ma dal punto di vista prolog invece non lo è perché la query (vera): `connessi2(a, d)` va in loop infinito e da come errore Stack limit probable infinite recursion (cycle).

Vediamo il search tree di questa query:

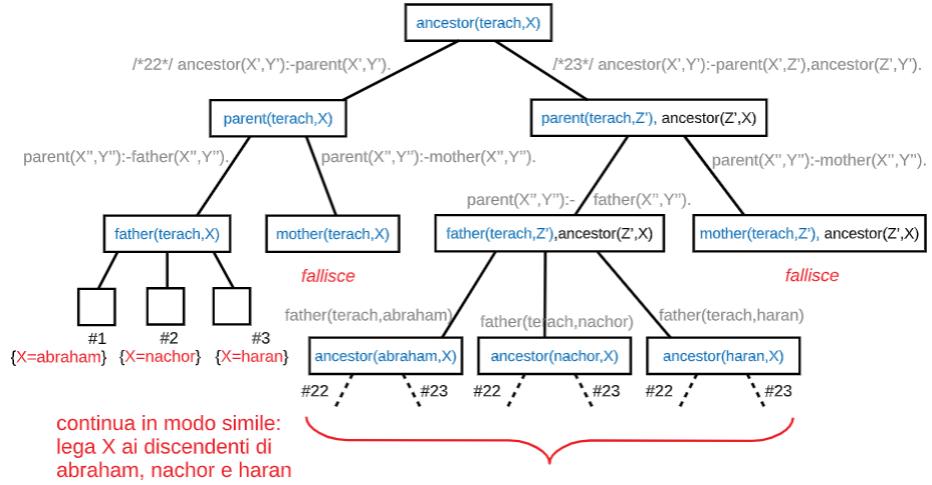


Dobbiamo fare attenzione anche alla prima versione di connessi perché se abbiamo un grafo ciclico abbiamo lo stesso problema, per risolvere questo problema ci serviranno dei nuovi costrutti. Vediamo adesso un esempio simile ma senza i problemi che abbiamo riscontrato:

- Aggiungiamo ora al programma sulla genealogia biblica la nozione di **antenato** (*ancestor*)
- Gli antenati di *X* sono i suoi genitori, nonni, bisnonni, ...
- Definizione ricorsiva: *X* è un antenato di *Y* **se** vale una di queste condizioni:
 1. *X* è genitore di *Y* (caso base)
 2. *X* è genitore di *Z* e *Z* è antenato di *Y* (per qualche *Z*)

```

ancestor(X,Y) :- parent(X,Y).
b. ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
  
```



9.5.6 Prolog e algebra relazionale

Una tabella relazionale possiamo rappresentarla come un elenco di fatti, con le regole si può facilmente simulare l'algebra relazionale, prolog genera le n-uple della risposta una per una.

r :

a_1	b_1	c_1	d_1
a_2	b_2	c_2	d_2
a_3	b_3	c_3	d_3
\vdots	\vdots	\vdots	\vdots

$r(a_1, b_1, c_1, d_1).$ $r(a_2, b_2, c_2, d_2).$ $r(a_3, b_3, c_3, d_3).$ $.$ $.$ $.$

```

/* UNIONE di r e s */
r_union_s(X1,...,Xn) :- r(X1,...,Xn).
r_union_s(X1,...,Xn) :- s(X1,...,Xn).

/* INTERSEZIONE di r e s */
r_inters_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).

/* PROIEZIONE di r, ad es. su colonne 1 e 3 */
r13(X1,X3) :- r(X1,...,Xn).

/* SELEZIONE di r */
r_sel(X1,...,Xn) :- r(X1,...,Xn), <condizione>.

/* ad esempio: */
r_with_2_less_than_3(X1,...,Xn) :- r(X1,...,Xn), X2 < X3.

```

- Per la differenza tra relazioni occorre un operatore di **negazione** denotato da $\setminus+$ ¹

```

/* DIFFERENZA INSIEMISTICA tra r e s */
r_minus_s(X1,...,Xn) :- r(X1,...,Xn), \+ s(X1,...,Xn).

```

- La negazione trasforma false in true, ovvero fallimenti in successi.
 - ◆ $\setminus+$ $p(X_1, \dots, X_n)$ è *true* se tutti i rami del suo albero di ricerca terminano con un fallimento
 - ◆ L'albero deve essere finito. Se un programma cade in una ricorsione infinita ovviamente non termina.

9.6 Liste

La rappresentazione è analoga a quella in ML:

- **Costruttore lista vuota:** []
- **costruttore nodi:** [elem—resto]

Altre annotazioni equivalenti:

Abbreviata	Costruttori esplicativi
[a]	[a []]
[a , b]	[a [b []]]
[a , b , c]	[a [b [c []]]]
[a X]	[a X]
[a , b X]	[a [b X]]
[a , b , c X]	[a [b [c X]]]

- Notare la possibilità di esprimere liste *parzialmente specificate*, dove alcuni elementi ed eventualmente la coda sono variabili.

[a , **X** , b | **Y**]

Vedere l'esempio di member dalle slide 56 a 60.

9.6.1 Evanescenza di parametri di input e output

Non c'è una chiara distinzione tra parametri IN e OUT:

- Ogni parametro può essere legato a un termine con costruttori (input).
- Ogni parametro attuale con una variabile libera produce delle sostituzioni (output).
- I parametri con almeno un costruttore e una variabile forniscono sia un input sia un output.

Guardiamo gli esempi con member:

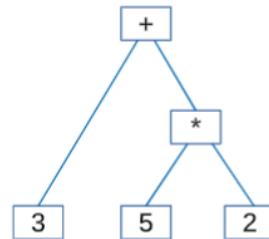
- member(X, <lista ground>) prende una lista e restituisce i suoi elementi, modalità out, in.
- member(<elem. ground>L) prende un elemento e restituisce le liste che lo contengono, modalità in, out.

In programmazione logica una volta definito un predicato tutti i suoi argomenti possono essere usati sia in ingresso che in uscita simmetricamente, sfrutteremo questa simmetria per il predicato append che però devi farlo tu. Vedere slide 64.

9.7 Calcolo simbolico in Prolog

Gli operatori in prolog sono presenti ma sono dei costruttori:

- ◆ $3+5*2$ non è uguale a 13!
- ◆ denota invece l'albero sintattico qui a destra
- ◆ Per calcolare $3+5*2$ usare **is**:
Ris is 3+5*2
- ◆ questo goal è vero se Ris è uguale al valore di $3+5*2$



- Questo rende il calcolo simbolico particolarmente semplice.
Facciamo un esempio basato sulle derivate, stile Matlab

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```

der(X, X, 1).
der(X^N, X, N*X^(N-1)) :- 
    N is N-1.
der(sen(X), X, cos(X)).
der(cos(X), X, -sen(X)).
der(log(X), X, 1/X).
der(F+G, X, DF+DG) :- 
    der(F, X, DF), der(G, X, DG).
der(F-G, X, DF-DG) :- 
    der(F, X, DF), der(G, X, DG).
der(F*G, X, F*DG+DF*G) :- 
    der(F, X, DF), der(G, X, DG).
...
  
```

```

?- der(x^3*cos(x), x, D).
D = x^3* -sen(x)+3*x^2*cos(x).
  
```

[Si possono aggiungere predicati per semplificare il risultato]

9.8 Cammini su grafi

Ora possiamo definire i cammini su un grafo, supportando anche i cicli. Definiamo il predicato connessiEvitando(X, Y, Avoid) che sarà vero se c'è un cammino da X a Y che evita i nodi nella lista Avoid.

```
connessiEvitando(X, X, Avoid) :- \+ member(X, Avoid),  
connessiEvitando(X, Y, Avoid) :- arco(X, Z),  
\+ member(Z, Avoid), connessiEvitando(Z, Y, [X|Avoid])
```

A questo punto possiamo definire la terza versione di connessi a cui passiamo tre parametri:

```
connessi3 :- connessiEvitando(X, Y, []).
```

9.9 Programmazione nondeterministica

E' un pattern che consiste nel **non** dire a prolog **come** trovare la soluzione di un problema ma dire **cosa** è una soluzione e si lascia che prolog la cerchi con il backtrack (visita del search tree). Lo schema generale di questo pattern consiste in una regola base (**generete and test**) del tipo:

```
solution(X) :- generete(X), test(X).
```

Cioè X è una soluzione se X è generato da certe regole che dicono sintatticamente le possibili soluzioni e X supera un test che dice se X è effettivamente una soluzione, cioè divide la ricerca in un processo di generazione di tipo sintattico che genera tutte le soluzioni possibili, e un test che filtra quelle che sono davvero soluzioni ammissibili al nostro problema. Se la soluzione che è stata unificata con generete e poi testata fallisce il test, prolog automaticamente proverà a generare un'altra soluzione sfruttando il backtrack. Se si chiedono altre risposte, genera altre soluzioni.

La programmazione nondeterministica è una caratteristica unica del paradigma logico, possiamo cercare di approssimarla negli altri paradigmi:

```
/* paradigma imperativo */
X := primo_candidato;
while not test(X) and X ≠ null do
    X := prossimo_candidato(X)
return X

/* paradigma funzionale */
fun soluzione( X ) =
    if test(X) then X
    else soluzione( prossimo_candidato(X) )

/* invocare così */
soluzione( primo_candidato );
```

In entrambi i casi verrebbe generata solo la prima soluzione trovata (se si vogliono le altre occorre complicare il codice). Sta cosa è super op tanto che dopo il professore programma un AI di un gioco. Per ora vediamo un esempio di una ricerca dei membri pari di una lista:

■ Ricerca dei membri pari di una lista

```
/* stile generate and test */
membro_pari(X,L) :- member(X,L), 0 is X mod 2.

/* invece di ricorsione ad hoc */
membro_pari(X,[X|_]) :- 0 is X mod 2.
membro_pari(X,[_|Resto]) :- membro_pari(X,Resto).
```

■ Notare come l'approccio generate & test possa giocare il ruolo delle funzioni di ordine superiore in ML

- ◆ permette di comporre facilmente nuovi predicati da quelli dati
- ◆ in questo caso `member`, che diventa uno strumento generale per visitare una lista
- ◆ funge da *filter*: la query congiuntiva

```
member(X,Lista), predicato(X).
```

è l'analogo di: `filter predicato Lista`



9.9.1 Applicazione ai giochi

- Un possibile pattern generate-and-test per i giochi

```
next_move(Player, Move) :-  
    possible_move(Player, Move), optimal(Player, Move).
```

- A sua volta il controllo di ottimalità (cioè verificare se con Move sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test
 - Nota: *optimal* mi dice se Move è la prima mossa di una strategia di gioco ottima → posso usare *optimal* per analizzare il gioco
 - ◆ ad es. il primo giocatore ha una strategia vincente?
 - ◆ mostrerà che è utile generare *tutte* le soluzioni

Vediamo ora l'esempio con il tris. Vedremo prima il codice per l'intelligenza del programma e poi il codice per i turni e l'interfaccia utente.

Rappresentazione della scacchiera

Rappresentiamo la scacchiera come una lista di 3 righe (liste):

```
start ([[1 ,2 ,3] ,[4 ,5 ,6] ,[7 ,8 ,9]]).
```

Questa è la configurazione iniziale della scacchiera, quindi abbiamo un predicato start che è verso solo su questa lista di liste.

I giocatori

Introduciamo due simboli x e o per i giocatori. Quando un giocatore fa una mossa, il suo simbolo occuperà quella casella sulla scacchiera al posto del numero.

```
adversary (x,o).  
adversary (o, x).
```

Condizione di vittoria

Abbiamo visto fino ad ora fatti ground dove avevamo delle costanti, ora per definire le condizioni di vittoria dobbiamo usare dei fatti non ground:

```
win(P, [[P,P,P] ,-,-]).  
win(P, [-,[P,P,P] ,-]).
```

```

win(P, [_, _, [P, P, P]]).

win(P, [[P, _, _], [P, _, _], [P, _, _]]).
win(P, [[_, P, _], [_, P, _], [_, P, _]]).
win(P, [[_, _, P], [_, _, P], [_, _, P]]).

win(P, [[P, _, _], [_, P, _], [_, _, P]]).
win(P, [[_, _, P], [_, P, _], [P, _, _]]).

```

Spieghiamo a voce una delle condizioni, la prima ci dice che avere lo stesso simbolo sulla prima riga orizzontata porta alla vittoria, lo stesso ragionamento si applica alle altre condizioni. Le prime tre descrivono le condizioni di vittoria orizzontale, poi abbiamo quelle verticali e infine le ultime due sono quelle diagonali.

Situazione non finale

Definiamo una situazione non finale, un predicato che riceve un argomento, Board, che è vero se Board descrive una scacchiera in cui la partita non è ancora finita, cioè nessuno ha vinto e la scacchiera non è piena:

```

nonFinal(Board) :-  

  \+ win(_, Board),  

  member(Row, Board),  

  member(Cell, Row),  

  number(Cell).

```

Spieghiamo: non final riceve una scacchiera Board, per essere vero non final allora win deve essere falso, però dobbiamo controllare anche che non ci troviamo in una situazione di patta, quindi il primo member cerca una riga nella board, da questa riga viene cercata una cella che contiene un numero, sfruttiamo il predicato built in number che ci dice proprio se una certa variabile è un numero, se esiste una cella che contiene un numero allora vuol dire che le mosse non sono ancora finite, per questo motivo abbiamo definito lo stato di inizio con dei numeri.

Situazione finale

Molto semplice:

```
final(Board) :- \+ nonFinal(Board).
```

Possibili mosse

Descriviamo ora la dinamica del gioco, cioè come si passa da uno stato di una scacchiera ad un altro, quindi descriviamo le mosse e i loro effetti. Lo schema è il seguente:

```
move(+P, +N, +Board1, +Board2)
```

Dove:

- **P (player):** è il simbolo del giocatore che fa la mossa.
- **N:** è la mossa indicata dal numero della cella da 1 a 9.
- **Board1:** è la scacchiera prima della mossa.
- **Board2:** è la scacchiera dopo la mossa.

Questo predicato sarà vero se Board2 è la scacchiera ottenuta da Board1 con la mossa N del giocatore P. I simboli + e - sono semplice documentazione, + indica che l'argomento è di tipo In, - vuol dire che è di tipo Out, mentre se non mettiamo niente vuol dire che l'argomento è In-Out, è solo utile al programmatore, abbiamo già visto che i parametri in Prolog sono invertibili. Definiamo ora il predicato:

```
move(P, N, Board1, Board2) :-  
  \+ win(_, Board1),  
  append(RowsBefore, [Row | RowsAfter], Board1),  
  append(CellsBefore, [N| CellsAfter], Row),  
  number(N),  
  append(CellsBefore, [P| CellsAfter], NewRow),  
  append(RowsBefore, [NewRow | RowsAfter], Board2).
```

La prima cosa da vedere è che nessuno ha ancora vinto. Ricordiamo che append fa la concatenazione di liste, questi due append dicono che N che è la mossa che vogliamo fare è presente in Board1, servono due append perché Board1 è una lista di liste, infatti il primo prende la riga mentre il secondo prende la cella. Più nello specifico: il primo scomponete Board1 in tre parti, le righe che precedono la riga che contiene N, la riga Row che contiene n e le righe che seguono Row, poi il secondo append fa la stessa cosa ma andando a prendere proprio al cella che contiene N. Adesso che abbiamo trovato N in Board1, dobbiamo sostituire N con P, per farlo creiamo una nuova riga che è composta dalle celle precedenti di N, P e le celle successive ad N, ottenuta questa nuova riga, andiamo a concatenarla con le righe precedenti a quella che conteneva N, la nuova riga e le righe successive a quella che conteneva N, otteniamo così Board2. Con questa regola abbiamo descritto tutte le mosse del tris.

Le strategie

Ora vediamo i predicati che useremo per l'analisi strategica, introduciamo i predicati chiamati:

```
hasXXXStart(+P, -Move, +Board)
```

Dove:

- **P** è il giocatore.
- **Move** è la prima mossa della strategia.

- **Board** è l'attuale scacchiera.

Le X stanno per strategia vincente o non perdente, Definiamo adesso le regole:

```
hasWinStrat(P, _, Board) :-  
    win(P, Board).
```

```
hasWinStrat(P, Move, Board) :-  
    move(P, Move, Board, Board2),  
    adversary(P, Adv),  
    \+ hasTieStrat(Adv, _, Board2).
```

Se Board è già una situazione di win per P allora P ha una win strategy con qualsiasi mossa perché la partita è già vinta, altrimenti il secondo caso in cui P ha una win strategy a partire dallo stato Board e la prima mossa ottimale è Move è il seguente: il giocatore P può fare move ottenendo Board2 a partire da Board, Adv è l'avversario di P e Adv non ha una strategia per pareggiare a partire da Board2.

Vediamo ora la strategia di pareggio:

```
hasTieStrat(_, _, Board) :-  
    final(Board),  
    \+ win(_, Board).
```

```
hasTieStrat(P, Move, Board) :-  
    move(P, Move, Board, Board2),  
    adversary(P, Adv),  
    \+ hasWinStrat(Adv, _, Board2).
```

In questo caso qualsiasi giocatore ha una strategia per pareggiare quando board è già un pareggio e quindi quando ci troviamo in uno stato finale e nessuno ha vinto. Il secondo caso invece abbiamo un giocatore P e una mossa ottimale (Move) che si può fare e il suo avversario non ha una strategia vincente a partire da Board2 (ottenuta con la mossa Move del giocatore P) con qualsiasi mossa.

9.9.2 Analisi del gioco

Possiamo chiedere se esiste una strategia vincente per il primo giocatore:

```
start(Board), hasWinStrat(x, Move, Board).  
false
```

Non esiste nessuna mossa iniziale che garantisce al primo giocatore di vincere. Oppure possiamo chiedere se esiste una strategia vincente per il secondo giocatore:

```
start(B0), hasTieStart(x, Mv, B0).  
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],  
Mv = 1;  
...
```

```

B0 = [[1 ,2 ,3] , [4 ,5 ,6] ,[7 ,8 ,9]] ,
Mv = 9;
false .

```

Ogni mossa iniziale permette al primo giocatore di pareggiare (se non commette errori), tutte le mosse iniziali fanno parte di una strategia di pareggio. Ora chiediamo come può pareggiare il secondo giocatore:

```

start(B0) , move(x , 1 , B0 , B1) , hasTieStrat(o , Mv , B1) .
B0 = [[1 ,2 ,3] , [4 ,5 ,6] ,[7 ,8 ,9]] ,
B1 = [[1 ,2 ,3] , [4 ,5 ,6] ,[7 ,8 ,9]] ,
Mv = 5;
false .

```

Se la prima mossa è su un angolo, l'unica risposta è 5, in tutti gli altri casi vince il primo giocatore. Poi abbiamo:

```

start(B0) , move(x , 2 , B0 , B1) , hasTieStrat(o , Mv , B1) .
...
Mv = 1;
...
Mv = 3;
...
Mv = 5;
...
Mv = 8;
false .

```

Se la prima mossa è 2, non rispondere con 4, 6, 7, o 9.

9.9.3 Interfaccia utente testuale

Per curiosità puoi andare alle slide 82 e vedere anche la regola turn.

9.10 Unicità di Prolog

- **Invertibilità dei predicati:** un singolo predicato implementa molte funzioni, questo è dovuto al fatto che le variabili logiche sono in IN/OUT/IN-OUT a seconda dei parametri attuali.
- **Programmazione nondeterministica:** con ricerca automatica delle soluzioni basato sul backtracking (cioè il meccanismo di visita dei search tree).
- Esistono poi molte altre funzionalità come: strutture dati parziali (permettono append e creazione dizionari in tempo costante), meta-predicati / reflection, aggregati (setof), forall...