Dispense del Corso di

Linguaggi di Programmazione

20 Settembre 2006

Le parti contrassegnate con ♠ sono approfondimenti.

Indice

Introduzione **2**

1 Definizione e Implementazione dei L. di P. **8**

1.1 Stringhe e linguaggi . . . 9
1.2 Grammatiche context-free . . . 10
1.3 Alberi di derivazione . . . 11
1.4 Ambiguità . . . 13
1.5 Algebre sintattiche e sintassi astratta . . . 15
1.6 Grammatiche come definizioni induttive . . . 16
1.7 Analisi lessicale . . . 16
1.8 Rappresentazione interna . . . 19
1.9 Schema di parsing . . . 22
1.10 Vincoli contestuali . . . 26
1.11 Semantica statica di FL . . . 28
1.12 Type-checker per FL . . . 30
1.13 Semantica dinamica . . . 34
1.14 Interprete per FL . . . 36

2 Paradigma Object-Oriented **40**

2.1 Introduzione . . . 40
2.2 Oggetti e Classi . . . 42
2.2.1 Classi . . . 42
2.2.2 Uguaglianza tra oggetti e copia . . . 42
2.2.3 Ricorsione tra classi, confronto tra oggetti e puntatori . . . 44
2.2.4 Uso di `this` . . . 46
2.2.5 Campi e metodi di classe . . . 47
2.2.6 Un algoritmo "a oggetti" . . . 48
2.2.7 Inizializzazione e distruzione di oggetti . . . 49
2.3 Inheritance e Sottotipo . . . 49
2.3.1 Classi eredi . . . 49
2.3.2 Overriding . . . 50
2.3.3 Inheritance e programmazione "per casi" . . . 52
2.3.4 Esempi di classi wrapper . . . 57
2.3.5 Inheritance e visibilità . . . 58
2.3.6 Interfacce e classi astratte . . . 58
2.3.7 Varianti della nozione di inheritance . . . 59
2.4 Il linguaggio Java . . . 60
2.4.1 Generalità . . . 60
2.4.2 Blocchi di inizializzazione statici . . . 62
2.4.3 Uso di `super` . . . 63
2.4.4 Costruttori . . . 64
2.4.5 Hiding di campi . . . 65
2.4.6 Risoluzione dell'overloading . . . 66
2.4.7 Array . . . 68
2.4.8 Casting . . . 69
2.4.9 Eccezioni . . . 70
2.4.10 Package . . . 74
2.4.11 Visibilità e modificatori . . . 75
2.4.12 Cenni alle classi predefinite . . . 76
2.4.13 Riassunto, caratteristiche non trattate e confronto con C,C++ . . . 78

3 Paradigma Funzionale **79**
3.1 Concetti base . . . 79
3.2 Simulazione di algoritmi imperativi . . . 85
3.3 Liste . . . 86
3.4 Parametri di accumulazione . . . 87
3.5 Funzioni higher-order su liste . . . 88
3.6 Tipi user-defined . . . 91
3.7 Tipi Caml e inferenza di tipo . . . 93
3.8 Caratteristiche non trattate . . . 93

A Appendice Tecnica **94**
A.1 Nozioni Base . . . 94
A.2 Algebre Eterogenee . . . 95

Introduzione

In questo capitolo si descrivono gli obiettivi del corso, si analizzano le caratteristiche dei linguaggi di programmazione (L. di P. da qui in avanti) dal punto di vista dello sviluppo del software, si fornisce una breve panoramica storica dei L. di P. e una rapida presentazione dei diversi paradigmi di programmazione.

Obiettivo del corso Lo scopo del corso è fornire allo studente le nozioni fondamentali relative ai linguaggi di programmazione e un buon livello di conoscenza di diversi paradigmi di programmazione (imperativo, funzionale, object-oriented, logico), dal punto di vista sia concettuale che operativo.

Caratteristiche dei L. di P. nel quadro dello sviluppo del software (Questa parte è essenzialmente tratta da [3] e [5]).
I linguaggi di programmazione "ad alto livello" sono stati introdotti come strumento "intermedio" che permette all'uomo di comunicare con la macchina. All'inizio, esistevano solo i linguaggi macchina (in cui le istruzioni sono sequenze di 0 e 1) e i linguaggi simbolici (in cui le sequenze binarie sono sostituite da simboli mnemonici). Negli anni '50, con il FORTRAN, sono nati i linguaggi ad alto livello, il cui aspetto più importante è l'indipendenza dalla macchina: infatti un programma in un linguaggio ad alto livello è definito in maniera autonoma, e può essere *implementato* su diversi sistemi, per esempio tramite uno strumento detto *compilatore* che lo traduce nel linguaggio macchina del sistema sottostante. Quindi questi linguaggi sono detti ad alto livello in contrapposizione ai linguaggi assembler o simbolici; tuttavia, oggi in genere si tralascia questa precisazione e si parla semplicemente di linguaggi di programmazione. I L. di P. sono di solito anche dotati di librerie e di test di consistenza (per esempio type checking) che trovano alcuni tipi di errori.
Un linguaggio di programmazione è uno strumento utilizzato per produrre software. Quindi uno studio dei L. di P. deve prendere in considerazione questo contesto.
Il ciclo di vita del software nel "modello a cascata" tradizionale consta delle seguenti fasi:

Analisi e specifica dei requisiti Insieme al cliente, si cerca di stabilire *cosa* dovrà fare il sistema (non solo aspetti funzionali, ma anche studio dei costi e della performance, interfaccia utente, etc.). Questa fase è spesso la più difficile.

Progetto (design) e specifica del sistema Si descrive come dovrà essere fatto il sistema a un livello più astratto rispetto al L. di P. Questa fase rappresenta il *come* dal punto di vista del livello precedente, il *cosa* rispetto al successivo.

Implementazione È la produzione del codice nel L.di.P. scelto. Rappresenta il *come* rispetto alla fase precedente.

Verifica e validazione Si cerca di provare la correttezza del programma attraverso testing e/o metodi formali.

Manutenzione(maintenance) Questa fase riguarda tutto ciò che viene dopo: scoperta e correzione di malfunzionamenti, aggiunta di funzionalità, miglioramenti, cambiamenti nell'ambiente di esecuzione.

In realtà questo modello è oggi obsoleto perchè le varie fasi tendono a essere ripetute più volte prima di arrivare a un prodotto completo.
Ogni fase nello sviluppo di software può essere facilitata da strumenti automatici, detti *ambienti di sviluppo*. Attualmente quella per cui vi sono più strumenti è quella di implementazione, con text editor, compilatori, type-checker, linker, librerie con browser, debugger e così via. In uno scenario ideale ogni fase dovrebbe essere "computer aided". Attualmente questo è solo approssimato da strumenti detti CASE (computer aided software engineering).

I linguaggi di programmazione intervengono ovviamente nella fase di implementazione e, quelli più moderni, anche nella fase di design.
Ci sono varie metodologie di design (per esempio, design procedurale o object-oriented). Un linguaggio di programmazione può essere influenzato dal metodo di design. Per esempio, i linguaggi più vecchi, come il Fortran, non facilitavano lo sviluppo di algoritmi in modo top-down (cioè decomponendo via via un problema in sottoproblemi), mentre il Pascal è stato progettato proprio per facilitare la cosiddetta *programmazione strutturata* ed è nato insieme a questa. In Pascal e in C, per contro, non è direttamente possibile una strutturazione del programma in moduli che corrispondano a tipi di dato, che è invece possibile nei linguaggi object-oriented come Java.
Un linguaggio di programmazione può anche essere influenzato, d'altro lato, dall'architettura. Per esempio, il paradigma imperativo corrisponde all'architettura di Von Neumann, basata sull'idea di una memoria che contiene dati e istruzioni, un'unità di controllo e un'unità di input/output: condivide infatti con essa almeno due cose, l'esecuzione sequenziale delle istruzioni e la memorizzazione di valori modificabili; è stata storicamente molto importante la critica di Backus a questo modello nel 1978.
I requisiti di un prodotto software che dipendono anche dal linguaggio di programmazione sono i seguenti.

Affidabilità (reliability) Consiste nel cercare di minimizzare gli errori di programmazione. Comprende varie caratteristiche anche in conflitto tra loro.

- "Scrivibilità", cioè possibilità di risolvere un problema nel modo più "naturale" possibile, senza essere distratti da problemi accessori legati a dettagli del linguaggio. È difficile da quantificare, ma è chiaro per esempio che un linguaggio ad alto livello è più "scrivibile" di un linguaggio assembler.
- Leggibilità. È la stessa cosa ma dal punto di vista di chi legge il programma. Dal punto di vista della reliability entrambi i requisiti sono importanti per evitare o diagnosticare gli errori.
- Semplicità (minimizzare le caratteristiche del linguaggio). Ossia: pochi concetti ma chiari. È stata per esempio una delle caratteristiche ricercate *inizialmente* nel design di Java . In linea di principio, in un linguaggio semplice il programmatore dovrebbe avere un'unica via naturale per risolvere un determinato problema. Ovviamente presenta degli svantaggi, perchè alcune caratteristiche potrebbero non essere fornite direttamente dal linguaggio.
- Safety, cioè il linguaggio non dovrebbe offrire caratteristiche "pericolose"; per esempio, `goto` o puntatori.
- Robustezza, cioè la capacità del programma di non reagire catastroficamente in caso di imprevisti, per esempio attraverso strumenti per il trattamento degli errori, come le eccezioni.

Modificabilità (maintainability) Ossia il facilitare le modifiche ai programmi.

- Fattorizzazione, cioè la possibilità di modellare una caratteristica che costituisce un'unità logica in un unico segmento di codice: per esempio uso di nomi di costanti, procedure o moduli.
- Localizzazione, cioè l'effetto di cambiamenti piccoli è concentrato in una piccola porzione del programma.

Sono due concetti strettamente collegati.

Efficienza Questo requisito era inizialmente molto importante, oggi lo è ancora, ma si tende a considerare oltre all'efficienza di spazio e tempo anche parametri come il costo di produrre e mantenere il software, la portabilità, la riusabilità e così via.

Paradigmi di programmazione Il concetto di paradigma di programmazione non è facile da definire precisamente. In modo informale, possiamo dire che vi sono famiglie di linguaggi che a grandi linee seguono lo stesso modello di esecuzione. Un tentativo di descrizione sommaria dei principali paradigmi di programmazione è dato dalla tabella seguente.

Paradigma	**Modello**	**Un programma è ...**	**Esempi**
Imperativo	Stato = astrazione della memoria	[eseguire] un comando	Pascal, C, Ada
Funzionale	Funzioni (definizione e applicazione)	[valutare] un'espressione	Lisp, ML
Logico	Predicati e deduzione logica	[soddisfare] un goal	Prolog
Ad oggetti	universo di oggetti	[mandare un messaggio a] un oggetto	Smalltalk, Eiffel, Java

Vediamo un semplicissimo esempio che illustra come lo stesso problema può essere risolto in maniera molto diversa utilizzando i vari paradigmi. Il problema che consideriamo è quello di calcolare la somma degli elementi di una lista di interi.
Ecco una possibile versione in C (assumendo che la variabile `l` denoti la lista, che supponiamo implementata con un tipo puntatore a una struttura con due campi `head` e `tail`):

```
int sum = 0;
list aux = l;
while (aux != NULL) {
  sum += aux->head;
  aux = aux->tail;
}
```

In un programma imperativo, l'effetto di un programma è essenzialmente quello di eseguire, a partire da un certo stato iniziale, una sequenza di trasformazioni di stato attraverso l'esecuzione di *comandi*, fino ad arrivare a uno stato finale. Uno stato è un'astrazione della memoria della macchina, cioè una funzione[1] da celle di memoria (locazioni) in valori. Formalmente $State = Loc \to Val$. Quindi un programma può essere interpretato come una funzione $p\colon State \to State$.
Nel paradigma funzionale invece il punto di vista è completamente diverso. Un programma è semplicemente una collezione di definizioni di funzioni. Una soluzione del problema precedente scritta in un linguaggio funzionale (per esempio Caml, il linguaggio della famigliai ML utilizzato nel corso) è la seguente.

```
let rec sumlist = function
  [] -> 0
  | a :: l -> a + sumlist l;;
```

L'idea, come si vede, è quella di definire induttivamente, per casi (il simbolo `|` denota infatti alternativa), una funzione matematica. Data la definizione, sarà possibile valutare espressioni del tipo `sumlist [1;2;3;4]`. Un programma non è in questo caso interpretato come una funzione in $State \to State$, ma (nell'esempio) come una funzione $f: \mathbb{Z}^\star \to \mathbb{Z}$, dove $\mathbb{Z}^\star$ è l'insieme delle sequenze finite di numeri interi.
Nel caso del paradigma logico, un programma è una collezione di definizioni di predicati tramite *clausole*. Sempre in riferimento all'esempio:

```
sumlist([],0).
sumlist([0|Ns],Sum) :- sumlist(Ns,Sum).
sumlist([s(N) | Ns],s(Sum)) :- sumlist([N|Ns],Sum).
```

La prima clausola (senza parte destra) è (nella terminologia della programmazione logica) un *fatto* e asserisce una verità nota: nel nostro caso che la somma degli elementi della lista vuota è zero. La seconda e terza clausola si interpretano come l'asserzione che, per ogni possibile istanziazione delle variabili, se la parte destra è vera anche la parte sinistra deve essere vera[2]. Data tale definizione, sarà possibile valutare un *goal* del tipo `?-sumlist([0,s(0),s(s(s(0)))],S)`, cioè trovare se ci sono valori di `S` che rendono vera la formula (*soluzioni* del goal, in questo caso una sola).
Il paradigma object-oriented è caratterizzato da diverse caratteristiche ortogonali. Dal punto di vista computazionale, cioè per quel che riguarda il modello di esecuzione (parleremo in questo caso più propriamente di paradigma *a oggetti*), l'idea è simile a quella del paradigma imperativo. Anche in questo caso infatti l'esecuzione di un programma effettua, a partire da uno stato iniziale, una serie di trasformazioni di un sistema fino ad arrivare in uno stato finale. Tuttavia in questo caso, piuttosto che corrispondere alla memoria fisica dell'elaboratore, il "sistema" che si trasforma può essere visto informalmente come un universo di "oggetti", e la trasformazione ha inizio con l'invio di un "messaggio" a uno di tali oggetti. L'oggetto ricevente potrà a sua volta inviare un messaggio a un altro oggetto e così via. Una similitudine può essere fatta con ciò che accade quando si utilizza un'interfaccia grafica, con finestre, bottoni, menu etc.; a ogni azione su un determinato oggetto corrisponde una reazione che può coinvolgerne altri.
Inoltre, nel paradigma object-oriented un'altra idea chiave è quella di definire tante *classi* (schemi) di oggetti, una per ogni diverso tipo di oggetto che vogliamo manipolare. Relativamente all'esempio, una lista può essere vuota oppure no, e a seconda di tale fatto reagirà diversamente alla richiesta (messaggio) di calcolare la somma dei propri elementi. Se mandiamo tale messaggio a una lista vuota, questa risponderà 0; se lo stesso messaggio viene inviato a una lista non vuota, la risposta sarà ottenuta sommando il primo elemento alla risposta ottenuta rimandando il messaggio alla lista "resto". Vediamo una versione Java che corrisponde a quest'idea.

[1] Per le definizioni e notazioni relative alle funzioni si veda l'Appendice, Sez.A.1.

[2] Il linguaggio dei programmi logici "puri" utilizzato nell'esempio non ha tipi primitivi, quindi i numeri naturali sono rappresentati come termini costruiti con la costante `0` e l'operatore unario `s` (successore).

```
abstract class List {
  abstract int sum ();
}

class EmptyList extends List {
  int sum () { return 0;}
}

class NonEmptyList extends List {
  int head;
  List tail;
  int sum () { return head + tail.sum(); }
}
```

Quelli di cui abbiamo parlato finora sono paradigmi computazionali, cioè relativi al modello di esecuzione del programma. Ci sono anche paradigmi *organizzativi*, relativi cioè non a come vengono impostati i singoli algoritmi ma a come viene strutturato un programma complesso. Vediamone alcuni.

Procedurale I moduli corrispondono a unità algoritmiche (procedure).

Abstract Data Type (ADT) I moduli corrispondono a tipi di dato, cioè a unità che definiscono certi tipi di valori e le operazioni per manipolarli. La nozione informale di tipo di dato è formalizzata dalla nozione di *algebra (parziale) eterogenea*, vedi Def.A.8.

Module-based Più generale; i moduli sono collezioni di componenti eterogenee (tipi, procedure, eccezioni, variabili, etc)

Generic Programming I moduli possono essere *parametrici* (per esempio si pensi a un modulo `LIST(T)` che implementa liste di elementi di un tipo generico `T`).

Object-Oriented I moduli sono le classi, nel senso visto sopra, e nuovi moduli possono essere costruiti attraverso modifiche parziali di moduli già esistenti attraverso il meccanismo dell'*inheritance* (vedi Sez.2.3).

Ogni linguaggio può avere il suo paradigma computazionale e il suo paradigma organizzativo, ma possono anche coesistere più paradigmi. Per esempio Smalltalk è un linguaggio a paradigma unico object-oriented; ML è un linguaggio a paradigma computazionale funzionale e paradigma organizzativo ADT; C++ è un linguaggio a paradigma misto (imperativo e object-oriented insieme).
Esistono diversi paradigmi anche per quel che riguarda le metodologie di design (per esempio, il paradigma object-oriented ha avuto molto successo anche da questo punto di vista); chiaramente se il paradigma di progettazione coincide con quello del linguaggio che si vuole poi utilizzare sarà più facile effettuare il passaggio dall'uno all'altro.

Breve panoramica storica I primi tentativi di definizione di linguaggi ad alto livello sono negli anni '50. In particolare Fortran nel 1957 permette per la prima volta di scrivere formule usando i simboli matematici tradizionali, come $+$ e $*$ (infatti Fortran viene da Formula Translator). Dato che l'hardware era molto costoso, i requisiti di efficienza erano il vincolo principale nel design. I linguaggi più significativi di questa fase sono FORTRAN, ALGOL 60 (progenitore di Pascal e C) e COBOL, i primi due per problemi numerici, il terzo orientato a problemi di processing di dati in ambito aziendale. Questi linguaggi sono stati molto importanti; FORTRAN e COBOL sono ancora tra i più largamente usati (anche per la riluttanza degli utenti a cambiare, e per il fatto che si sono nel frattempo evoluti).
Intorno agli anni '60 ci sono tentativi di definire linguaggi con modelli di calcolo basati su principi matematici, piuttosto che sull'efficienza dell'implementazione. Per esempio LISP (1958) è basato sulla teoria delle funzioni ricorsive e sul lambda calcolo, e ha dato l'avvio al paradigma funzionale. LISP ha un unico tipo di dato, le liste (Lisp = List Processor); ha avuto un successo notevole nel campo dell'intelligenza artificiale. Alla fine degli anni '60 nascono Algol 68, SIMULA 67 e Pascal. Algol 68 (successore di Algol 60) è basato sull'*ortogonalità* delle caratteristiche. È il primo linguaggio fornito di specifica formale completa. Troppo "puro", non ha avuto applicazione industriale. SIMULA 67 (Nygaard e Dahl) introduce la nozione di *classe* ed è quindi considerato il progenitore del paradigma object-oriented. Pascal fu progettato soprattutto come strumento didattico. È semplice e consente la programmazione strutturata. BASIC ha una sintassi semplicissima con poche strutture di controllo e tipi di dato, ed è facile da implementare in modo efficiente; questo lo ha reso molto popolare.
Negli anni '70 si impongono i requisiti di reliability e maintainability. Si considerano concetti come ADT, modularità, type checking statico, eccezioni, concorrenza. Nascono C (Ritchie 1972) e Modula-2. C ha molto successo anche per il suo legame con il sistema operativo Unix, e per il suo potere espressivo ed efficienza per la programmazione di sistema. PROLOG (Colmerauer

e Roussel 1972) è il capostipite del paradigma *logico*. Acquisterà popolarità quando viene lanciato il cosiddetto Programma della Quinta Generazione dal governo giapponese e la programmazione logica è scelta come base per la nuova generazione di macchine.
Negli anni 80 nasce ML (Milner 1984), e il Dipartimento della Difesa degli U.S.A. sponsorizza il design di un nuovo linguaggio, che sarà Ada. Smalltalk-80 è il capostipite del paradigma object-oriented modernamente inteso. Non è solo un linguaggio, ma un ambiente di programmazione. Ne sono derivati Eiffel, C++ e Java.
Attualmente: nel campo dei sistemi informativi si stanno diffondendo generatori di applicazioni e linguaggi visuali. Come linguaggio general purpose (cioè adatto a ogni tipo di applicazioni) ha avuto molto successo C++, e recentemente Java.

Programmazione "in piccolo" e "in grande" Un programma può essere composto da migliaia, anche milioni di linee di codice. Evidentemente non può essere progettato, scritto, testato come un tutt'uno. Quindi il linguaggio di programmazione deve anche fornire, oltre che un modo di scrivere singole computazioni, modi per organizzarle in modo che ogni frammento possa essere considerato separatamente.
"The art of programming is the art of organising complexity" (E. W. Dijkstra).
Useremo il termine generico "modulo" per un frammento di programma che costituisce un'unità logica separata e indipendente. Introdurre l'idea di modulo porta in modo naturale all'introduzione di altri concetti: specifica, implementazione, correttezza, incapsulazione.

Specifica Sorge la necessità di fornire all'utente una descrizione, attraverso un qualche linguaggio formale o informale, di *che cosa fa* il modulo, e non di *come* lo fa. Questa è una distinzione che riteniamo ovvia quando utilizziamo uno strumento fisico (per esempio quando usiamo il videoregistratore non ci interessa sapere come è realizzato internamente il suo funzionamento); lo stesso atteggiamento va applicato a uno strumento software.

Implementazione È il *come*, cioè il modo in cui il modulo realizza le funzionalità descritte dalla specifica.

Correttezza Un modulo è corretto rispetto alla sua specifica se l'implementazione data soddisfa i requisiti descritti nella specifica.

Incapsulazione (information hiding) Si intende con questo termine la possibilità offerta dal linguaggio di nascondere effettivamente all'utente l'implementazione, permettendogli di manipolare il modulo solo attraverso le funzionalità descritte nella specifica.

I vantaggi della modularità sono facilmente intuibili, e relativi a vari aspetti. Ne elenchiamo alcuni.

Leggibilità È più facile capire cosa fa una piccola porzione di codice che un intero programma.

Debugging Il testing di un programma di grandi dimensioni non viene effettuato globalmente, ma ogni singolo modulo viene testato separatamente. Nel momento in cui un modulo ha passato la fase di testing possiamo con buona approssimazione considerarlo corretto rispetto alla sua specifica e quindi assumere che errori futuri provengano da altri moduli.

Prove di correttezza Vale lo stesso discorso del punto precedente.

Sviluppo in parallelo L'attività di produrre software solitamente viene svolta in équipe, per cui moduli diversi possono essere sviluppati da sottogruppi diversi; se un modulo A usa un modulo B gli sviluppatori di A semplicemente assumeranno che B implementi correttamente la specifica concordata.

Riusabilità Se il sistema è suddiviso in moduli è molto probabile che vi sia qualche modulo che potrà essere riutilizzato in progetti futuri. Un discorso analogo vale per la modificabilità.

Compilazione separata Infine, una caratteristica auspicabile è che ogni modulo sia anche un'unità di compilazione, in modo tale da potere compilare l'intero progetto in passi successivi. È auspicabile anche che, se si decide di cambiare un modulo, questo non richieda la ricompilazione di altri moduli ma solo del modulo cambiato.

Implementazione dei L. di P. Come abbiamo visto il linguaggio di programmazione è ad alto livello, quindi indipendente dalla macchina. Tuttavia, alla fine deve poter "essere eseguito" su una particolare macchina. Ci sono due approcci fondamentali al problema.
Nel primo, il linguaggio è tradotto nel linguaggio della macchina sottostante attraverso uno strumento detto *compilatore* (quindi è il linguaggio a essere portato al livello della macchina). Fondamentale nei linguaggi compilati è la netta distinzione tra il momento della compilazione (*compile-time*) e quello dell'esecuzione del programma in linguaggio macchina ottenuto (*run-time*).

Nel secondo viene simulata una macchina ad alto livello che può eseguire direttamente i programmi nel linguaggio, attraverso uno strumento detto *interprete*; è quindi la macchina a essere portata al livello del linguaggio. Un interprete prende quindi in input insieme un programma e il suo input, e produce in uscita l'output. Si parla anche di *macchina astratta*, nel senso appunto che è come se si avesse una macchina in grado di eseguire direttamente un programma nel linguaggio ad alto livello.
Si dicono in generale proprietà *statiche* dei programmi quelle proprietà che possono essere rilevate dall'esame del testo; *dinamiche* quelle che possono essere rilevate solo eseguendo il programma. I linguaggi compilati sono più orientati verso le proprietà statiche, poichè tutte le decisioni del compilatore sono prese esaminando il codice sorgente al momento della traduzione. I linguaggi interpretati sono più orientati verso le proprietà dinamiche. Inoltre:

- La compilazione può essere più efficiente. Infatti, un interprete può dover riesaminare molte volte lo stesso frammento di codice, mentre il compilatore lo traduce una volta per tutte.
- L'interpretazione può essere più flessibile. I programmi possono essere cambiati interattivamente; è possibile scrivere pezzi di codice e controllare subito come si comporta il sistema.

In pratica si possono usare tecniche miste. Per esempio per Java esiste una prima fase compilativa in cui viene prodotto un codice intermedio che poi viene eseguito da una macchina astratta. Inoltre, nelle moderne macchine virtuali (sia la Java Virtual Machine, sia, per esempio, Microsoft .NET) il codice intermedio viene compilato "al volo" da una componente nota come *Just In Time Compiler* (JIT-compiler).

Elementi di un L. di P. Un L. di P. è l'analogo di un linguaggio naturale, con la sola differenza che viene utilizzato per comunicare con la macchina e non con altri essere umani. Il L. di P. sarà quindi il più possibile formalizzato.
Un linguaggio naturale è caratterizzato dalle seguenti componenti.

- L'insieme delle *parole* o *simboli* che possono essere utilizzate per formare frasi del linguaggio (il "dizionario" o "alfabeto"); la terminologia duplice deriva dal fatto che se consideriamo come linguaggio ad esempio quello formato dalle frasi italiane l'insieme delle parole utilizzabili è dato dal dizionario; possiamo però considerare a sua volta come un linguaggio quello delle parole italiane, e allora le componenti utilizzabili per formare le parole italiane sono le lettere dell'alfabeto italiano.
- Noto il vocabolario, non è detto che tutte le frasi composte con parole del vocabolario siano corrette; a questo punto infatti intervengono le regole grammaticali, cioè la *sintassi*.
- Sarà poi necessario conoscere la *semantica* cioè il significato delle frasi del linguaggio.
- Infine, per utilizzare correttamente un linguaggio occorre conoscerne anche la *pragmatica* cioè *come* usare il linguaggio (per esempio quali frasi è opportuno usare a seconda del contesto).

Naturalmente i linguaggi naturali non sono completamente formali; in altri termini, dizionario, sintassi e semantica non sono fissati in maniera univoca, ma vi è sempre un certo margine di vaghezza o ambiguità: un linguaggio naturale è uno strumento molto flessibile e anche in continua evoluzione. Per esempio, anche parole inesistenti possono essere usate in certi contesti, sarebbe estremamente difficile formalizzare le regole grammaticali, e la semantica è spesso ambigua o comunque non formalizzata.
Per i L. di P. invece alfabeto, sintassi e semantica devono essere definiti in maniera inequivocabile. In realtà, mentre per la sintassi vi è uno stile di definizione formale (la BNF, vedi Sez.1.2) comunemente seguito, per quel che riguarda la semantica è difficile dare definizioni di ciò che fa un programma che uniscano la leggibilità e il rigore. Infatti, una definizione in linguaggio naturale si presta ad ambiguità, mentre una definizione di tipo matematico è difficilmente comprensibile ai non esperti. Lo stato dell'arte è che esistono tre principali tipi di descrizione di un L. di P., di cui solo i primi due forniti in genere per tutti i linguaggi:

- I *tutorial*, che cercano di presentare efficacemente le varie caratteristiche del linguaggio anche attraverso esempi.
- I *reference manual*, tipicamente organizzati sulla sintassi, che contengono la sintassi formale data tramite BNF più una descrizione a parole il più possibile precisa e non ambigua della semantica.
- le *definizioni formali*, in genere indirizzate agli specialisti.

1 Definizione e Implementazione dei L. di P.

Come accennato alla fine dell'Introduzione, per definire un L. di P. è necessario darne:

- *la* sintassi *cioè una specifica dell'insieme dei programmi,*
- *la* semantica *cioè una descrizione dell'effetto dell'esecuzione dei programmi.*

In realtà, per ragioni che saranno spiegate all'inizio della Sez.1.10, la specifica dell'insieme dei programmi del linguaggio viene usualmente data in due fasi: prima si definisce, attraverso una grammatica libera da contesto*, un soprainsieme dei programmi del linguaggio, poi all'interno di questo insieme si definiscono alcuni* vincoli contestuali *che solo i programmi* corretti *o* ben formati *verificano. Avendo introdotto questa distinzione, si intende più propriamente come* sintassi *del linguaggio la descrizione fornita nella prima fase (ossia le regole della grammatica), mentre la descrizione dei vincoli contestuali fornita nella seconda fase viene chiamata* semantica statica *in quanto si descrivono proprietà dei programmi osservabili staticamente, cioè prima dell'esecuzione; la descrizione dell'effetto dell'esecuzione si chiama allora più propriamente* semantica dinamica*.*
Nell'implementazione di un linguaggio, è possibile distinguere diverse componenti che corrispondono ai diversi elementi illustrati sopra. Il parser *o* analizzatore sintattico *riceve in input una stringa di simboli e stabilisce se questa è in accordo con la sintassi, cioè verifica le regole della grammatica data. In caso positivo, il parser provvede anche a convertire la stringa di simboli in una rappresentazione interna più comoda per le fasi successive. Il* type-checker *o* analizzatore statico *riceve in input un programma e stabilisce se questo è un programma corretto, cioè verifica i vincoli contestuali dati. Infine (consideriamo qui un'implementazione basata su interprete) l'*interprete *propriamente detto riceve in input un programma (la cui correttezza è stata provata nella fase precedente) e ne simula l'esecuzione.*

1.1 Stringhe e linguaggi

Possiamo vedere un programma in un L. di P. come una stringa di simboli costruita in modo da soddisfare certe regole (la *sintassi* del linguaggio). Per illustrare il modo in cui viene in genere descritta formalmente la sintassi dei L. di P. occorre quindi anzitutto precisare cosa si intende per simbolo, stringa, linguaggio.

Def. 1.1 [Alfabeto] Un *alfabeto* è un insieme finito non vuoto di oggetti detti *simboli*.

Def. 1.2 [Stringa] Una *stringa* u su un alfabeto A è una funzione (totale) da $[1, n]$ in A, per qualche $n \in \mathbb{N}$; n si dice *lunghezza* di u, e si indica con $|u|$.

Useremo u, v, w per indicare generiche stringhe.
L'unica stringa u tale che $|u| = 0$ si chiama *stringa vuota* e si indica con Λ; l'insieme delle stringhe su A si indica con $A^\star$ e l'insieme delle stringhe non vuote su A si indica con A^+. Indicando con A^n, per $n \in \mathbb{N}$, l'insieme delle funzioni da $[1, n]$ in A, si ha dunque per definizione che $A^+ = \cup_{n>0} A^n$.

Def. 1.3 [Linguaggio] Un *linguaggio* su un alfabeto A è un insieme di stringhe su A.

Si noti che A^+ è sempre un insieme infinito; più precisamente è un insieme numerabile.
Siamo tutti abituati a manipolare stringhe di simboli, senza chiederci *cos'è* una stringa. In genere, scriviamo le stringhe utilizzando la rappresentazione *per giustapposizione* cioè semplicemente scrivendo i simboli uno dopo l'altro da sinistra a destra. Per esempio se $A_{it} = \{a, b, c, d, \ldots, z\}$ è l'alfabeto italiano, una stringa è la parola italiana *amica*. Tuttavia, l'unico modo *rigoroso* di definire tale stringa è di dire che (in accordo con la definizione data sopra) essa è la funzione $v\colon [1, 5] \to A_{it}$ definita da:

$$v(i) = \begin{cases} a & \text{se } i = 1 \\ m & \text{se } i = 2 \\ i & \text{se } i = 3 \\ c & \text{se } i = 4 \\ a & \text{se } i = 5 \end{cases}$$

Infatti, la rappresentazione per giustapposizione (oltre che arbitraria e legata a ragioni culturali: altri popoli utilizzano per esempio la rappresentazione da destra a sinistra), può risultare ambigua per certi alfabeti. Ad esempio con questa rappresentazione la stringa sull'alfabeto D_{it} costituito dalle parole italiane (quindi la frase italiana) formata dalle due parole *arco* e *baleno* è indistinguibile da quella formata dall'unica parola *arcobaleno*. In casi di questo genere si ricorre tipicamente all'uso di *separatori*, cioè simboli che *non* fanno parte dell'alfabeto ma servono solo come convenzione per delimitare i simboli dell'alfabeto, come il blank.
Un'altra rappresentazione spesso usata è quella a n-uple, cioè un elemento $w \in A^n$ definito da

$$w(i) = \begin{cases} a_1 & \text{se } i = 1 \\ a_2 & \text{se } i = 2 \\ \vdots & \\ a_n & \text{se } i = n \end{cases}$$

è scritto come $(a_1, \ldots, a_n)$. La stringa precedente v in questo modo viene scritta (a, m, i, c, a). Anche questa rappresentazione può creare ambiguità, nel caso in cui i separatori parentesi aperta, chiusa e virgola facciano parte dell'alfabeto.
L'unico modo rigoroso e indipendente dalla rappresentazione di definire le stringhe su un alfabeto è quello dato nella Def.1.2, ed è conveniente richiamarsi a questa definizione tutte le volte che sorge qualche problema di ambiguità. Nella pratica, si può invece utilizzare in ogni caso concreto una rappresentazione non ambigua.

Def. 1.4 [Concatenazione di stringhe e linguaggi] Se v e w sono due stringhe su un alfabeto A di lunghezza n e m rispettivamente, allora $v \cdot w$ è la stringa su A, di lunghezza $n + m$, definita da

$$(v \cdot w)(k) = \begin{cases} v(k) & \text{se } 1 \leq k \leq n \\ w(k-n) & \text{se } n < k \leq n+m \end{cases}$$

Inoltre $v^0 = \Lambda$, $v^{n+1} = v \cdot v^n$.
Se X, Y sono insiemi di stringhe su A, $X \cdot Y = \{x \cdot y \mid x \in X, y \in Y\}$.
Utilizzando l'ultima definizione, si può dare la seguente definizione equivalente induttiva di A^n, per A alfabeto e $n \in \mathbb{N}$:

$X^0 = \{\Lambda\}$
$X^{n+1} = X \cdot X^n$.

Un altro esempio "classico" di stringhe è dato dai numeri (interi positivi) nella rappresentazione in base 10, che sono esattamente le stringhe su *Digit* $= \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}$, se si accetta `0` in posizione non significativa (es: `002`); altrimenti sono un sottoinsieme proprio *Num* di queste stringhe, descritto da

$$\textit{Num} = \{v \mid v(1) \neq \texttt{0} \text{ oppure } |v| = 1\}.$$

Chiameremo nel seguito *numerali* gli elementi di *Num*.
Analogamente le espressioni costruite a partire dai numerali con le operazioni + e $\star$ e le parentesi tonde `(` e `)` sono un linguaggio sull'alfabeto $\textit{Digit} \cup \{\texttt{+}, \star, \texttt{(}, \texttt{)}\}$. Alternativamente possono essere viste come un linguaggio su $\textit{Num} \cup \{\texttt{+}, \star, \texttt{(}, \texttt{)}\}$, ma in questo caso l'alfabeto non è più finito (si veda la Sez.1.9 per ulteriori considerazioni su queste due possibili alternative).
Infine, si chiama anche, con abuso di terminologia, *linguaggio* su un alfabeto A, una famiglia (Def.A.6) di linguaggi su A indiciata su un insieme S di *tipi* o *sort*. Si considerino per esempio i due insiemi *Exp* delle espressioni intere (definito come visto prima) e *BExp* delle espressioni booleane, costruite con le costanti `true` e `false`, l'operatore = di uguaglianza tra espressioni intere e gli operatori `and` e `or`. Questi due insiemi costituiscono insieme un linguaggio L indiciato sui due tipi $\{\textit{Exp}, \textit{BExp}\}$ (per semplicità denotiamo nello stesso modo il tipo e l'insieme corrispondente).

1.2 Grammatiche context-free

Il modo usuale in cui si trova descritta formalmente la sintassi di un linguaggio è tramite una *grammatica libera da contesto*. Tale metodo deriva dagli studi di Noam Chomsky negli anni '50 sulle grammatiche per i linguaggi naturali. La Backus & Naur Form (BNF), o Backus Normal Form, è un particolare stile di presentazione di una grammatica libera da contesto, e fu introdotta alla fine degli anni cinquanta per descrivere la sintassi dell'Algol 60. L'aggettivo "libera da contesto" verrà chiarito in Sez.1.10.

Def. 1.5 [Grammatica] Una *grammatica libera da contesto*, nel seguito semplicemente *grammatica*, è una tripla (T, N, P) dove:

- T è un alfabeto di simboli, detti *terminali*,
- N è un alfabeto di simboli (diversi da quelli di T), detti *non terminali* o *categorie sintattiche* o *metasimboli*,
- P è un insieme finito di coppie dette *produzioni* (A, α) con $A \in N$ e $\alpha \in (T \cup N)^\star$ (cioè, A è un simbolo non terminale e α è una stringa formata da simboli terminali e non terminali). Nello stile BNF le produzioni sono scritte nella forma $A ::= \alpha$.

Useremo u, v, w per indicare generiche stringhe di terminali e α, β, γ per indicare generiche stringhe di terminali e non terminali. I terminali sono i simboli del linguaggio che si vuole definire tramite la grammatica , mentre i non terminali indicano i possibili "tipi" di oggetti sintattici (da qui il nome "categorie sintattiche"). Il nome "metasimboli" fa riferimento invece al fatto che si tratta di simboli utilizzati a livello meta, cioè per descrivere un linguaggio su un alfabeto di altri simboli.
In genere si usa una convenzione per distinguere terminali e non terminali (per esempio non terminali tra parentesi acute, o terminali tra apici). Nel seguito useremo la convenzione di scrivere i non terminali in *italico*, i terminali in `typewriter`.
Storicamente, si usava anche distinguere tra i non terminali un simbolo particolare detto *assioma* (intuitivamente, la categoria sintattica dei *programmi* nel linguaggio).
Per esempio $GExp$:

$$\begin{array}{lcl} \textit{Exp} & ::= & \texttt{(}\textit{Exp} \texttt{ + } \textit{Exp}\texttt{)} \mid \texttt{(}\textit{Exp} \star \textit{Exp}\texttt{)} \mid \textit{Num} \\ \textit{Num} & ::= & \texttt{0} \mid \texttt{1} \mid \ldots \end{array}$$

è una grammatica (dove per il momento abbiamo considerato per semplicità infinite produzioni, una per ogni numerale: vedremo in Sez.1.7 come ridurci a un insieme finito) che definisce le espressioni descritte precedentemente. L'alfabeto dei terminali è formato dagli elementi di *Num*, più i simboli `+`, `*`, `(`, `)`. Invece i simboli *Exp* e *Num* non fanno parte del linguaggio, ma servono a definirlo.
Nello stile BNF, si usa l'abbreviazione $A ::= \alpha \mid \beta \mid \gamma \ldots$ per indicare le produzioni $A ::= \alpha$, $A ::= \beta$, $A ::= \gamma \ldots$
Vediamo un altro esempio in cui vengono definite espressioni intere e booleane utilizzando due diversi non terminali.

$$\begin{array}{lll} \mathit{Exp} & ::= & \texttt{(}\mathit{Exp} + \mathit{Exp}\texttt{)} \mid \texttt{(}\mathit{Exp} \star \mathit{Exp}\texttt{)} \mid \mathit{Num} \\ \mathit{Num} & ::= & \texttt{0} \mid \texttt{1} \mid \ldots \\ \mathit{BExp} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{(}\mathit{BExp}\ \texttt{or}\ \mathit{BExp}\texttt{)} \mid \texttt{(}\mathit{BExp}\ \texttt{and}\ \mathit{BExp}\texttt{)} \mid \texttt{(}\mathit{Exp} = \mathit{Exp}\texttt{)} \end{array}$$

Per capire in che senso una grammatica definisca un linguaggio, introduciamo il concetto di derivazione.

Def. 1.6 [Derivazione in un passo] Sia data una grammatica $G = (T, N, P)$, e due stringhe $\beta, \gamma \in (T \cup N)^\star$. Diremo che *$\gamma$ è derivabile da β in un passo* se e solo se esistono delle stringhe $\alpha_1, \alpha_2 \in (T \cup N)^\star$, ed esiste in P una produzione $A ::= \alpha$, tali che $\beta = \alpha_1 A \alpha_2$ e $\gamma = \alpha_1 \alpha \alpha_2$. Diremo anche che $\beta \rightarrow \gamma$ è una *derivazione (in un passo)*.

È chiaro che $\rightarrow$ è una relazione (Def.A.1) su $(T \cup N)^\star$. Indicheremo con $\rightarrow^\star$ la sua chiusura riflessiva e transitiva (Def.A.3), e diremo che γ è derivabile da β se vale che $\beta \rightarrow^\star \gamma$. Diremo anche che $\beta \rightarrow^\star \gamma$ è una *derivazione*.
Il linguaggio generato da una grammatica G, rispetto a un suo non terminale A, è l'insieme delle stringhe *composte solo di terminali* che si possono derivare da A. Formalmente:

Def. 1.7 [Linguaggio generato da una grammatica] Sia data una grammatica $G = (T, N, P)$ e un non terminale $A \in N$. Il *linguaggio generato da G, rispetto ad A*, è l'insieme:

$$L_A(G) = \{w \in T^\star \mid A \rightarrow^\star w\}.$$

Con l'abuso di terminologia già menzionato, si parla del *linguaggio generato* da G anche intendendo la famiglia $\{L_A(G)\}_{A \in N}$.
Per semplicità, si usa spesso lo stesso nome (per esempio *Exp*) per denotare una categoria sintattica e il linguaggio generato corrispondente. Nel seguito adotteremo questa convenzione. Occorre però aver presente che sono due concetti distinti: il primo è semplicemente un simbolo, il secondo è un insieme di stringhe (di simboli terminali).
Consideriamo per esempio le seguenti grammatiche G_1, G_2, G_3, G_4.

$$\begin{array}{lll} S & ::= & \texttt{a}S \mid S\texttt{b} \mid c \\ S & ::= & \texttt{a}S\texttt{b} \mid \Lambda \\ S & ::= & \texttt{a}S\texttt{b} \mid \texttt{ab} \\ S & ::= & \texttt{a}S\texttt{a} \mid \texttt{b}S\texttt{b} \mid \Lambda \mid \texttt{a} \mid \texttt{b} \end{array}$$

I linguaggi generati da queste grammatiche sono rispettivamente:

$L(G_1) = \{\texttt{a}^n\texttt{cb}^m \mid n, m \geq 0\}$
$L(G_2) = \{\texttt{a}^n\texttt{b}^n \mid n \geq 0\}$
$L(G_3) = \{\texttt{a}^n\texttt{b}^n \mid n \geq 1\}$
$L(G_4) = \{u \mid u \in \{\texttt{a}, \texttt{b}\}^\star, u \text{ palindroma}\}$.

1.3 Alberi di derivazione

Data una stringa nel linguaggio generato da una grammatica, questa può essere in genere ottenuta tramite molte derivazioni diverse, che corrispondono a diverse possibili scelte del non terminale da espandere, cioè cui applicare di volta in volta una produzione. Consideriamo per esempio la stringa `(5+(4*2))` del linguaggio *Exp*. Questa stringa può essere ottenuta per esempio con le seguenti derivazioni (dove abbiamo sottolineato a ogni passo il non terminale cui viene applicata una produzione):

$\underline{\mathit{Exp}} \rightarrow \texttt{(}\underline{\mathit{Exp}} + \mathit{Exp}\texttt{)} \rightarrow \texttt{(}\underline{\mathit{Num}} + \mathit{Exp}\texttt{)} \rightarrow \texttt{(5 +}\ \underline{\mathit{Exp}}\texttt{)} \rightarrow \texttt{(5 + (}\mathit{Exp} \star \mathit{Exp}\texttt{))} \ldots$
$\underline{\mathit{Exp}} \rightarrow \texttt{(}\mathit{Exp} + \underline{\mathit{Exp}}\texttt{)} \rightarrow \texttt{(}\mathit{Exp} + \texttt{(}\underline{\mathit{Exp}} \star \mathit{Exp}\texttt{))} \rightarrow \texttt{(}\mathit{Exp} + \texttt{(}\underline{\mathit{Num}} \star \mathit{Exp}\texttt{))} \rightarrow \texttt{(}\mathit{Exp} + \texttt{(4} \star \mathit{Exp}\texttt{))} \ldots$

Un modo per astrarre rispetto a queste diverse derivazioni è quello di considerare gli *alberi di derivazione*. La nozione di albero di derivazione corrisponde a espandere "in parallelo" tutti i non terminali. Per esempio un albero di derivazione per la stringa `(5+(4*2))` è illustrato in Fig.1.

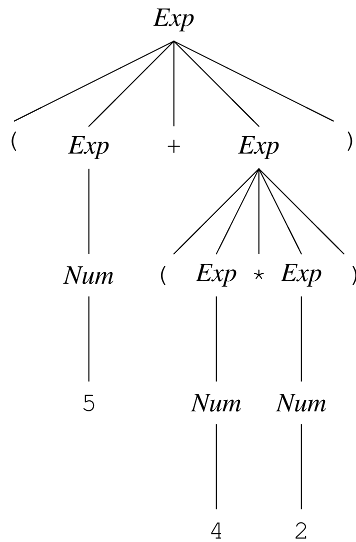


Figura 1: Un albero di derivazione

Def. 1.8 [Albero di derivazione] Data una grammatica $G = (T, N, P)$, un non terminale A ed una stringa di terminali w, un *albero di derivazione (in* G*) a partire da* A *per* w è un albero (ordinato) etichettato (Def.A.4) in $T \cup N$ tale che:

- la radice è etichettata con A;
- se un nodo è etichettato con un terminale, allora è una foglia;
- se un nodo è etichettato con un non terminale B e α è la stringa formata dalle etichette dei figli da sinistra a destra, allora $B ::= \alpha$ è una produzione in G;
- w è la stringa formata dalle etichette terminali delle foglie da sinistra a destra.

Si noti che è possibile avere anche foglie etichettate con non terminali, nel caso vi siano produzioni con parte destra Λ (una definizione alternativa è quella in cui Λ è ammessa come etichetta; in questo caso le foglie risultano essere esattemente i nodi etichettati con un terminale o con Λ).
Il seguente teorema stabilisce che il linguaggio generato da una grammatica può essere indifferentemente definito in termini di derivazioni o di alberi di derivazione.

Teorema 1.9 Data una grammatica $G = (T, N, P)$, un non terminale A ed una stringa di terminali w, w appartiene a $L_A(G)$ se e solo se esiste un albero di derivazione in G a partire da A per w.

Quindi, per provare che una stringa w appartiene al linguaggio $L_A(G)$, basta esibire una derivazione o un albero di derivazione per w a partire da A in G. Per provare invece che $w \notin L_A(G)$, occorre provare che *non esiste* un albero di derivazione per w a partire da A in G, e in genere questa non è una cosa banale. Nei casi semplici si può fare un'analisi per casi delle produzioni applicabili provando che si arriva sempre ad una situazione in cui non è più possibile ottenere w; oppure, si può provare per induzione che tutte le stringhe in $L_A(G)$ verificano una certa proprietà che w non soddisfa.
L'albero di derivazione mette in evidenza la "struttura" della stringa, ossia il modo in cui la stringa può essere decomposta. Ad esempio, l'albero di derivazione per `(5+(4*2))` mostra che tale espressione può essere ottenuta applicando un operatore binario, il `+`, a due sottoespressioni, la seconda delle quali è a sua volta ottenuta applicando l'operatore binario `*` a due sottoespressioni. L'albero fornisce quindi una guida per la valutazione della stringa: nell'esempio, il valore dell'espressione `(5+(4*2))` sarà ottenuto applicando l'operazione di somma ai valori delle due sottoespressioni, e così via. Questa idea di struttura può essere espressa in maniera rigorosa attraverso la nozione di *algebra sintattica* (Def.1.12), che vedremo nel seguito.

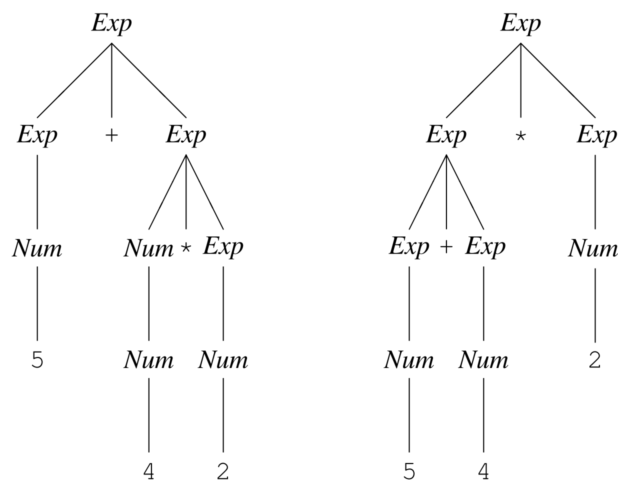


Figura 2: Due diversi alberi di derivazione per la stringa `5+4*2`.

1.4 Ambiguità

Introduciamo ora il concetto di ambiguità per le grammatiche. Intuitivamente, una grammatica è ambigua se esiste una stringa che può essere decomposta in due modi diversi (quindi interpretata semanticamente in due modi).
Consideriamo per esempio la seguente variante $GExp^{amb}$ della grammatica per le espressioni, in cui non utilizziamo parentesi.

$$\begin{array}{lcl} \mathit{Exp} & ::= & \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} \star \mathit{Exp} \mid \mathit{Num} \\ \mathit{Num} & ::= & \mathtt{0} \mid \mathtt{1} \mid \ldots \end{array}$$

Si consideri per esempio la stringa `5+4*2` che appartiene al linguaggio generato di tipo *Exp*. Intuitivamente, non sappiamo come interpretare tale espressione, se come la somma di `5` e `4*2` oppure il prodotto di `5+4` e `2`. È possibile quindi attribuire più di un significato alla stringa (è un problema che si verifica anche nei linguaggi naturali). Formalmente, ciò corrisponde al fatto che esiste più di un albero di derivazione per la stringa.

Def. 1.10 [Grammatica ambigua] Una grammatica è *ambigua* (rispetto a un non terminale A) se e solo se esiste una stringa che ammette due alberi di derivazione diversi a partire da A.

Si noti che, anche se frequentemente omessa, la precisazione *rispetto ad* A è necessaria; infatti la situazione in cui una stringa ammette due alberi di derivazione rispetto a due tipi diversi (per esempio se si hanno le produzioni $A ::= \mathtt{a}$, $B ::= \mathtt{a}$, oppure se si ha una produzione del tipo $A ::= B$) rappresenta un caso di *overloading* (uno stesso simbolo è usato in contesti di tipo diverso): si ha quindi un'ambiguità di diverso genere, che può però essere risolta se si ha in più l'informazione relativa a qual è il tipo che si sta considerando. Per esempio nel secondo caso si ha semplicemente che il linguaggio di tipo B risulta essere un sottoinsieme del linguaggio di tipo A.
La grammatica $GExp^{amb}$ è ambigua, infatti per esempio abbiamo due alberi di derivazione per la stringa `5+4*2`, illustrati in Fig.2.
Non sempre l'ambiguità è "indesiderata". Per esempio, la grammatica vista prima è ambigua non solo rispetto alle espressioni che contengono, per esempio, un $\star$ e un +, ma anche a quelle che contengono due occorrenze di +, come per esempio `5+3+2`. In tale espressione, non si sa se la strutturazione è $\langle$`5+3`$\rangle$`+2` o `5+`$\langle$`3+2`$\rangle$. Analoga situazione si ha per il $\star$. Ciò non ha importanza però dal punto di vista semantico, poiché le operazioni + e $\star$ sono associative, quindi nei due casi si ottiene lo stesso valore. Si dice in questo caso che l'ambiguità *non* è *semanticamente rilevante*. Per altri operatori binari infissi (per esempio −) l'ordine può essere rilevante; si dice che un operatore infisso *op associa a sinistra* (risp. destra) se un'espressione del tipo e_1 *op* e_2 *op* e_3 va decomposta come $\langle e_1$ *op* $e_2\rangle$ *op* e_3, *a destra* se va decomposta come e_1 *op* $\langle e_2$ *op* $e_3\rangle$. Nei L. di P. vale usualmente l'associatività a sinistra tra operatori con la stessa precedenza (ossia la convenzione è di svolgere i calcoli da sinistra a destra).
In generale si vorrebbero evitare i casi di ambiguità semanticamente rilevante. Si pongono quindi i seguenti problemi:

- riconoscere le grammatiche ambigue, cioè dare un algoritmo che mi dica se una grammatica è ambigua o no;

- eliminare l'ambiguità, cioè dare un algoritmo che data G ambigua (rispetto ad A) fornisca G' non ambigua e tale che $L_A(G) = L_A(G')$.

Entrambi questi problemi non sono però risolvibili per una *qualunque* grammatica; valgono infatti i seguenti risultati:

- il primo problema non è decidibile (ossia, non esiste un algoritmo che presa in input una grammatica qualunque produca in output sì o no a seconda se la grammatica è o non è ambigua);
- eliminare l'ambiguità è impossibile, cioè esistono linguaggi *inerentemente* ambigui, cioè tali che ogni grammatica che li genera è ambigua.

♠ Un esempio di linguaggio inerentemente ambiguo è $\{\mathtt{a}^n\mathtt{b}^p\mathtt{c}^q \mid n, p, q \geq 0, p = n \text{ oppure } p = q\}$ (le stringhe ambigue sono tutte quelle del tipo $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n, n \geq 0\}$). Una grammatica che lo genera è:

$$
\begin{array}{lcl}
S & ::= & A\,X \mid Y\,C \\
A & ::= & \mathtt{a}A \mid \Lambda \\
X & ::= & \mathtt{b}X\mathtt{c} \mid \Lambda \\
Y & ::= & \mathtt{a}Y\mathtt{b} \mid \Lambda \\
C & ::= & \mathtt{c}C \mid \Lambda
\end{array}
$$

È tuttavia possibile stabilire se *una particolare grammatica* che genera un linguaggio L è ambigua o no e, nel caso, costruire una grammatica non ambigua che generi L o un linguaggio "in relazione" con L, in un senso che sarà precisato alla fine della sezione.
Vediamo alcune tecniche di eliminazione dell'ambiguità:

Uso di parentesi/parole chiave La grammatica *GExp* vista prima genera un linguaggio diverso da quello generato da GExp^{amb} in cui le parole ambigue non compaiono; per esempio `5+4*2` scompare, "sostituita da" `((5+4)*2)` e `(5+(4*2))`.

Uso di notazione polacca prefissa o postfissa Un modo non ambiguo di rappresentare le espressioni è quello di utilizzare la cosiddetta notazione *polacca prefissa*, cioè di definire le espressioni con la seguente grammatica GExp^{prefix}.

$$
\begin{array}{lcl}
\mathit{Exp} & ::= & \star\ \mathit{Exp}\ \mathit{Exp} \mid +\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Num} \\
\mathit{Num} & ::= & \mathtt{0} \mid \mathtt{1} \mid \ldots
\end{array}
$$

Anche in questo caso la stringa ambigua scompare sostituita da `* + 5 4 2` e `+ 5 * 4 2`.
Analogamente, la notazione *polacca postfissa* consiste nel definire le espressioni con la seguente grammatica GExp^{post}.

$$
\begin{array}{lcl}
\mathit{Exp} & ::= & \mathit{Exp}\ \mathit{Exp}\ \star \mid \mathit{Exp}\ \mathit{Exp}\ + \mid \mathit{Num} \\
\mathit{Num} & ::= & \mathtt{0} \mid \mathtt{1} \mid \ldots
\end{array}
$$

La notazione utilizzata usualmente per gli operatori binari `+` e `*` è detta *infissa*, e si generalizza nel caso di operatori con arità[3] maggiore di due alla notazione *mixfix* o *distribuita infissa* (per esempio `if _ then _ else _ endif`, indicando con `_` la posizione degli argomenti). Le notazioni polacche si generalizzano facilmente a operatori di arità qualunque. La notazione prefissa è particolarmente comoda per il riconoscimento, quella postfissa per la valutazione automatica. Una variante della notazione prefissa è quella del tipo $op(e_1, \ldots, e_n)$; un'altra è $(op\ e_1 \ldots e_n)$ (utilizzata in Lisp).

Uso di grammatiche "a precedenza" Un modo per garantire che ogni stringa del linguaggio sia interpretata in modo univoco è quello di introdurre una gerarchia di priorità tra gli operatori. Per esempio, seguendo la convenzione standard, possiamo stabilire che il `*` ha maggiore priorità del `+`, quindi la struttura dell'espressione `5+4*2` sarà `5+`$\langle$`4*2`$\rangle$.
È possibile formalizzare questa regola di precedenza introducendo altri non terminali e strutturando le produzioni in base alle priorità, come illustrato dalla seguente grammatica GExp^{prec}.

$$
\begin{array}{lcl}
\mathit{Exp} & ::= & \mathit{Term} \mid \mathit{Term} + \mathit{Exp} \\
\mathit{Term} & ::= & \mathit{Num} \mid \mathit{Term} \star \mathit{Num} \\
\mathit{Num} & ::= & \mathtt{0} \mid \mathtt{1} \mid \ldots
\end{array}
$$

Si può dimostrare che vale:

[3] L'*arità* di un operatore è data dal numero di argomenti o, nel caso eterogeneo, dal numero e dal tipo, vedi Def.A.7.

1. $GExp^{prec}$ non è ambigua.

2. $L_{Exp}(GExp^{prec}) = L_{Exp}(GExp^{amb})$.

Si noti che, anche se la grammatica genera esattamente lo stesso linguaggio di prima, dato che ora ogni stringa del linguaggio è interpretata in un unico modo si ha l'inconveniente che non vi è più una stringa corrispondente alla decomposizione $\langle$`5 + 4`$\rangle$ `* 2`. Per ottenere anche queste stringhe, è necessario comunque utilizzare delle parentesi, per esempio aggiungendo la produzione *Num* ::= `(`*Exp*`)`.
Dato che questa tecnica complica notevolmente la grammatica, nella pratica si usa spesso distinguere due grammatiche per un L. di P., una ambigua accompagnata da regole di precedenza definite a parte (ad esempio $GExp^{amb}$ più la regola "il $\star$ ha la precedenza sul +"), e una non ambigua equivalente che viene utilizzata unicamente in fase di parsing (vedi Sez.1.9).

1.5 Algebre sintattiche e sintassi astratta

L'esempio visto precedentemente del linguaggio delle espressioni illustra il fatto che, quando definiamo un linguaggio (famiglia di insiemi) per mezzo di una grammatica, implicitamente definiamo anche degli *operatori* su stringhe. Per esempio, un'espressione della forma `(`e_1 `+` e_2`)` può essere vista come il risultato dell'applicazione dell'operatore binario + alle sottoespressioni e_1, e_2.
Quest'idea intuitiva è formalizzata dalla nozione di *segnatura* ed *algebra sintattica associate a una grammatica*.

Def. 1.11 [Segnatura associata a una grammatica] La *segnatura associata* alla grammatica $G = (T, N, P)$ è la segnatura (Def.A.7) $\Sigma(G) = (S, O)$ dove:

- $S = N$, cioè le sort sono esattamente i non terminali;
- per ogni produzione $A ::= u_0B_1u_1 \ldots B_nu_n$, con $B_1, \ldots, B_n \in N$, $u_1, \ldots, u_n \in T^\star$, vi è in O un operatore distribuito infisso $u_0_u_1 \ldots _u_n \colon B_1 \ldots B_n \to A$, e non vi sono altri operatori in O.

La segnatura associata a una grammatica G è anche detta *sintassi astratta* del linguaggio $L(G)$, perché tale segnatura descrive la struttura algebrica del linguaggio (i tipi e gli operatori) astraendo rispetto alle particolari rappresentazioni concrete possibili (cioè le algebre sintattiche, vedi sotto).
Volendo astrarre ulteriormente, si può osservare che anche i simboli di sort e di operazione effettivamente utilizzati sono irrilevanti; quindi la definizione precedente può anche essere espressa in modo più generale dicendo che la sintassi astratta è determinata da una qualunque segnatura in cui le sort sono in corrispondenza biunivoca con i non terminali e gli operatori sono in corrispondenza biunivoca con le produzioni nel senso precisato sopra.

Def. 1.12 [Algebra sintattica] L'*algebra sintattica* associata a una grammatica G è un'algebra L (Def.A.8) su $\Sigma(G)$ definita nel modo seguente:

- per ogni $s \in S$, se s corrisponde al non terminale A, $L_s = L_A(G)$ (ossia, i supporti dell'algebra sono esattamente i linguaggi generati dei vari tipi);
- per ogni operatore *op* corrispondente alla produzione $A ::= u_0B_1u_1 \ldots B_nu_n$, $op^L \colon L_{B_1}(G) \times \ldots \times L_{B_n}(G) \to L_A(G)$ è definito da

$$op^L(w_1, \ldots, w_n) = u_0w_1u_1 \ldots w_nu_n.$$

La definizione sopra esprime formalmente il fatto che il linguaggio generato da una grammatica non è semplicemente una famiglia di insiemi di stringhe, ma possiede anche una struttura algebrica (ossia, la grammatica definisce anche degli operatori). Questa struttura algebrica è esattamente quella che guida la valutazione semantica. Quando consideriamo il linguaggio come un'algebra e non semplicemente come una famiglia di stringhe, chiamiamo *espressioni*[4] o *termini* del linguaggio gli elementi dell'algebra, ossia le stringhe viste però come oggetti strutturati ottenuti applicando un operatore a delle sottocomponenti.
♠ Abbiamo parlato sopra di grammatiche che generano linguaggi "in relazione" tra loro. Questa nozione può ora essere espressa formalmente utilizzando le nozioni di segnatura e algebra sintattica associate a una grammatica.
Infatti, se G_1, G_2 sono due grammatiche a cui corrisponde la stessa sintassi astratta (a meno dei nomi di sort e operatori), una relazione tra le due algebre sintattiche associate, siano L_1 e L_2 può essere espressa formalmente con la nozione di *omomorfismo* di algebre (Def.A.9).

[4]In questo caso la parola "espressioni" viene usata in modo assolutamente generale: per esempio nel caso del linguaggio C espressioni del linguaggio saranno le dichiarazioni, i comandi, anche le espressioni propriamente dette, etc.

Per esempio le grammatiche $GExp$ e $GExp^{prefix}$ viste prima non generano esattamente lo stesso linguaggio, ma linguaggi "equivalenti" nel senso che le due algebre sintattiche $LExp$ e $LExp^{prefix}$ associate sono isomorfe. Invece la grammatica ambigua $GExp^{amb}$ genera un linguaggio $LExp^{amb}$ dove, intuitivamente, stringhe diverse in $LExp$ oppure $LExp^{prefix}$ vengono identificate (per esempio, alle stringhe `((5+4)*2)` e `(5+(4*2))` corrisponde la stessa stringa `5+4*2`. Tale fatto può essere espresso formalmente definendo un omomorfismo da $LExp$ in $LExp^{prefix}$; si noti che non esiste un omomorfismo in senso inverso.

1.6 Grammatiche come definizioni induttive

Si noti che una grammatica può essere vista come una definizione induttiva (di una famiglia di insiemi), in quanto ogni produzione

$$A ::= u_0 B_1 u_1 \ldots B_n u_n$$

può essere letta come

> se la stringa w_1 appartiene al linguaggio di tipo B_1, ..., e la stringa w_n appartiene al linguaggio di tipo B_n, allora la stringa $u_0 w_1 u_1 \ldots w_n u_n$ appartiene al linguaggio di tipo *A*.

Si noti che le definizioni induttive sono molto più generali; ad esempio, nelle definizioni induttive è possibile esprimere il fatto che due sottocomponenti della stringa devono essere uguali. Questo è un esempio di vincolo contestuale (vedi Sez.1.10).

1.7 Analisi lessicale

Nelle definizioni di alfabeto e grammatica date precedemente abbiamo posto un vincolo di finitezza. È chiaro infatti che, da un punto di vista pratico (cioè ai fini del riconoscimento automatico), siamo interessati a linguaggi su un alfabeto finito e descritti da una grammatica finita (cioè in cui anche l'insieme dei non terminali e delle produzioni sono finiti).
Tuttavia, i L. di P., così come i linguaggi naturali, possono essere visti come linguaggi "a due livelli", in cui i simboli dell'alfabeto sono a loro volta stringhe su un altro alfabeto, quindi l'alfabeto è infinito. Anche nel semplice esempio del linguaggio delle espressioni visto precedentemente, si ha che *Exp* è un linguaggio sull'alfabeto infinito *Num* $\cup \{$`+`, `*`, `(`, `)`$\}$, descritto da infinite produzioni. Analogamente, le frasi italiane sono stringhe sull'alfabeto D_{it} costituito dalla parole italiane, che a loro volta sono stringhe sull'alfabeto A_{it} delle lettere italiane.
Facendo riferimento ancora all'esempio del linguaggio *Exp*, come già osservato, questo può anche essere visto come un linguaggio sull'alfabeto *Digit* $\cup \{$`+`, `*`, `(`, `)`$\}$ dove *Digit* è l'insieme delle cifre decimali. In questo modo si avrebbe un alfabeto finito, e si potrebbe dare una grammatica finita che genera *Exp*.
Tuttavia nell'analisi sintattica dei L. di P. si preferisce adottare l'approccio a due livelli, in cui alcuni simboli del linguaggio sono visti come stringhe su un certo alfabeto di caratteri ammissibili. Più precisamente l'approccio è il seguente.
Nella definizione del linguaggio, alcune categorie sintattiche (tipicamente, identificatori e "literal", cioè rappresentazioni di valori di tipi primitivi) sono trattate in modo particolare, cioè sono considerate a tutti gli effetti come terminali nella grammatica, e per esse viene data a parte una descrizione del linguaggio generato. Questa descrizione è usualmente data attraverso un tipo più semplice di grammatiche dette *regolari*, che non vediamo in questo corso. Useremo nel seguito la convenzione di scrivere queste categorie sintattiche in MAIUSCOLETTO. Il vantaggio di questo approccio è di essere modulare (infatti, la definizione precisa di identificatori e literal è poco rilevante ai fini delle proprietà del linguaggio). Per esempio, per il linguaggio *Exp*, la grammatica sarà data nel modo seguente:

Exp ::= `(`*Exp* `+` *Exp*`)` | `(`*Exp* `*` *Exp*`)` | NUM

e la categoria sintattica NUM sarà definito a parte (per esempio, come l'insieme delle stringhe di cifre decimali senza zeri non significativi).
Corrispondentemente, un parser (analizzatore sintattico) per un L. di P. consta tipicamente di due sottocomponenti, lo *scanner* (*analizzatore lessicale*) e il *parser* propriamente detto.
Infatti, il testo di un programma di per sè è una stringa su un certo insieme di caratteri *Ch*; lo scanner lo trasforma in una stringa sull'alfabeto dei simboli del linguaggio *Sym*, tipicamente eliminando separatori come spazi, andate a capo e simili e compattando stringhe di caratteri che vanno considerati come un unico simbolo (detto anche "token"), come != in C, numerali, identificatori e parole chiave. Il parser invece utilizza la grammatica del linguaggio G e stabilisce se la stringa di simboli in input ricevuta dallo scanner appartiene o meno al linguaggio generato da G. La situazione è schematizzata in Fig.3.
Diamo ora, a titolo di esempio, un'implementazione in Java di uno scanner per un semplice linguaggio funzionale FL che sarà definito più avanti (vedi Fig.4). Si noti infatti che, ai fini dell'analisi lessicale di un linguaggio, non ci interessa avere la grammatica che lo definisce ma è sufficiente conoscere:

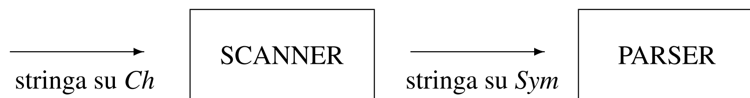


Figura 3: Analisi lessicale

- l'alfabeto dei terminali del linguaggio,
- la definizione precisa delle eventuali categorie sintattiche definite a parte,
- la definizione precisa di quali simboli vanno considerati separatori.

Nel caso di FL, l'alfabeto dei terminali è il seguente:

$$\Sigma_{\text{FL}} = \{\, \texttt{(}, \texttt{+}, \texttt{)}, \texttt{*}, \texttt{true}, \texttt{false}, \texttt{or}, \texttt{and}, \texttt{=}, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{endif}, \texttt{let}, \texttt{in}, \texttt{endlet}, \texttt{;}, \texttt{:}, \texttt{,}, \texttt{int}, \texttt{bool}\}.$$

Le categorie sintattiche definite a parte di FL sono due, NUM e ID, che assumiamo, per semplificare l'implementazione in Java, corrispondenti ai numerali in Java (cioè, le stringhe che costituiscono rappresentazioni di valori di tipo[5] `int`) e agli identificatori in Java.
Le convenzioni sui separatori sono, ancora per semplificare l'implementazione, quelle standard in Java (implementate nello scanner predefinito fornito dalla classe `StreamTokenizer`).
La seguente classe implementa uno scanner per FL. La classe provvede anzitutto a definire delle costanti (implementate come campi statici `final` interi) corrispondenti a tutti i simboli dell'alfabeto Σ_{FL}, più una costante per ogni categoria sintattica definita a parte (`TT_NUMBER` e `TT_IDENT`) più due costanti che corrispondono alla fine dell'input (`TT_EOF`) e all'aver incontrato un simbolo sconosciuto (`TT_UNKNOWN`). Un singolo oggetto scanner contiene inizialmente una certa stringa di caratteri (ciò è realizzato in Java passando come parametro al costruttore un oggetto di tipo `Reader`). Il metodo `nextToken` restituisce, a ogni successiva invocazione, il primo simbolo letto esaminando tale stringa (più precisamente viene restituita la costante intera che corrisponde al simbolo). L'implementazione data estende la classe predefinita Java `StreamTokenizer`, modificandone opportunamente il comportamento.

```
package FL;

import java.io.*;
import java.util.*;

public class FLTokenizer extends StreamTokenizer {

    protected static final Hashtable<String, Integer> reservedWords=new Hashtable<String, Integer>() ;

    public static final int TT_EOF=-1;
    public static final int TT_NUMBER=-2;
    public static final int TT_UNKNOWN=-3;
    public static final int TT_IDENT=-4;
    public static final int TT_OROUND=-5;
    public static final int TT_CROUND=-6;
    public static final int TT_SEMICOLON=-7;
    public static final int TT_COMMA=-8;
    public static final int TT_COLON=-9;
    public static final int TT_EQUAL=-10;
    public static final int TT_IF=-11;
    public static final int TT_THEN=-12;
    public static final int TT_ELSE=-13;
    public static final int TT_AND=-14;
    public static final int TT_PLUS=-15;
```

[5] In realtà lo scanner per FL è implementato come estensione della classe `StreamTokenizer`, che riconosce come numerali tutti i literal di tipo `double`, e abbiamo omesso per semplicità il controllo dell'assenza del punto decimale.

```
public static final int TT_TIMES=-16;
public static final int TT_TRUE=-17;
public static final int TT_FALSE=-18;
public static final int TT_OR=-19;
public static final int TT_LET=-20;
public static final int TT_IN=-21;
public static final int TT_ENDLET=-22;
public static final int TT_ENDIF=-23;
public static final int TT_INT=-24;
public static final int TT_BOOL=-25;

static {
    reservedWords.put("if",TT_IF);
    reservedWords.put("then",TT_THEN);
    reservedWords.put("else",TT_ELSE);
    reservedWords.put("true",TT_TRUE);
    reservedWords.put("false",TT_FALSE);
    reservedWords.put("let",TT_LET);
    reservedWords.put("in",TT_IN);
    reservedWords.put("endif",TT_ENDIF);
    reservedWords.put("endlet",TT_ENDLET);
    reservedWords.put("and",TT_AND);
    reservedWords.put("or",TT_OR);
    reservedWords.put("int",TT_INT);
    reservedWords.put("bool",TT_BOOL);
}

public FLTokenizer(Reader reader) {
    super(reader);
    resetSyntax();
}

public void resetSyntax(){
    super.resetSyntax();
    parseNumbers();
    ordinaryChar('-');
    ordinaryChar('.');
    whitespaceChars('\u0000','\u0020');
    wordChars('a','z');
    wordChars('A','Z');
    wordChars('_','_');
}

public int nextToken() throws IOException {
    return ttype = myNextToken() ;
}

private int myNextToken() throws IOException {
    super.nextToken();
    if (ttype==super.TT_EOF)
        return TT_EOF;
    else if (ttype==super.TT_NUMBER)
        return TT_NUMBER;
    switch(ttype) {
    case TT_WORD:
        Integer i=reservedWords.get(sval) ;
        return ttype = i!=null ? i : TT_IDENT ;
    case '+': return TT_PLUS;
    case '*': return TT_TIMES;
    case ':': return TT_COLON;
    case ';': return TT_SEMICOLON;
    case ',': return TT_COMMA;
```

```
        case '(': return TT_OROUND;
        case ')': return TT_CROUND;
        case '=': return TT_EQUAL;
        }
        return TT_UNKNOWN;
    }
}
```

Il codice sfrutta alcune caratteristiche di J2SE[6] 5.0 nuove rispetto alle precedenti versioni di Java. In particolare, si noti che la hashtable `reservedWords` viene dichiarata specificando che le sue chiavi avranno tipo `String` e i suoi valori tipo `Integer`. Inoltre, si noti che il compilatore effettua automaticamente la conversione da valori di tipi primitivi a oggetti delle corrispondenti classi "wrapper" (vedi Sez.2.3.4,2.4.12) e viceversa, quindi possiamo inserire nella hashtable direttamente degli interi, senza preoccuparci di passare attraverso la classe `Integer` e, analogamente, possiamo restituire un `Integer` quando il tipo di ritorno é `int` (senza dover chiamare il metodo standard `intValue`).

1.8 Rappresentazione interna

Per quanto visto sinora, un analizzatore sintattico (parser) per un linguaggio L è un programma che prende in input una stringa di simboli (ottenuta preliminarmente da una stringa di caratteri attraverso lo scanner) e restituisce vero o falso a seconda se la stringa appartiene o no al linguaggio generato dalla grammatica data.
Tuttavia, un parser non si limita in genere a eseguire il test di appartenenza ma, nel caso in cui la stringa in input sia effettivamente un programma nel linguaggio, produce anche una *rappresentazione interna* del programma.
A cosa serve la rappresentazione interna? È ovvio che se il nostro scopo finale fosse solo riconoscere le stringhe di un linguaggio, la rappresentazione interna sarebbe inutile; in genere, però, la fase di parsing non è fine a se stessa poiché, su un programma p letto dall'input e riconosciuto, vengono compiute altre elaborazioni: per esempio, controllare che p sia corretto staticamente (fase di type-checking, vedi Sez.1.10) oppure eseguire p (fase di interpretazione) oppure, ancora, tradurre p in un linguaggio macchina (fase di compilazione).
In teoria, si potrebbe pensare di unire tutte le fasi di elaborazione in un'unica fase, per esempio eseguendo l'analisi sintattica, statica e l'interpretazione di un programma in un unico passo, evitando così di doversi preoccupare di una rappresentazione interna. Tuttavia, tale approccio è raramente adottato poiché la realizzazione di un interprete (o un compilatore) a un solo passo presenta i seguenti problemi.

- Può essere molto più complessa, al punto di essere per certi linguaggi non fattibile in pratica.
- È poco modulare: una modifica che di per sè coinvolgerebbe una solo sottofase necessariamente si ripercuote su altre sottofasi. Per esempio, se si deve cambiare la sintassi concreta del linguaggio, invece di limitare la corrispondente modifica a una parte di codice ben precisa e relativamente limitata (ossia, il parser), si è obbligati a operare sull'intero programma. Lo stesso discorso vale per l'individuazione e la conseguente correzione di errori di programmazione.

Vediamo ora quale tipo di rappresentazione interna si adotta in genere. Anzitutto ricordiamo che, come visto in Sez.1.5, un linguaggio può essere visto, anzichè come semplicemente una famiglia di insiemi di stringhe, come un'*algebra*, i cui elementi sono le *espressioni* o *termini* del linguaggio, cioè le stringhe viste però come oggetti strutturati ottenuti applicando un operatore a certe sottocomponenti.
Dato che le successive fasi di elaborazione di un linguaggio sono sempre guidate dalla struttura delle espressioni (cioè, tipicamente, per effettuare una certa azione su un'espressione $op(e_1, \ldots, e_n)$ si effettuano azioni sulle sue sottocomponenti $e_1, \ldots, e_n$ e poi si combinano i risultati in un modo che è determinato dall'operatore op), risulta conveniente scegliere una rappresentazione per cui l'accesso alle sottocomponenti e la determinazione dell'operatore siano immediate. Se rappresentassimo internamente le espressioni del linguaggio allo stesso modo in cui vengono rappresentate esternamente, ossia mediante stringhe, questo non sarebbe vero. Infatti, il tipo di dato stringa non riflette questa struttura; per poter riconoscere le varie sottocomponenti di un'espressione rappresentata da una stringa, siamo obbligati a scrivere un programma abbastanza complicato (appunto il parser). Inoltre, la rappresentazione tramite stringhe è strettamente legata alla sintassi concreta.
Una buona rappresentazione interna dovrà invece permettere l'estrazione diretta delle sottocomponenti di un'espressione, e dovrà essere dipendente unicamente dalla sintassi astratta (cioè, dovrà restare la stessa anche cambiando algebra sintattica). Facendo riferimento per concretezza a un'implementazione in Java, uno schema possibile per ottenere una rappresentazione interna che soddisfi tale proprietà è il seguente.

[6] Java 2 Standard Edition.

Prog	::=	*Exp*
Exp	::=	`(`*Exp* + *Exp*`)` | `(`*Exp* $\star$ *Exp*`)` | NUM | `true` | `false` | `(`*Exp* `or` *Exp*`)` | `(`*Exp* `and` *Exp*`)` | `(`*Exp* = *Exp*`)`
		| `if` *Exp* `then` *Exp* `else` *Exp* `endif` | `let` *Decs* `in` *Exp* `endlet` | ID | ID`(`*Exps*`)`
Decs	::=	*Dec* | *Dec*`;`*Decs*
Dec	::=	ID : *PrimType* = *Exp* | ID`(`*Params*`)` : *PrimType* = *Exp*
Params	::=	*Param* | *Param*`,`*Params*
Param	::=	ID : *PrimType*
PrimType	::=	`int` | `bool`
Exps	::=	*Exp* | *Exp*`,`*Exps*

Figura 4: Sintassi di FL

Per ogni non terminale *A* del linguaggio, si definirà un'interfaccia `A` in Java. Per ogni produzione avente *A* come parte sinistra, sia

$$A ::= u_0 B_1 u_1 \dots B_n u_n,$$

si definirà una classe che implementa l'interfaccia `A` corrispondente alla parte destra della produzione. Questa classe avrà, per ogni non terminale B_i, un campo del tipo (interfaccia Java) corrispondente `B_i`. In questo modo è chiaro che le sottocomponenti di un'espressione possono essere ottenute semplicemente selezionando il campo corrispondente.
A titolo di esempio, diamo nel seguito parte della rappresentazione interna in Java[7] per il linguaggio FL definito in Fig.4.
Come già detto, è stata introdotta un'interfaccia per ogni non terminale della grammatica e una classe che la implementa per ogni produzione che lo definisce (seguendo la metodologia generale di implementazione nel paradigma object-oriented di tipi unione, vedi Sez.2.3.3). La classe `BinExp` è stata introdotta per evitare di duplicare codice. Per le categorie sintattiche di tipo "lista non vuota" (*Decs*, *Params*, *Exps*), è stata introdotta, oltre all'interfaccia, un'unica classe che la implementa (sfruttando la classe predefinita `ArrayList`); ovviamente questa è una scelta implementativa e si poteva fare diversamente (per esempio due classi che la implementano, una per il tipo "singolo elemento" e una per il tipo "coppia elemento, lista", analogamente all'esempio illustrato in Sez.2.3.3).

```
public interface Prog { }

public class ExpProg implements Prog{
    private Exp exp;
    ExpProg(Exp exp){
        this.exp=exp;
    }
}

public interface Exp { }

public abstract class BinExp implements Exp {
    protected Exp exp1, exp2;
    protected BinExp(Exp exp1, Exp exp2) {
        this.exp1=exp1;
        this.exp2=exp2;
    }
}

public class SumExp extends BinExp {
    public SumExp(Exp exp1,Exp exp2){
        super(exp1,exp2);
    }
}
```

[7] L'implementazione completa di un interprete per FL è disponibile sulla pagina web del corso.

```
public class ProdExp extends BinExp {
    public ProdExp(Exp exp1,Exp exp2){
        super(exp1,exp2);
    }
}

public class NumExp implements Exp {
    private IntValue nval;

    public NumExp(int i){
        this.nval=new IntValue(i);
    }
}

...

public interface Decs extends Iterable<Dec> {
    void add(Dec dec);
}

public class DecsClass implements Decs {
    private List<Dec> decs=new ArrayList<Dec>() ;

    public DecsClass() {}

    public DecsClass(Dec dec){
        add(dec);
    }

    public void add(Dec dec){
        decs.add(dec);
    }

    public Iterator<Dec> iterator() {
        return decs.iterator() ;
    }
}

public interface Dec {}

public class ConstDec implements Dec{
    private Ident id;
    private PrimType type;
    private Exp exp;

    public ConstDec(Ident id,PrimType type,Exp exp){
        this.id=id;
        this.type=type;
        this.exp=exp;
    }
}

public class FunDec implements Dec{
    private Ident id;
    private Params params;
    private PrimType type;
    private Exp exp;

    public FunDec(Ident id,Params params,PrimType type,Exp exp){
        this.id=id;
        this.params=params;
        this.type=type;
```

```
        this.exp=exp;
    }
}

...
```

Si noti che tutte le interfacce che corrispondono a categorie sintattiche i cui elementi sono liste estendono l'interfaccia predefinita `Iterable<T>`, dove `T` rappresenta il tipo degli elementi. Per esempio, `Decs` estende `Iterable<Dec>`. In questo modo, un oggetto di tipo `Decs` offre la possibilità di effettuare iterativamente una certa azione su una sequenza di oggetti di tipo `Dec`. Questa caratteristica permette di utilizzare il costrutto `for`, introdotto in J2SE 5.0, su oggetti di queste categorie sintattiche. Per esempio:

```
void printAllDecs(Decs decs) {
    for(Dec d : decs)
        System.out.println(d) ;
}
```

Il nuovo costrutto `for` può, allo stesso modo, essere usato per iterare su un array; ad esempio:

```
int sum(int[] a) {
    int result = 0;
    for (int i : a) result += i;
    return result;
}
```

1.9 Schema di parsing

Considereremo il caso particolarmente semplice in cui la grammatica è tale che è sufficiente l'esame del primo simbolo letto per accertare quale è la produzione da applicare.
Facendo riferimento per comodità a un'implementazione in Java, uno schema di algoritmo di parsing è il seguente.
Per ogni non terminale *A* andrà previsto un metodo `parseA` che, esaminando via via una sequenza di simboli del linguaggio (prodotti dallo scanner), restituisce un oggetto di tipo *A* se la sequenza corrisponde alla parte destra di una produzione con parte sinistra *A*. Questo metodo per prima cosa deciderà, guardando il primo simbolo, qual è la produzione applicabile: sia

$$A ::= u_0 B_1 u_1 \dots B_n u_n$$

con $B_1, \dots, B_n$ non terminali e $u_0, \dots, u_n$ stringhe di terminali. Allora il codice successivo seguirà lo schema:

- controllo che ci sia u_0; se sì
- richiamo il metodo `parseB_1`; se questo termina con successo (quindi mi restituisce un oggetto o_1 di tipo `B_1`)
- controllo che ci sia u_1; se sì
- ...
- controllo che ci sia u_{n-1}; se sì
- richiamo il metodo `parseB_n`; se questo termina con successo (quindi mi restituisce un oggetto o_n di tipo `B_n`)
- controllo che ci sia u_n; se sì
- restituisco l'oggetto ottenuto componendo $o_1, \dots, o_n$.

Si noti che, applicando questo schema, quando viene invocato il metodo `parseA` il simbolo corrente è sempre il primo simbolo "utile" (cioè il primo simbolo della stringa di tipo A da riconoscere), e quando termina l'esecuzione di `parseA` il simbolo corrente è sempre il primo successivo all'ultimo simbolo "utile" (quindi il successivo `parseB` invocato si trova in situazione analoga).
Diamo, a titolo di esempio, un'implementazione in Java di un parser per FL.

```
package FL;

import java.io.*;

public class FLParser {

    protected FLTokenizer tkr;

    public FLParser(FLTokenizer tkr) {
        this.tkr=tkr;
    }

    static private boolean isBinaryOp(int op) {
        switch(op){
             case FLTokenizer.TT_PLUS:
             case FLTokenizer.TT_TIMES:
             case FLTokenizer.TT_OR:
             case FLTokenizer.TT_AND:
             case FLTokenizer.TT_EQUAL:
                 return true;
        }
        return false;
    }

    public Prog parseProg(boolean interactiveMode) throws ParseException,IOException {
        tkr.nextToken();
        Prog p=new ExpProg(parseExp());
        if (interactiveMode) {
            if (tkr.ttype!=FLTokenizer.TT_SEMICOLON)
                throw new ParseException("programs must be ended by ';'");
        } else if (tkr.ttype!=FLTokenizer.TT_EOF)
                throw new ParseException("unexpected token at the end of the program");
        return p;
    }

    private void expecting(int tok, String message) throws ParseException, IOException {
        if (tkr.ttype==tok)
             tkr.nextToken();
        else throw new ParseException(message) ;
    }

    private Exp parseBoolean(int ttype) throws IOException {
        tkr.nextToken();
        return ttype==FLTokenizer.TT_TRUE ? TrueExp.TRUE : FalseExp.FALSE ;
    }

    public Exp parseExp() throws ParseException,IOException {
        switch (tkr.ttype){
        case FLTokenizer.TT_OROUND: return parseRound() ;
        case FLTokenizer.TT_NUMBER: return parseNumber() ;
        case FLTokenizer.TT_IF: return parseIf() ;
        case FLTokenizer.TT_LET: return parseLet() ;
        case FLTokenizer.TT_IDENT: return parseId() ;
        case FLTokenizer.TT_TRUE:
        case FLTokenizer.TT_FALSE: return parseBoolean(tkr.ttype) ;
```

```
    }
    throw new ParseException("syntax error while reading an expression");
}

private Exp parseRound() throws ParseException,IOException {
    tkr.nextToken();
    Exp exp1=parseExp();
    int op=tkr.ttype;
    if (!isBinaryOp(op))
        throw new ParseException("expecting a binary operator");
    tkr.nextToken();
    Exp exp2=parseExp();
    expecting(FLTokenizer.TT_CROUND, "expecting ')'");
    switch(op) {
    case FLTokenizer.TT_PLUS: return new SumExp(exp1,exp2);
    case FLTokenizer.TT_TIMES: return new ProdExp(exp1,exp2);
    case FLTokenizer.TT_OR: return new OrExp(exp1,exp2);
    case FLTokenizer.TT_AND: return new AndExp(exp1,exp2);
    case FLTokenizer.TT_EQUAL: return new EqExp(exp1,exp2);
    }
    throw new ParseException("Internal error in parseExp(); op="+op) ;
}

private NumExp parseNumber() throws ParseException,IOException {
    int nval=(int)tkr.nval;
    tkr.nextToken();
    return new NumExp(nval);
}

private IfExp parseIf() throws ParseException,IOException {
    tkr.nextToken();
    Exp ifExp=parseExp();
    expecting(FLTokenizer.TT_THEN, "expecting 'then'");
    Exp thenExp=parseExp();
    expecting(FLTokenizer.TT_ELSE, "expecting 'else'");
    Exp elseExp=parseExp();
    expecting(FLTokenizer.TT_ENDIF, "expecting 'endif'");
    return new IfExp(ifExp,thenExp,elseExp);
}

private LetExp parseLet() throws ParseException,IOException {
    tkr.nextToken();
    Decs decs=parseDecs();
    expecting(FLTokenizer.TT_IN, "expecting 'in'");
    Exp exp=parseExp();
    expecting(FLTokenizer.TT_ENDLET, "expecting 'endlet'");
    return new LetExp(decs,exp);
}

private Exp parseId() throws ParseException,IOException {
    Ident id=new Ident(tkr.sval);
    if (tkr.nextToken()!=FLTokenizer.TT_OROUND)
        return new IdExp(id);
    tkr.nextToken();
    Exps exps=parseExps();
    expecting(FLTokenizer.TT_CROUND, "expecting ')'");
    return new FunCallExp(id,exps);
}

public Decs parseDecs() throws ParseException,IOException {
    Decs decs=new DecsClass(parseDec());
    while (tkr.ttype==FLTokenizer.TT_SEMICOLON) {
```

```
        tkr.nextToken();
        decs.add(parseDec());
    }
    return decs;
}

public Dec parseDec() throws ParseException,IOException {
    if (tkr.ttype!=FLTokenizer.TT_IDENT)
        throw new ParseException("expecting an identifier") ;
    Ident id=new Ident(tkr.sval);
    switch(tkr.nextToken()) {
    case FLTokenizer.TT_COLON: return parseConstDec(id) ;
    case FLTokenizer.TT_OROUND: return parseFunDec(id) ;
    }
    throw new ParseException("expecting either ':' or '('");
}

private ConstDec parseConstDec(Ident id) throws ParseException,IOException {
    tkr.nextToken();
    PrimType type=parsePrimType();
    expecting(FLTokenizer.TT_EQUAL, "expecting '='");
    Exp exp=parseExp();
    return new ConstDec(id,type,exp);
}

private FunDec parseFunDec(Ident id) throws ParseException,IOException {
    tkr.nextToken();
    Params params=parseParams();
    expecting(FLTokenizer.TT_CROUND, "expecting ')'");
    expecting(FLTokenizer.TT_COLON, "expecting ':'");
    PrimType retType=parsePrimType();
    expecting(FLTokenizer.TT_EQUAL, "expecting '='");
    Exp exp=parseExp();
    return new FunDec(id,params,retType,exp);
}

public Params parseParams() throws ParseException,IOException {
    Params params=new ParamsClass(parseParam()) ;
    while (tkr.ttype==FLTokenizer.TT_COMMA) {
        tkr.nextToken();
        params.add(parseParam());
    }
    return params;
}

public Param parseParam() throws ParseException,IOException {
    if (tkr.ttype!=FLTokenizer.TT_IDENT)
        throw new ParseException("expecting an identifier");
    Ident id=new Ident(tkr.sval) ;
    tkr.nextToken();
    expecting(FLTokenizer.TT_COLON, "expecting ':'");
    return new PrimTypeParam(id, parsePrimType()) ;
}

public PrimType parsePrimType() throws ParseException,IOException {
    switch(tkr.ttype){
    case FLTokenizer.TT_INT:
        tkr.nextToken();
        return IntType.CONST;
    case FLTokenizer.TT_BOOL:
        tkr.nextToken();
        return BoolType.CONST;
```

```
        }
        throw new ParseException("expecting a primitive type");
    }

    public Exps parseExps() throws ParseException,IOException {
        Exps exps=new ExpsClass(parseExp());
        while (tkr.ttype==FLTokenizer.TT_COMMA){
            tkr.nextToken();
            exps.add(parseExp());
        }
        return exps;
    }

}
```

Nel caso nella fase di parsing si trovi un errore, viene sollevata la seguente eccezione.

```
public class ParseException extends Exception {

    public ParseException(){
        super("Parsing error");
    }

    public ParseException(String msg){
        super("Parsing error: "+msg);
    }

}
```

1.10 Vincoli contestuali

Ci sono vari motivi per cui in genere l'insieme dei programmi corretti in un determinato L. di P. non può essere definito semplicemente da una grammatica libera da contesto. Vediamo alcuni esempi.

Uso di identificatori Un primo esempio tipico è quello degli identificatori non dichiarati. Nei linguaggi di programmazione è possibile in genere *dichiarare* identificatori, cioè introdurre nomi simbolici per entità di vario tipo (identificatori di costante, di funzione, di variabile, di tipo, ...) che possono essere utilizzati in altri punti del programma. Per esempio, il semplicissimo linguaggio di espressioni visto precedentemente può essere arricchito con la possibilità di introdurre identificatori di costante, come illustrato dalla seguente grammatica:

$$Exp \;::=\; \texttt{(}Exp + Exp\texttt{)} \mid \texttt{(}Exp \star Exp\texttt{)} \mid \textsc{Num} \mid \textsc{Id} \mid \texttt{let}\ \textsc{Id} = Exp\ \texttt{in}\ Exp\ \texttt{endlet}$$

dove ID è una categoria sintattica che definisce un insieme di identificatori, definita a parte.
Un esempio di espressione in questo linguaggio è, assumendo che X sia un identificatore:

```
let X = ((1 + 2) * (5 + 3)) in
   let X = (X + X) in (X * X) endlet
endlet
```

Questo semplice esempio illustra due fondamentali caratteristiche dell'uso di identificatori nei L. di P.: anzitutto, l'uso di un nome simbolico permette di non riscrivere porzioni di codice utilizzate più volte (per esempio, la sottoespressione `((1 + 2) * (5 + 3)))` e quindi, in caso di necessità di modifica, di circoscrivere tale modifica alla sola definizione dell'identificatore (principio di localizzazione visto nell'Introduzione); inoltre, ogni dichiarazione di un identificatore *id* "è valida" solo all'interno di una determinata porzione di codice, detta *scope* della dichiarazione; le occorrenze di *id* che compaiono in tale porzione di codice si dicono *legate* all'occorrenza di *id* che appare nella dichiarazione, detta *legante*. Nell'esempio, le due X nella sottoespressione `(X + X)` fanno riferimento (sono legate) alla prima dichiarazione, quindi denotano il valore dell'espressione `((1 + 2) * (5 + 3))`, cioè 24, mentre le due X nella sottoespressione `(X * X)` fanno riferimento (sono legate) alla seconda dichiarazione, quindi denotano il valore 48. Questa caratteristica è molto importante in quanto in tal modo non è necessario che *tutti* gli identificatori

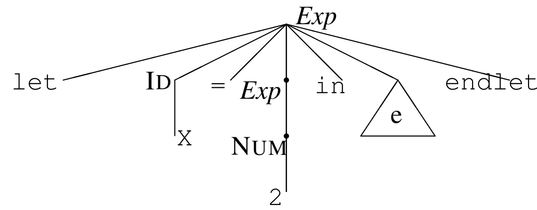


Figura 5:

dichiarati in un programma siano diversi (cosa che sarebbe impraticabile), ma ogni modulo può utilizzare nomi *locali*, cioè visibili solo al suo interno, che non creano quindi conflitto con nomi uguali utilizzati in altri moduli con cui il modulo venga successivamente composto.
Se in un'espressione e compare un'occorrenza di un identificatore *id* che non è legato a nessuna dichiarazione, si dice che *id* è *libero* in e. Intuitivamente, un'espressione che contiene identificatori liberi non denota un unico valore, ma un valore dipendente dal contesto. Per esempio l'espressione `(X + 5)` denoterà il valore 7 se inserita in un contesto in cui `X` è stato dichiarato con valore 2. Formalmente quindi il valore dell'espressione sarà una funzione da contesti (ambienti) in valori (vedi in Sez.1.13).
Possiamo definire induttivamente l'insieme $FV(e)$ delle variabili[8] *libere* di un'espressione e nel modo seguente.

$FV(\texttt{(}e_1 + e_2\texttt{)}) = FV(e_1) \cup FV(e_2)$
$FV(\texttt{(}e_1 \star e_2\texttt{)}) = FV(e_1) \cup FV(e_2)$
$FV(n) = \emptyset$, *per* $n \in$ NUM
$FV(id) = \{id\}$, *per id* $\in$ ID
$FV(\texttt{let}\ id = e_1\ \texttt{in}\ e_2\ \texttt{endlet}) = FV(e_1) \cup (FV(e_2) \setminus \{id\})$

Supponiamo ora di voler definire formalmente il linguaggio *Prog* formato da tutte le espressioni *chiuse*, che cioè non contengano variabili libere. Si può provare che tale linguaggio *non* è ottenibile come il linguaggio generato da una grammatica (libera da contesto), cioè non esiste G grammatica[9] tale che $L(G) = Prog$. Il motivo per cui non si riesce a dare tale grammatica può essere illustrato intuitivamente nel modo seguente. Si consideri un'espressione della forma `let X = 2 in` e `endlet`. Per determinare se questa è un'espressione corretta del linguaggio generato da una grammatica si cerca un albero di derivazione che sarà quindi della forma illustrata in Fig.5.
A questo punto si cerca un albero di derivazione per la sottoespressione e. Si noti che questa ricerca è effettuata in modo assolutamente indipendente dal *contesto* (infatti come già osservato in un albero di derivazione i diversi non terminali vengono espansi *in parallelo*), quindi va bene qualunque espressione, anche un'espressione che contenga variabili libere diverse da `X`. È proprio per questo motivo che si parla di grammatiche *libere da contesto*. Nei L. di P. invece per determinare se un'espressione è una sottocomponente corretta di un programma si ha bisogno in genere di *informazioni contestuali*; nell'esempio, per determinare se e è una sottoespressione corretta occorre sapere che dovrà essere inserita in un contesto in cui l'unica variabile dichiarata è `X`.
Si noti che, se l'insieme degli identificatori è finito, è possibile risolvere il problema complicando la grammatica, introducendo un non terminale per ogni possibile insieme di identificatori (si provi a farlo per esercizio).

Uso di tipi Un altro motivo per cui in genere i L. di P. non sono esprimibili attraverso una grammatica libera da contesto è dato dalla presenza di *vincoli di tipo*. Illustriamo il significato di tale termine attraverso alcuni semplici esempi.
Consideriamo un linguaggio "misto" formato da espressioni intere e booleane, definito dalla seguente grammatica:

$$Exp\ ::=\ \texttt{(}Exp + Exp\texttt{)} \mid \texttt{(}Exp \star Exp\texttt{)} \mid \text{NUM} \mid \texttt{true} \mid \texttt{false} \mid \texttt{(}Exp\ \texttt{or}\ Exp\texttt{)} \mid \texttt{(}Exp\ \texttt{and}\ Exp\texttt{)} \mid \texttt{(}Exp = Exp\texttt{)}$$

È chiaro che la grammatica data sopra definisce un linguaggio in cui sono presenti, oltre a tutte le espressioni intuitivamente corrette, anche espressioni "errate", come per esempio `(true and 2)`.

[8]Talvolta si dice "variabile" invece di "identificatore", utilizzando la terminologia della matematica e della logica. Tuttavia, occorre stare attenti a non confondersi con la nozione di "variabile" nei linguaggi imperativi, che significa una locazione di memoria. Gli identificatori nei linguaggi di programmazione possono denotare vari tipi di entità, tra le quali anche le variabili.

[9]Il problema può essere ridotto a quello di trovare una grammatica libera da contesto che generi il linguaggio $\{ww \mid w \in \Sigma^\star\}$ per un dato alfabeto Σ.

In questo semplice esempio, tuttavia, è possibile dare una grammatica che definisca effettivamente solo le espressioni corrette, utilizzando due non terminali, uno per le espressioni di tipo intero, uno per le espressioni di tipo booleano, come mostrato in Sez.1.2.
Ciononostante, spesso nella pratica si preferisce scegliere una grammatica più semplice, in modo da facilitare la fase di parsing, e rimandare ad una fase successiva, detta di *type-checking* o *analisi statica* i controlli di tipo. Inoltre, diventa impossibile descrivere tramite una grammatica libera da contesto i vincoli di tipo nel caso, molto comune, in cui i *tipi stessi* non siano un insieme finito, ma siano a loro volta descritti tramite una grammatica. Consideriamo per esempio il semplice linguaggio funzionale FL definito in Sez.1.9.
In questo linguaggio possono essere introdotti identificatori di costante o di funzione. Un identificatore di costante o funzione deve ovviamente essere utilizzato in accordo con il tipo di ritorno ed i tipi per i parametri con cui è stato dichiarato; per esempio, le espressioni `let X : int = 0 in X(1) endlet` e `let F(X : int) : int = (X + 1) in (F + F) endlet` non sono programmi corretti, mentre lo è l'espressione `let F(X : int) : int = (X + 1) in F(1) endlet`.

Semantica statica Quanto visto finora può essere riassunto nel modo seguente. In genere, la specifica dell'insieme dei programmi corretti in un L. di P. viene strutturata in due parti, cui corrispondono due fasi distinte nell'implementazione del linguaggio.

- Si fornisce una grammatica G che genera un *soprainsieme* $L(G)$ del linguaggio; nell'implementazione del linguaggio, come già visto, si chiama *parsing* (*analisi sintattica*) la fase che si occupa di controllare che la stringa in input sia effettivamente un elemento di $L(G)$.
- A parte, si fornisce una descrizione, formale o informale, dei cosiddetti *vincoli contestuali*, cioè tutte quelle proprietà aggiuntive che gli elementi di $L(G)$ devono soddisfare per essere espressioni del linguaggio dato (esempi tipici sono l'assenza di identificatori usati e non dichiarati e il rispetto dei tipi); nell'implementazione del linguaggio, si chiama *type-checking* o *analisi statica* la fase che si occupa di controllare che il programma soddisfi tali vincoli. Tale descrizione è detta anche *semantica statica* del linguaggio, perchè si specificano proprietà che è possibile controllare *prima* dell'esecuzione del programma.

Chiameremo quindi errori *statici* gli errori rilevati dal type-checker (corrispondenti a violazioni dei vincoli contestuali), *dinamici* quelli che vengono rilevati in fase di esecuzione. Diremo che un programma è *staticamente corretto* se non contiene errori statici. Si noti che l'organizzazione della descrizione di un linguaggio in sintassi, semantica statica e semantica dinamica è sempre in parte arbitraria. In altri termini, il confine tra errori di parsing, errori statici ed errori dinamici non è assoluto, ma è spesso più che altro metodologico. Infatti, abbiamo già visto che il rilevamento di alcuni errori di tipo statico (violazione di vincoli contestuali) può essere anticipato alla fase di parsing complicando la grammatica (anche se non sempre questo è possibile); analogamente, il confine tra errori statici (rilevati prima dell'esecuzione) ed errori dinamici non è sempre netto, e spesso dipende anche da considerazioni metodologiche. In alcuni casi è abbastanza chiaro che un errore non può essere rilevato staticamente: per esempio, assumendo di avere l'operatore di divisione intera, l'errore "divisione per zero" non può in genere essere rilevato prima di valutare l'espressione per cui si divide, ed è quindi un tipico esempio di errore dinamico (rilevabile solo al momento dell'esecuzione). Tuttavia, anche in questo esempio si potrebbe anticipare la gestione dell'errore all'analisi statica almeno in qualche caso evidente, per esempio quando il divisore è proprio l'espressione `0`, anche se non sarebbe di molto vantaggio pratico. In altri casi la scelta è molto più arbitraria, e in genere avviene sulla base di un compromesso tra il vantaggio di rilevare il maggior numero di errori possibile staticamente e lo svantaggio di restringere esageratamente l'insieme dei programmi corretti e/o complicare eccessivamente la semantica statica (quindi il type-checker). In genere, una semantica statica più restrittiva è più sicura dal punto di vista della produzione di software corretto, in quanto il programmatore è costretto a soddisfare più vincoli che corrispondono a requisiti di consistenza logica del codice ed è avvertito automaticamente di eventuali errori in un maggior numero di casi.

1.11 Semantica statica di FL

In questa sezione, a titolo di esempio, diamo la semantica statica formale del semplice linguaggio funzionale FL introdotto precedentemente.
L'idea intuitiva è che dobbiamo dire quali programmi (espressioni chiuse) del linguaggio sono staticamente corretti (cioè soddisfano i vincoli contestuali). Formalmente, questo può essere modellato da una funzione a due valori

$$wf_{Prog} : Prog \to \{ok, error\}$$

che dato un programma p restituisce *ok* se il programma è staticamente corretto (o *ben formato*, da cui *wf* per "well-formed"), *error* altrimenti.

Naturalmente però il fatto che un programma sia o meno corretto dipende da come sono fatte le sue sottocomponenti. Quindi, per definire questa funzione avremo bisogno di dire, per ogni categoria sintattica, quali termini del linguaggio corrispondente sono staticamente corretti, quindi di definire un analoga funzione wf_A per ogni non terminale A. Per esempio, nel caso di FL, dovremo dire quando sono staticamente corrette le dichiarazioni, le espressioni, etc.
Si noti però che, mentre per un programma p del linguaggio ha senso porsi la domanda "p è staticamente corretto o no", per un generico termine tale domanda non ha senso, in quanto, come già osservato, la correttezza o meno dipende dal contesto in cui questo si trova; ad esempio, in FL l'espressione `((x + y) = 0)` sarà corretta solo se inserita in un contesto in cui `x` e `y` sono stati dichiarati come identificatori di costante di tipo intero.
Inoltre, in generale non avremo semplicemente bisogno di sapere se un termine è ben formato, ma in qualche caso avremo anche bisogno di sapere (per poter controllare la correttezza dei termini che lo contengono come sottocomponente) di altre informazioni. Per esempio per concludere che l'espressione precedente è corretta avremo bisogno di sapere non solo che le sottocomponenti `(x + y)` e `0` sono corrette, ma anche che sono di tipo intero, altrimenti non si potrebbe applicare l'operatore somma.
Di conseguenza, le funzioni wf_A in generale potranno avere un argomento aggiuntivo (detto *ambiente statico*) che corrisponde alle informazioni contestuali necessarie per poter controllare la correttezza, e potranno restituire, in caso di correttezza, non semplicemente *ok* ma altre informazioni (per esempio un tipo).
Nel caso di FL, è facile rendersi conto che l'informazione necessaria per controllare la correttezza consiste nel sapere quali identificatori di costante e di funzione sono stati dichiarati e con quale tipo. Quindi definiremo gli ambienti statici nel modo seguente:

$$\mathit{StaticEnv} = \textsc{Id} \rightarrow \mathit{Type}$$

dove l'insieme *Type* dei tipi del linguaggio è definito dalla seguente grammatica:

$$\begin{array}{lcl} \mathit{Type} & ::= & \mathit{PrimType} \mid \mathit{FunType} \\ \mathit{FunType} & ::= & \mathit{PrimTypes} \rightarrow \mathit{PrimType} \\ \mathit{PrimTypes} & ::= & \mathit{PrimType} \mid \mathit{PrimType} \star \mathit{PrimTypes} \end{array}$$

I tipi sono cioè costituiti dai tipi primitivi del linguaggio (`int` e `bool`) e dai tipi funzionali, che consistono di una coppia formata dalla sequenza dei tipi degli argomenti e dal tipo di ritorno.
Per esempio, la precedente espressione `((x + y) = 0)` risulterà corretta e di tipo `int` in ogni ambiente r per cui valga $r(\mathtt{x}) = \mathtt{int}, r(\mathtt{y}) = \mathtt{int}$.
Diamo ora la semantica statica formale completa del linguaggio FL.
Avremo le seguenti funzioni di type-checking.

- wf_{Prog}: $\mathit{Prog} \rightarrow \{ok, error\}$ per i programmi (espressioni chiuse): il type-checking di un programma produce *ok* oppure *error*;
- wf_{Exp}: $\mathit{Exp} \rightarrow \mathit{StaticEnv} \rightarrow \mathit{PrimType} \cup \{error\}$ per le espressioni: il type-checking di un'espressione, in un ambiente che assegna tipi agli identificatori, produce un tipo primitivo oppure *error*;
- wf_{Decs}: $\mathit{Decs} \rightarrow \mathit{StaticEnv} \rightarrow \mathit{StaticEnv} \cup \{error\}$ per le (sequenze di) dichiarazioni: l'elaborazione di una sequenza di dichiarazioni, in un ambiente che assegna tipi agli identificatori, produce un nuovo ambiente in cui sono state aggiunte associazioni per gli identificatori dichiarati oppure *error*.

Le funzioni di type-cheking sono definite formalmente in Fig.6.
Un programma è un'espressione corretta di un certo tipo primitivo senza variabili libere, cioè un'espressione che risulta di quel tipo nell'ambiente vuoto $\emptyset$.
Le prime nove clausole di definizione della funzione wf_{Exp} formalizzano gli ovvi vincoli di tipo relativi agli operatori aritmetici e booleani e all'operatore `if _ then _ else _ endif`.
Un'espressione della forma `let` ds `in` e `endlet` è un'espressione corretta di tipo PT in r se ds è una sequenza di dichiarazioni corretta in r ed e è un'espressione corretta di tipo PT nell'ambiente locale ottenuto elaborando le dichiarazioni ds.
Un'identificatore id è un'espressione corretta di tipo PT in r se id è stato dichiarato come identificatore di costante di tipo PT; analogamente, una chiamata di funzione id`(`e_1`, ... ,`e_n`)` è un'espressione corretta di tipo PT in r se id è stato dichiarato come identificatore di funzione con argomenti di tipo $PT_1, \ldots, PT_n$ e tipo di ritorno PT ed $e_1, \ldots, e_n$ sono espressioni corrette di tipo $PT_1, \ldots, PT_n$, rispettivamente, in r.
Una sequenza di dichiarazioni d `;` ds è corretta in r se d è una dichiarazione corretta in r e ds è una sequenza di dichiarazioni corretta nell'ambiente ottenuto elaborando d in r. In altre parole, le dichiarazioni vengono elaborate nell'ordine. Si noti che, in base alla definizione data, nel caso in una sequenza di dichiarazioni lo stesso identificatore venga dichiarato più volte l'ultima

$wf_{Prog}(p) = ok$ se $wf_{Exp}(p)\ \emptyset\ \in PrimTypes$, *error* altrimenti

$wf_{Exp}(\texttt{(}e_1 \texttt{ + } e_2\texttt{)})\ r = \texttt{int}$ se $wf_{Exp}(e_1)\ r = \texttt{int} = wf_{Exp}(e_2)\ r$, *error* altrimenti
$wf_{Exp}(\texttt{(}e_1 \texttt{ * } e_2\texttt{)})\ r = \texttt{int}$ se $wf_{Exp}(e_1)\ r = \texttt{int} = wf_{Exp}(e_2)\ r$, *error* altrimenti
$wf_{Exp}(n)\ r = \texttt{int}$
$wf_{Exp}(\texttt{true})\ r = \texttt{bool}$
$wf_{Exp}(\texttt{false})\ r = \texttt{bool}$
$wf_{Exp}(\texttt{(}e_1 \texttt{ or } e_2\texttt{)})\ r = \texttt{bool}$ se $wf_{Exp}(e_1)\ r = \texttt{bool} = wf_{Exp}(e_2)\ r$, *error* altrimenti
$wf_{Exp}(\texttt{(}e_1 \texttt{ and } e_2\texttt{)})\ r = \texttt{bool}$ se $wf_{Exp}(e_1)\ r = \texttt{bool} = wf_{Exp}(e_2)\ r$, *error* altrimenti
$wf_{Exp}(\texttt{(}e_1 \texttt{ = } e_2\texttt{)})\ r = \texttt{bool}$ se $wf_{Exp}(e_1)\ r = PT = wf_{Exp}(e_2)\ r$, *error* altrimenti
$wf_{Exp}(\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \texttt{ endif})\ r = PT$ se $wf_{Exp}(e)\ r = \texttt{bool}$ e $wf_{Exp}(e_1)\ r = PT = wf_{Exp}(e_2)\ r$, *error* altrimenti
$wf_{Exp}(\texttt{let } ds \texttt{ in } e \texttt{ endlet})\ r = wf_{Exp}(e)\ r'$ se $wf_{Decs}(ds)\ r = r'$, *error* altrimenti
$wf_{Exp}(id)\ r = r(id)$ se $r(id)$ definito, *error* altrimenti
$wf_{Exp}(id\texttt{(}e_1\texttt{, } \ldots \texttt{, } e_n\texttt{)})\ r = PT$ se
    $wf_{Exp}(e_1)\ r = PT_1, \ldots, wf_{Exp}(e_n)\ r = PT_n$, $r(id) = PT_1 \ldots PT_n \to PT$, *error* altrimenti

$wf_{Decs}(d\texttt{ ; }ds)\ r = wf_{Decs}(ds)\ r'$ se $wf_{Decs}(d)\ r = r'$, *error* altrimenti
$wf_{Decs}(id \texttt{ : } PT \texttt{ = } e)\ r = r[PT/id]$ se $wf_{Exp}(e)\ r = PT$, *error* altrimenti
$wf_{Decs}(id\texttt{(}id_1 \texttt{ : } PT_1\texttt{, } \ldots \texttt{, } id_n \texttt{ : } PT_n\texttt{)} \texttt{ : } PT \texttt{ = } e)\ r = r[PT_1 \star \ldots \star PT_n \to PT/id]$
    se $wf_{Exp}(e)\ r[PT_1/id_1 \ldots PT_n/id_n] = PT$, *error* altrimenti.

Figura 6: Definizione delle funzioni di type-checking per FL

dichiarazione cancella l'effetto delle precedenti. Si provi per esercizio a definire e/o implementare (modificando opportunamente il codice dato nel seguito) una semantica statica in cui è vietato ridichiarare un identificatore in una sequenza di dichiarazioni.
Una dichiarazione di costante $id \texttt{ : } PT \texttt{ = } e$ è corretta in r se e è un'espressione corretta di tipo PT in r e, in tal caso, produce l'ambiente ottenuto aggiornando[10] r con l'associazione tra id e PT. Analogamente, una dichiarazione di funzione $id\texttt{(}id_1 \texttt{ : } PT_1\texttt{, } \ldots \texttt{, } id_n \texttt{ : } PT_n\texttt{)} \texttt{ : } PT \texttt{ = } e$ è corretta in r se il body e della funzione è un'espressione corretta di tipo PT in r aggiornato con le associazioni di tipo per i parametri; in tal caso, produce l'ambiente ottenuto aggiornando r con l'associazione tra id e il tipo funzionale $PT_1 \star \ldots \star PT_n \to PT$. Si noti che in base alla definizione data non è ammessa ricorsione (ossia, nel body non sono ammesse chiamate alla funzione che viene dichiarata). Una definizione che invece corrisponde a ammettere ricorsione (semplice) è la seguente:

$wf_{Decs}(id\texttt{(}id_1 \texttt{ : } PT_1\texttt{, } \ldots \texttt{, } id_n \texttt{ : } PT_n\texttt{)} \texttt{ : } PT \texttt{ = } e)\ r = r[PT_1 \star \ldots \star PT_n \to PT/id]$
    se $wf_{Exp}(e)\ r[PT_1/id_1 \ldots PT_n/id_n][PT_1 \star \ldots \star PT_n \to PT/id] = PT$, *error* altrimenti.

Il codice dato nel seguito implementa ricorsione singola ma non mutua ricorsione. Si provi per esercizio a definire e/o implementare (modificando opportunamente il codice) una semantica statica alternativa in cui è ammessa mutua ricorsione in una sequenza di dichiarazioni di funzione.
Si noti infine che in base alla definizione data sopra nel body i parametri "nascondono" il nome della funzione, cioè in una dichiarazione del tipo

```
f(f : int) : int = (f + f(f))
```

la prima occorrenza di `f` nel body è corretta, mentre la seconda non lo è (questo è ciò che avviene per esempio in Caml). Si provi per esercizio a definire e/o implementare (modificando opportunamente il codice dato nel seguito) una semantica statica alternativa in cui non è ammesso utilizzare come parametro lo stesso identificatore utilizzato come nome della funzione.

1.12 Type-checker per FL

Nell'implementazione di un linguaggio, la componente che ha il compito di controllare se un programma (riconosciuto sintatticamente corretto dal parser) è staticamente corretto si chiama *type-checker* o *analizzatore statico*. In genere, sarà possibile sviluppare un type-checker in maniera guidata dalla definizione formale della semantica statica del linguaggio.

[10] Ricordiamo che $f[b/a]$ denota la funzione che vale b su a, è uguale a f altrove, vedi Def.A.2.

Per esempio, un'implementazione in Java di un type-checker per il linguaggio FL ricavata in modo molto semplice dalla definizione data precedentemente è illustrata sotto. Viene utilizzata la rappresentazione interna dei programmi introdotta in Sez.1.9. Si noti che in questo modo il type-checker risulta indipendente dalla sintassi concreta del linguaggio (dipende solo dalla sintassi astratta).
Come si vede dal codice riportato sotto, ogni funzione di type-checking per una categoria sintattica A viene implementata in Java con un metodo dichiarato nell'interfaccia corrispondente `A`; il risultato *error* viene implementato con un'eccezione. In particolare nel caso di FL si ha:

- $wf_{Prog} : Prog \rightarrow \{ok, error\}$ è implementata dal metodo

  ```
  void typeCheck() throws TypeCheckException
  ```

  dichiarato in `Prog`;

- $wf_{Exp} : Exp \rightarrow StaticEnv \rightarrow PrimType \cup \{error\}$ è implementata dal metodo

  ```
  PrimType typeCheck(StaticEnv env) throws TypeCheckException
  ```

  dichiarato in `Exp`;

- $wf_{Decs} : Decs \rightarrow StaticEnv \rightarrow StaticEnv \cup \{error\}$ è implementata dal metodo

  ```
  void typeCheck(StaticEnv env) throws TypeCheckException
  ```

  dichiarato in `Decs` (si è adottata infatti la scelta di modificare direttamente l'ambiente statico passato come parametro anzichè restituirne uno aggiornato; si veda anche l'implementazione della funzione di type-checking in `LetExp` per vedere come viene costruito un ambiente locale).

```
public interface Prog {
    PrimType typeCheck() throws TypeCheckException;
}

public class ExpProg implements Prog { ...
    public PrimType typeCheck() throws TypeCheckException {
        return exp.typeCheck(new StaticEnvClass());
    }
}

public interface Exp {
    PrimType typeCheck(StaticEnv env) throws TypeCheckException;
}

public class SumExp extends BinExp { ...
    public IntType typeCheck(StaticEnv env) throws TypeCheckException{
        if (exp1.typeCheck(env)!=IntType.CONST || exp2.typeCheck(env)!=IntType.CONST)
            throw new TypeCheckException("sum operands must be of type 'int'");
        return IntType.CONST;
    }
}

public class NumExp implements Exp { ...
    public IntType typeCheck(StaticEnv env) {
        return IntType.CONST;
    }
}

public class IfExp implements Exp { ...
    public PrimType typeCheck(StaticEnv env)throws TypeCheckException{
        PrimType thenType=thenExp.typeCheck(env);
        PrimType elseType=elseExp.typeCheck(env);
        if (ifExp.typeCheck(env)!=BoolType.CONST || thenType!=elseType)
            throw new TypeCheckException("type mismatch in if-expression");
        return thenType;
```

```
    }
}

public class IdExp implements Exp { ...
    public PrimType typeCheck(StaticEnv env)throws TypeCheckException{
        Type idType=env.apply(id);
        if (idType==null)
            throw new TypeCheckException("undeclared identifier "+id);
        else if (!(idType instanceof PrimType))
            throw new TypeCheckException("expressions must have a primitive type");
        return (PrimType) idType;
    }
}

public class FunCallExp implements Exp { ...
    public PrimType typeCheck(StaticEnv env) throws TypeCheckException{
        Type idType=env.apply(id);
        if (idType==null)
            throw new TypeCheckException("undeclared identifier "+id);
        if (!(idType instanceof FunType))
            throw new TypeCheckException(id+" must be a function");
        FunType funType=(FunType) idType;
        if (!exps.typeCheck(env).equals(funType.argsType()))
            throw new TypeCheckException("parameter types do not match");
        return funType.retType();
    }
}

public interface Decs extends Iterable<Dec> { ...
    void typeCheck(StaticEnv env) throws TypeCheckException;
}

public class DecsClass implements Decs { ...
    public void typeCheck(StaticEnv env) throws TypeCheckException {
        for(Dec d : decs)
            d.typeCheck(env) ;
    }
}

public class ConstDec implements Dec { ...
    public void typeCheck(StaticEnv env) throws TypeCheckException {
        if (exp.typeCheck(env)!=type)
            throw new TypeCheckException("type mismatch in constant declaration");
        env.update(id,type);
    }
}

public class FunDec implements Dec { ...
    public void typeCheck(StaticEnv env) throws TypeCheckException{
        StaticEnv env1=env.clone();
        FunType ft=new ArrowType(params.types(), type) ;
        env1.update(id, ft); // to allow recursion
        params.typeCheck(env1) ;
        if (exp.typeCheck(env1)!=type)
            throw new TypeCheckException("type mismatch in function declaration");
        env.update(id,ft) ;
    }
}

...
```

Gli errori statici sono stati implementati con la seguente classe di eccezioni.

```
public class TypeCheckException extends Exception{

    public TypeCheckException() {
        super("Type-checking error");
    }

    public TypeCheckException(String msg) {
        super("Type-checking error: "+msg);
    }

}
```

I tipi di FL sono stati implementati utilizzando la metodologia già vista in Sez.1.9 di rappresentazione interna di un linguaggio.

```
public interface Type { }

public interface PrimType extends Type { }

public class PrimTypesClass implements PrimTypes {
    private List<PrimType> types = new ArrayList<PrimType>() ;

    public void add(PrimType type) {
        types.add(type);
    }

    public Iterator<PrimType> iterator() {
        return types.iterator() ;
    }

    public boolean equals(Object that) {
        return that instanceof PrimTypesClass && types.equals( ((PrimTypesClass)that).types ) ;
    }

    public int hashCode() {
        return types.hashCode() ;
    }
}

public interface FunType extends Type {
    PrimTypes argsType();
    PrimType retType();
}

public class ArrowType implements FunType {
    private PrimType type;
    private PrimTypes types;

    public ArrowType(PrimTypes types, PrimType type) {
        this.type=type;
        this.types=types;
    }

    public PrimTypes argsType() {
        return types;
    }

    public PrimType retType() {
        return type;
    }
}
```

```
public final class IntType implements PrimType{
    public static final IntType CONST=new IntType();

    private IntType() {}

    public String toString(){
        return "int";
    }
}

public final class BoolType implements PrimType {
    public static final BoolType CONST=new BoolType();

    private BoolType() {}

    public String toString(){
        return "bool";
    }
}
```

Infine, l'insieme *StaticEnv* degli ambienti statici è stato implementato nel modo seguente:

```
public interface StaticEnv extends Cloneable {
    Type apply(Ident id);
    void update(Ident id,Type t);
    StaticEnv clone();
}

public class StaticEnvClass implements StaticEnv {
    private Map<Ident, Type> id2type = new Hashtable<Ident, Type>() ;

    public Type apply(Ident id){
        return id2type.get(id);
    }

    public void update(Ident id, Type t){
        id2type.put(id, t);
    }

    public StaticEnv clone() {
        try    {
            return (StaticEnv)super.clone() ;
        } catch (CloneNotSupportedException e) {
            return null ;  // cannot happen
        }
    }
}
```

1.13 Semantica dinamica

Per *semantica* di un linguaggio, naturale o formale, intendiamo il *significato* delle frasi del linguaggio. In particolare, per i L. di P., si tratta di specificare quale è il comportamento in fase di esecuzione dei programmi nel linguaggio. Si parla quindi anche di *semantica dinamica* per evidenziare che si descrivono proprietà dei programmi osservabili solo in fase di esecuzione.
Diamo, a titolo di esempio, una descrizione formale della semantica dinamica di FL e la sua implementazione in Java.
Analogamente a quanto già visto per la semantica statica, l'idea intuitiva è che dobbiamo dire quale è il risultato finale della valutazione dei programmi (espressioni chiuse) del linguaggio. Formalmente, questo può essere modellato da una funzione

$$\llbracket - \rrbracket_{Prog} : Prog \rightarrow PrimVal$$
$$PrimVal = \mathbb{Z} \cup \{T, F\}$$

che dato un programma p restituisce il risultato della sua valutazione (un intero o un booleano).

Naturalmente però il risultato della valutazione di un programma dipende da come sono valutate le sue sottocomponenti. Quindi, per definire questa funzione avremo bisogno di definire, per ogni categoria sintattica, il risultato della valutazione dei termini del linguaggio corrispondente, quindi di definire un analoga funzione $[\![-]\!]_A$ per ogni non terminale A. Ad esempio, nel caso di FL, dovremo definire i valori delle dichiarazioni, delle espressioni, etc.
Si noti però che, mentre per un programma p del linguaggio ha senso porsi la domanda "quale è il valore di p", per un generico termine tale domanda non ha senso, in quanto, analogamente a quanto visto per la correttezza, il valore dipende dal contesto in cui questo si trova; ad esempio, in FL il valore dell'espressione `((x + y) = 0)` dipenderà dai valori associati al momento della dichiarazione agli identificatori di costante `x` e `y`.
Di conseguenza, le funzioni $[\![-]\!]_A$ in generale potranno avere un argomento aggiuntivo (detto *ambiente dinamico*) che corrisponde alle informazioni contestuali necessarie per poter valutare un termine.
Nel caso di FL, l'informazione necessaria consiste nel sapere quali identificatori di costante e di funzione sono stati dichiarati e con quale valore. Quindi definiremo gli ambienti dinamici nel modo seguente:

$DynEnv = \textsc{Id} \rightarrow Val$
$Val = PrimVal \cup FunVal$
$FunVal = \textsc{Id}^{\star} \times Exp \times DynEnv$

I valori *Val* del linguaggio si distinguono in valori primitivi e funzionali. Un valore primitivo è un possibile valore per un'espressione, quindi un intero o un booleano. Un valore funzionale è il valore che viene associato nell'ambiente a un identificatore di funzione id al momento della sua dichiarazione. Nella definizione qui proposta, assumiamo di associare a id un'informazione di tipo sintattico, cioè la lista degli identificatori usati come parametri e il body presenti nella dichiarazione di id e un ambiente locale che tiene traccia dei valori associati agli identificatori al momento della dichiarazione di id (questo ambiente è necessario per valutare eventuali identificatori diversi dai parametri presenti nel body della funzione, vedi clausola semantica per la chiamata di funzione). Un approccio alternativo è quello di associare a id un'informazione di tipo semantico, analogamente a quanto fatto per gli identificatori di costante, cioè una funzione. La prima versione corrisponde a ciò che avviene a livello di implementazione.
Ad esempio, la valutazione della precedente espressione `((x + y) = 0)` restituirà , in ogni ambiente *r* in cui `x` e `y` sono definiti, vero o falso a seconda che la somma dei loro valori sia o meno uguale a zero.
Diamo ora la semantica dinamica formale completa del linguaggio FL.
Avremo le seguenti funzioni di valutazione.

- $[\![-]\!]_{Prog}: Prog \rightarrow PrimVal$ per i programmi (espressioni chiuse): la valutazione di un programma produce un valore primitivo;
- $[\![-]\!]_{Exp}: Exp \rightarrow DynEnv \rightarrow PrimVal$ per le espressioni: la valutazione di un'espressione, in un ambiente che assegna valori agli identificatori, produce un valore primitivo;
- $[\![-]\!]_{Decs}: Decs \rightarrow DynEnv \rightarrow DynEnv$ per le sequenze di dichiarazioni: l'elaborazione di una sequenza di dichiarazioni, in un ambiente che assegna valori agli identificatori, produce un nuovo ambiente in cui sono state aggiunte associazioni per gli identificatori dichiarati.

La semantica dinamica del linguaggio FL è definita formalmente in Fig.7.
La valutazione di un programma (espressione chiusa) p consiste nella valutazione di p come espressione nell'ambiente vuoto.
La valutazione di un'espressione somma, prodotto, congiunzione e disgiunzione logica, e uguaglianza, consiste nella valutazione delle sottocomponenti e poi nell'applicazione ai valori ottenuti del corrispondente operatore. Analogamente, la valutazione di una costante intera o booleana produce il corrispondente valore primitivo.
La valutazione di un'espressione `if` e `then` e_1 `else` e_2 `endif` consiste nella valutazione dell'espressione booleana e e poi, a seconda del valore vero o falso ottenuto, nella valutazione di e_1 o e_2.
La valutazione di un'espressione `let` ds `in` e `endlet` consiste nella valutazione dell'espressione e nell'ambiente ottenuto aggiornando quello corrente con le nuove associazioni prodotte dalle dichiarazioni in ds.
La valutazione di un identificatore di costante restituisce il valore associato a tale identificatore nell'ambiente corrente.
La valutazione di una chiamata di funzione consiste anzitutto nella valutazione degli argomenti; ottenuti i corrispondenti valori $v_1, \ldots, v_n$, si valuta il body della funzione nell'ambiente ottenuto modificando quello al momento della dichiarazione con le associazioni ai parametri dei valori $v_1, \ldots, v_n$.
L'elaborazione di di una sequenza di dichiarazioni d `;` ds consiste nell'elaborazione di ds nell'ambiente ottenuto aggiornando quello corrente con la nuova associazione prodotta dalla dichiarazione d. In altre parole, le dichiarazioni vengono elaborate nell'ordine.
L'elaborazione di una dichiarazione di costante id : PT = e aggiorna l'ambiente corrente aggiungendo l'associazione tra id ed il risultato della valutazione di e.

$[\![p]\!]_{Prog} = [\![p]\!]_{Exp}\,\emptyset$

$[\![\,\texttt{(}e_1\ \texttt{+}\ e_2\texttt{)}\,]\!]_{Exp}\, r = [\![e_1]\!]_{Exp}\, r +^{\mathbb{Z}} [\![e_2]\!]_{Exp}\, r$
$[\![\,\texttt{(}e_1\ \texttt{*}\ e_2\texttt{)}\,]\!]_{Exp}\, r = [\![e_1]\!]_{Exp}\, r \times^{\mathbb{Z}} [\![e_2]\!]_{Exp}\, r$
$[\![n]\!]_{Exp}\, r = n^{\mathbb{Z}}$
$[\![\texttt{true}]\!]_{Exp}\, r = T$
$[\![\texttt{false}]\!]_{Exp}\, r = F$
$[\![\,\texttt{(}e_1\ \texttt{or}\ e_2\texttt{)}\,]\!]_{Exp}\, r = [\![e_1]\!]_{Exp}\, r \vee^{\mathbb{B}} [\![e_2]\!]_{Exp}\, r$
$[\![\,\texttt{(}e_1\ \texttt{and}\ e_2\texttt{)}\,]\!]_{Exp}\, r = [\![e_1]\!]_{Exp}\, r \wedge^{\mathbb{B}} [\![e_2]\!]_{Exp}\, r$
$[\![\,\texttt{(}e_1\ \texttt{=}\ e_2\texttt{)}\,]\!]_{Exp}\, r = [\![e_1]\!]_{Exp}\, r = [\![e_2]\!]_{Exp}\, r$
$[\![\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{endif}]\!]_{Exp}\, r = [\![e_1]\!]_{Exp}\, r$ se $[\![e]\!]_{Exp}\, r = T$, $[\![e_2]\!]_{Exp}\, r$ se $[\![e]\!]_{Exp}\, r = F$
$[\![\texttt{let}\ ds\ \texttt{in}\ e\ \texttt{endlet}]\!]_{Exp}\, r = [\![e]\!]_{Exp}([\![ds]\!]_{Decs}\, r)$
$[\![id]\!]_{Exp}\, r = r(id)$
$[\![id\,\texttt{(}e_1\texttt{,}\ \ldots\texttt{,}\, e_n\texttt{)}\,]\!]_{Exp}\, r = [\![e]\!]_{Exp}\, r_{id}[v_1/id_1 \ldots v_n/id_n]$ se $[\![e_1]\!]_{Exp}\, r = v_1, \ldots, [\![e_n]\!]_{Exp}\, r = v_n$, $r(id) = (id_1 \ldots id_n, e, r_{id})$

$[\![d\ \texttt{;}\ ds]\!]_{Decs}\, r = [\![ds]\!]_{Decs}([\![d]\!]_{Decs}\, r)$
$[\![id\ \texttt{:}\ PT\ \texttt{=}\ e]\!]_{Decs}\, r = r[[\![e]\!]_{Exp}\, r/id]$
$[\![id\,\texttt{(}id_1\ \texttt{:}\ PT_1\texttt{,}\ \ldots\texttt{,}\, id_n\ \texttt{:}\ PT_n\texttt{)}\ \texttt{:}\ PT\ \texttt{=}\ e]\!]_{Decs}\, r = r[(id_1 \ldots id_n, e, r)/id]$

Figura 7: Definizione delle funzioni di valutazione per FL

Analogamente, l'elaborazione di una dichiarazione di funzione $id\,\texttt{(}id_1\ \texttt{:}\ PT_1\texttt{,}\ \ldots\texttt{,}\, id_n\ \texttt{:}\ PT_n\texttt{)}\ \texttt{:}\ PT\ \texttt{=}\ e$ aggiorna l'ambiente aggiungendo l'associazione tra id e la tripla $(id_1 \ldots id_n, e, r)id$ che consiste della sequenza dei parametri, il body e l'ambiente al momento della dichiarazione.
Si noti che questa semantica della dichiarazione di funzione corrisponde a scegliere un *binding statico* per gli identificatori globali utilizzati all'interno del corpo della funzione; ossia, in tutte le chiamate della funzione questi identificatori continuano a fare riferimento al valore che avevano al momento della dichiarazione (si veda l'esempio a pagina 81). Questa è la scelta della maggior parte dei linguaggi di programmazione; solo in pochi casi (per esempio, nel Lisp) si ha invece un *binding dinamico* in cui, per ogni chiamata della funzione, gli identificatori globali fanno riferimento al valore che hanno al momento della chiamata.
Si provi per esercizio a definire e/o implementare (modificando opportunamente il codice dato nel seguito) una semantica con binding dinamico per gli identificatori globali nelle dichiarazioni di funzione.

1.14 Interprete per FL

Nell'implementazione di un linguaggio, la componente che ha il compito di simulare l'esecuzione di un programma è l'*interprete* propriamente detto. In genere, sarà possibile sviluppare un interprete in maniera guidata dalla definizione formale della semantica dinamica del linguaggio.
Ad esempio, un'implementazione in Java di un interprete per il linguaggio FL ricavata in modo molto semplice dalla definizione data precedentemente è illustrata sotto. Viene utilizzata la rappresentazione interna dei programmi introdotta in Sez.1.9. Si noti che in questo modo l'interprete risulta indipendente dalla sintassi concreta del linguaggio (dipende solo dalla sintassi astratta).
Come si vede dal codice riportato sotto, ogni funzione di valutazione per una categoria sintattica *A* viene implementata in Java con un metodo dichiarato nell'interfaccia corrispondente `A`; il caso in cui il risultato della funzione di valutazione sia indefinito (errore dinamico) viene implementato con un'eccezione (in realtà nel caso di FL non ci sono errori dinamici, ma l'eccezione è stata comunque prevista per permettere eventuali estensioni del linguaggio). In particolare nel caso di FL si ha:

- $[\![-]\!]_{Prog}$: *Prog* → *PrimVal* è implementata dal metodo

  ```
  PrimValue eval()throws RunException
  ```

  dichiarato in `Prog`;

- $[\![-]\!]_{Exp}$: *Exp* → *DynEnv* → *PrimVal* è implementata dal metodo

  ```
  PrimValue eval(DynamicEnv env)throws RunException
  ```

dichiarato in `Exp`;

- $[\![-]\!]_{Decs}: Decs \rightarrow DynEnv \rightarrow DynEnv$ è implementata dal metodo

  ```
  void eval(DynamicEnv env)throws RunException
  ```

  dichiarato in `Decs` (si è adottata infatti la scelta di modificare direttamente l'ambiente dinamico passato come parametro anzichè restituirne uno aggiornato; si veda anche l'implementazione della funzione di valutazione in `LetExp` per vedere come viene costruito un ambiente locale).

```
public interface Prog { ...
    PrimValue eval() throws RunException;
}

public class ExpProg implements Prog { ...
    public PrimValue eval() throws RunException {
        return exp.eval(new DynamicEnvClass());
    }
}

public interface Exp { ...
    PrimValue eval(DynamicEnv env) throws RunException;
}

public class SumExp extends BinExp { ...
    public IntValue eval(DynamicEnv env)throws RunException{
        return new IntValue(exp1.eval(env).asInt() + exp2.eval(env).asInt());
    }
}

public class NumExp implements Exp { ...
    public IntValue eval(DynamicEnv env) {
        return nval ;
    }
}

public class TrueExp implements Exp { ...
    public BoolValue eval(DynamicEnv env) {
        return new BoolValue(true);
    }
}

public class OrExp extends BinExp { ...
    public BoolValue eval(DynamicEnv env) throws RunException {
        return new BoolValue(exp1.eval(env).asBoolean() || exp2.eval(env).asBoolean()) ;
    }
}

public class IfExp implements Exp { ...
    public PrimValue eval(DynamicEnv env) throws RunException{
        return ifExp.eval(env).asBoolean() ? thenExp.eval(env) : elseExp.eval(env) ;
    }
}

public class LetExp implements Exp { ...
    public PrimValue eval(DynamicEnv env)throws RunException {
        DynamicEnv env1=env.clone();
        decs.eval(env1);
        return exp.eval(env1);
    }
}
```

```
public class FunCallExp implements Exp { ...
    public PrimValue eval(DynamicEnv env) throws RunException{
        FunValue f = (FunValue) env.apply(id);
        return f.apply(exps.eval(env));
    }
}

public class FunDec implements Dec { ...
    public void eval(DynamicEnv env) throws RunException {
        FunValue f=new FunValue(params.ids(),exp,env);
        env.update(id,f); // to allow recursion
    }
}
```

Gli errori dinamici (che in realtà in FL non ci sono, ma potrebbero essere introdotti da estensioni del linguaggio) sono stati implementati con la seguente classe di eccezioni.

```
public class RunException extends Exception{

    public RunException(){
        super("Run-time error");
    }

    public RunException(String msg){
        super("Run-time error: "+msg);
    }

}
```

I valori di FL sono stati implementati usando la metodologia di implementazione nel paradigma object-oriented di tipi unione, vedi Sez.2.3.3.

```
interface Value {}

interface PrimValue extends Value {}

public final class IntValue implements PrimValue{
    private int i;

    public IntValue(int i) {
        this.i=i ;
    }

    public int asInt() {
        return i ;
    }

    public boolean asBoolean() throws RunException {
        throw new RunException("Cannot convert "+i+" to boolean") ;
    }

    public String toString() {
        return Integer.toString(i) ;
    }

    public boolean equals(Object that) {
        return that instanceof IntValue && ((IntValue)that).i==i ;
    }

    public int hashCode() {
        return i ;
    }
```

```
}

public final class BoolValue implements PrimValue{
    private boolean b;

    public BoolValue(boolean b){
        this.b=b;
    }

    public int asInt() throws RunException {
        throw new RunException("Cannot convert "+b+" to int") ;
    }

    public boolean asBoolean() {
        return b ;
    }

    public String toString(){
        return Boolean.toString(b);
    }

    public boolean equals(Object that) {
        return that instanceof BoolValue && ((BoolValue)that).b==b ;
    }

    public int hashCode() {
        return b ? 1:0 ;
    }
}

public class FunValue implements Value {
    private Idents ids;
    private Exp body;
    private DynamicEnv env;

    public FunValue(Idents ids,Exp body,DynamicEnv env) {
        this.ids=ids;
        this.body=body;
        this.env=env;
    }

    public PrimValue apply(PrimValues values) throws RunException {
        DynamicEnv env1=env.clone() ;
        Iterator<PrimValue> vs = values.iterator() ;
        for(Ident id : ids)
            env1.update(id, vs.next()) ;
        return body.eval(env1);
    }
}

public interface PrimValues extends Iterable<PrimValue> {
    void add(PrimValue val);
}

public class PrimValuesClass implements PrimValues {
    private List<PrimValue> values = new ArrayList<PrimValue>() ;

    public void add(PrimValue val){
        values.add(val);
    }

    public Iterator<PrimValue> iterator() {
```

```
        return values.iterator() ;
    }
}
```

Infine, l'insieme *DynEnv* degli ambienti dinamici è stato implementato nel modo seguente.

```
public interface DynamicEnv extends Cloneable {
    Value apply(Ident id);
    void update(Ident id, Value v);
    DynamicEnv clone();
}

public class DynamicEnvClass implements DynamicEnv {
    private Hashtable<Ident, Value> values = new Hashtable<Ident, Value>() ;

    public Value apply(Ident id) {
        return values.get(id);
    }

    public void update(Ident id, Value v) {
        values.put(id, v);
    }

    public DynamicEnv clone() {
        try    {
            return (DynamicEnv)super.clone() ;
        } catch (CloneNotSupportedException e) {
            return null ;  // cannot happen
        }
    }
}
```

2 Paradigma Object-Oriented

In questo capitolo illustreremo il paradigma di programmazione object-oriented utilizzando come esempio il linguaggio Java. Presenteremo inoltre alcune altre caratteristiche salienti di Java come linguaggio di programmazione.Per il linguaggio Java i testi di riferimento sono [1] e [4].

2.1 Introduzione

Un tentativo di definizione È difficile dare una definizione precisa di "paradigma object-oriented" (il termine è alquanto abusato). Noi individueremo tre aspetti principali:

Paradigma computazionale "a oggetti" L'idea è che le unità di calcolo non sono funzioni, o procedure se si pensa al caso imperativo, ma "oggetti", cioè l'enfasi è su "gruppi di algoritmi correlati" anzichè su singoli algoritmi; questo punto di vista è ortogonale alla nozione di inheritance, e perfino a quella di stato, in quanto può essere applicato a oggetti puramente funzionali. Usualmente però nei linguaggi reali gli oggetti sono modificabili, cioè si utilizza un paradigma a oggetti imperativo. Questo corrisponde a una visione di "oggetto" che assomiglia agli oggetti del mondo reale, che sono entità dotate di un'identità permanente nel tempo, il cui stato può cambiare, e che offrono all'esterno un'interfaccia di operazioni possibili con cui li si può manipolare. Un linguaggio il cui modello computazionale è basato su oggetti è detto *object-based*, e non necessita della nozione di classe e inheritance. Inizialmente i linguaggi object-based senza classi hanno avuto importanza come modello computazionale sottostante ai linguaggi object-oriented. In tempi più recenti hanno avuto successo linguaggi di scripting object-based come Javascript.

Paradigma organizzativo In questo senso il paradigma object-oriented può essere visto come una metodologia di organizzazione del software, secondo la quale i moduli corrispondono alle classi (cioè schemi di oggetti intesi nel senso detto al punto precedente) e nuovi moduli possono essere definiti a partire da altri attraverso un'operazione di estensione che prevede anche la possibilità di modificare (ridefinire) delle componenti (*inheritance*). Supponiamo, per esempio, di avere a disposizione un modulo che implementa le liste di interi e di voler scrivere un modulo che implementi gli insiemi di

interi (liste senza ripetizione), quindi modificare per esempio l'operazione di aggiunta di un elemento in modo tale che lo aggiunga solo se non è presente nella lista. Con un linguaggio di tipo tradizionale siamo obbligati a "ricopiare" tutto il codice modificandone delle parti. L'approccio object-oriented offre un supporto linguistico per ottenere lo stesso risultato.

Binding dinamico Questo aspetto è nei linguaggi object-oriented accoppiato al precedente (inheritance). Nei linguaggi è possibile talvolta utilizzare lo stesso nome per denotare oggetti semantici di tipo diverso; questa situazione si chiama *overloading* (= sovraccarico) del nome. Si noti la differenza tra funzione overloaded (una definizione diversa per ogni tipo del simbolo) e funzione *polimorfa* (la definizione è unica ma la funzione può essere applicata ad argomenti di diversi tipi).

Nei L. di P. l'overloading può essere comodo, perchè esistono molte situazioni in cui risulta naturale utilizzare lo stesso nome per operazioni su tipi diversi, ma che sono intuitivamente analoghe. Si pensi per esempio al simbolo di uguaglianza. Molti L. di P. permettono quindi forme di overloading; questo causa un problema interpretativo, in quanto occorrerà effettuare la cosiddetta *risoluzione dell'overloading* cioè determinare, in base al contesto, quale definizione del nome va effettivamente considerata. Nei L. di P. tradizionali la risoluzione dell'overloading è *statica* cioè avviene prima dell'esecuzione del programma. Questa forma di overloading si riduce quindi nella sostanza a una comoda possibilità di abbreviazione, che consente al programmatore di utilizzare un unico simbolo invece di dover inventare un nome diverso per ogni contesto di tipo possibile; l'algoritmo di risoluzione dell'overloading accetta in entrata il codice con simboli overloaded e produce in uscita un codice non overloaded in cui a ogni simbolo overloaded è stata associata l'informazione di tipo necessaria per disambiguarlo.

Nei linguaggi object-oriented invece il meccanismo di inheritance con ridefinizione illustrato sopra è accoppiato con una risoluzione dinamica; per alcuni simboli (quelli dei metodi) la risoluzione avviene al momento dell'esecuzione. Tale risoluzione dinamica permette in sostanza di avere delle operazioni la cui definizione dipende dal tipo dell'argomento.

Motivazioni Le principali motivazioni all'introduzione dell'approccio object-oriented sono state storicamente le seguenti.

Riusabilità ed estendibilità del software Nel corso dell'evoluzione dei L. di P. ci si è resi conto che era importantissimo permettere l'estendibilità e riusabilità del software; attraverso degli studi statistici, si era infatti scoperto che solo una piccola parte del software sviluppato è veramente "nuova", mentre per la maggior parte del tempo lo sviluppo del software è un'attività molto ripetitiva che consiste nell'adattare e utilizzare nel modo migliore schemi noti.

Influsso del paradigma ADT L'approccio object-oriented ha ripreso, se pure in maniera diversa, molti principi del paradigma cosiddetto "Abstract Data Types" (tipi di dato astratti). Questo paradigma, utilizzato soprattutto a livello dei linguaggi di specifica ma anche in alcuni L. di P., consiste nell'organizzazione del software in moduli corrispondenti a *tipi di dato*, cioè strutture formate da diversi insiemi di valori più le operazioni per manipolarli (la nozione formale corrispondente è quella di algebra, vedi Def.A.8). Ogni modulo corrispondente a un tipo di dato esporta solo la sua interfaccia operazionale (formalizzata dalla nozione di segnatura, vedi Def.A.7) mentre l'implementazione è nascosta (incapsulazione).

Metafora del mondo reale Il paradigma computazionale a oggetti si è proposto come paradigma "simile al mondo reale". L'approccio object-oriented è infatti nato, con il linguaggio Smalltalk, contemporaneamente ai primi sistemi operativi con "interfaccie iconiche" (Macintosh). Infatti tali interfacce sono basate sullo stesso paradigma di "oggetti che rispondono a messaggi" (in questo caso, azioni dell'utente) su cui si basa la programmazione object-oriented.

Note storiche

- L'antenato riconosciuto dei linguaggi object-oriented è Simula 67 (Dahl e Nygaard, Università di Oslo, fine anni '60): un linguaggio per simulazioni, che ha introdotto la nozione di classe.
- Il capostipite dei linguaggi object-oriented moderni è Smalltalk (le prime idee sono dovute ad Alan Key, e sono state sviluppate contemporaneamente ai sistemi operativi di nuova concezione; la prima versione stabile è dovuta a Goldberg e Robson, Xerox PARC, nel 1980): è con questo linguaggio che viene introdotta la terminologia oggi diffusa. È un linguaggio interpretato e non ha type checking statico.
- Altri due linguaggi importanti sono Eiffel (Meyer 1985), progettato con l'obiettivo di trasportare i concetti dell'object-oriented in un contesto di ingegneria del software, e C++, un'estensione object-oriented di C che segue però un paradigma misto, conservando caratteristiche imperative (per esempio i puntatori).
- Oggi esistono moltissimi linguaggi object-oriented. Ne citiamo alcuni.
  - Versioni object-oriented di linguaggi preesistenti, come: varie estensioni del Lisp (Loops della Xerox, Flavors del MIT, Ceyx dell'INRIA), confluite nel 1993 in CLOS (Common Lisp Object System); le estensioni del C (il già citato C++, di B. Stroustrup, AT&T, e Objective C di B. Cox); Ada 95; Borland Pascal; Delphi; C#.

- Altri: Oberon (di N. Wirth, successore di Modula-2); Modula-3 (DEC); Trellis (DEC), che concilia genericità e inheritance multipla; Sather (sottoinsieme di Eiffel); Beta (discendente diretto di Simula); Self (object-based, con la nozione di delegation).
- Esistono inoltre altre applicazioni importanti dell'approccio object-oriented: ad esempio nelle basi di dati e nelle metodologie di progettazione.

2.2 Oggetti e Classi

2.2.1 Classi

Vediamo subito un esempio di dichiarazione di classe in Java:

```
class Rectangle {
  int length,width;
  void setLength (int l) {length = l;}
  void setWidth (int w) {width = w;}
  int area () {return width * length;}
  void print ()  {System.out.println("I am a rectangle:  = " + length + ", " + width);}
}
```

Come illustrato dall'esempio, una classe definisce uno schema di oggetti. Le componenti di una definizione di classe sono essenzialmente di due tipi: le variabili che compongono lo stato, per cui esistono diversi nomi a seconda dei linguaggi: *attributi*, *campi*, *variabili di istanza* (*instance variables*), e le operazioni, dette *metodi*. Nell'esempio, i campi sono `length` e `width`, i metodi sono `setLength`, `setWidth`, `area` e `print`.
La classe `Rectangle` introduce il *tipo oggetto* `Rectangle` e fornisce uno *schema* per creare oggetti di tipo `Rectangle`, detti *istanze* della classe.
Vediamo un esempio di utilizzo.

```
  Rectangle r;
  r = new Rectangle();
  r.setLength(2);r.setWidth(3);r.print();
```

L' esecuzione di questo frammento di codice produrrà:

```
I am a rectangle:  = 2, 3
```

Nella prima linea di codice viene dichiarata una variabile di tipo `Rectangle`, che viene poi inizializzata con una nuova istanza della classe[11]. Ogni oggetto di classe `Rectangle` è dotato di un proprio stato (i campi `length` e `width`). Il termine *variabili di istanza* fa proprio riferimento al fatto che esiste una copia di queste variabili per ogni istanza della classe.
In generale, lo stato non dovrebbe essere visibile all'esterno, ma dovrebbe essere modificato e ispezionato solo tramite i metodi. In alcuni linguaggi object-oriented (per esempio in Smalltalk) gli attributi sono sempre nascosti. In altri la scelta è lasciata al programmatore (per esempio in Java ciò si può ottenere dichiarando i campi `private`).
Un oggetto ha poi un'*interfaccia operazionale*, che descrive l'interazione dell'oggetto col mondo esterno (i nomi e tipi dei metodi).
La "dot notation" `r.setLength(2)` suggerisce che il metodo invocato "appartiene" all'oggetto "ricevitore" `r`. Si noti infatti che i metodi corrispondono alle funzioni o procedure in altri linguaggi, ma in questo caso vi è un argomento (il ricevitore) che ha un ruolo privilegiato. Infatti la definizione del metodo da invocare si trova proprio nella classe di tale oggetto. Inoltre, come vedremo nella Sez.2.3.2, il tipo dinamico del ricevitore determina la versione del metodo da invocare in caso di ridefinizione.
Tutti gli oggetti della classe `Rectangle` hanno la stessa interfaccia operazionale e lo stesso comportamento (cioè, la stessa implementazione dei metodi).

2.2.2 Uguaglianza tra oggetti e copia

Ogni oggetto ha una propria *identità* che si conserva durante l'esecuzione.

```
Rectangle r1,r2,r3; r1=new Rectangle(); r2=new Rectangle(); r3=r1;
```

[11] Utilizzando il *costruttore di default* che inizializza i campi `length` e `width` a zero, come vedremo più in dettaglio in Sez.2.4.4.

Dopo l'esecuzione di questo frammento di codice, esistono due oggetti di tipo `Rectangle`; siano essi r_1, r_2. Le variabili `r1` e `r3` denotano il rettangolo r_1, mentre la variabile `r2` denota il rettangolo r_2. I due rettangoli hanno lo stesso stato (in entrambi i casi i campi `length` e `width` sono stati inizializzati a zero), ma identità diverse (ossia vale `r1 != r2`). Vale invece l'uguaglianza `r1 == r3` (si dice che `r3` è un *alias* di `r1`, vedi Sez.2.2.3).
Informalmente, il modello dell'esecuzione di un programma in un linguaggio object-oriented è un universo di oggetti in evoluzione. A ogni istante dell'esecuzione del programma vi è un certo numero di oggetti esistenti. Ognuno di essi ha un'identità e uno stato interno. L'universo può cambiare per le seguenti tre ragioni:

- creazione di nuovi oggetti,
- modifica dello stato di oggetti esistenti,
- cancellazione di oggetti esistenti. Quest'ultima operazione, in genere, nei linguaggi object-oriented è affidata a un garbage collector automatico.

Nei linguaggi object-oriented si distinguono quindi due diverse nozioni di uguaglianza. La prima è espressa in Java dal simbolo == ed è predefinita. Due espressioni sono uguali in questo senso se e solo se denotano *lo stesso* oggetto, come `r1` e `r3` nell'esempio; dal punto di vista implementativo, `r1` e `r3` contengono lo stesso indirizzo. Infatti in genere nell'implementazione dei linguaggi object-oriented si utilizza una tavola degli oggetti esistenti, detta *object table*, dove ogni oggetto ha un identificatore unico, detto *object identifier* (*oid*). Quando si crea un nuovo oggetto viene allocata una nuova riga della tavola.
La seconda uguaglianza corrisponde invece a una qualche uguaglianza degli stati, e deve essere implementata dall'utente. Per esempio, nella classe `Rectangle` possiamo aggiungere un metodo `equals` come segue.

```
class Rectangle{
  ...
  boolean equals (Rectangle r) {
    return length == r.length && width == r.width;
  }
}
```

Tra le diverse uguaglianze degli stati che si possono considerare, due particolarmente interessanti sono le seguenti:

1. due oggetti sono uguali se i loro campi corrispondenti sono *identici*, cioè denotano lo stesso valore (se di tipo primitivo, cioè non oggetto) o oggetto;
2. due oggetti sono uguali se i loro campi corrispondenti denotano lo stesso valore (se di tipo primitivo) o oggetti (ricorsivamente) uguali.

Per esempio considerando una classe `RectanglePair` le cui istanze modellano coppie di rettangoli, le due uguaglianze possono essere implementate nel modo seguente.

```
class RectanglePair{
   Rectangle first, second;
  ...
  boolean equals1 (RectanglePair rp) {
    return first == rp.first && second == rp.second;
  }
  boolean equals2 (RectanglePair rp) {
    return first.equals(rp.first) && second.equals(rp.second);
  }
}
```

Si noti che le due definizioni coincidono nel caso di oggetti i cui campi sono tutti di tipi primitivi (come le istanze di `Rectangle`).
Un'analoga distinzione vale per la *copia* (vedi dopo).
♠ Diciamo che un oggetto o è sottocomponente di un oggetto o' se o' ha un campo che denota (un oggetto che ha come sottocomponente) l'oggetto o. Si noti che la seconda uguaglianza sopra risulta ben definita (dal punto di vista implementativo, terminante) solo nei casi in cui la relazione di sottocomponente è aciclica. In caso contrario, occorre dare un tipo diverso di definizione (dal punto di vista implementativo, dare un algoritmo che tenga traccia degli oggetti già esaminati). Analoghe considerazioni valgono per la copia, vedi sotto.
Nei linguaggi object-oriented esiste talvolta un'operazione di uguaglianza predefinita. In Java per esempio `boolean equals (Object)` è un metodo della classe `Object` (la classe più generale) che come implementazione di default ha semplicemente ==, vedi Sez.2.4.12. Tale metodo può essere ridefinito in modo da ottenere altre uguaglianze desiderate.

Si noti anche, osservando l'esempio precedente, che in un linguaggio object-oriented il fatto che i metodi hanno sempre un argomento privilegiato, il primo (il ricevitore del messaggio), costringe ad esprimere in modo "asimmetrico" anche operazioni a due argomenti intuitivamente "simmetriche", come l'uguaglianza. Questo aspetto dei linguaggi object-oriented viene talvolta criticato; vedremo in seguito che in alcuni linguaggi, per esempio Java, è possibile definire operazioni senza argomenti privilegiati utilizzando i cosiddetti *metodi di classe* (vedi Sez.2.2.5).
L'assegnazione a variabili di tipo oggetto ha un effetto simile a un'assegnazione a variabili di tipo puntatore in un linguaggio imperativo. Dopo l'assegnazione

```
r3=r1;
```

`r3` e `r1` denotano lo stesso oggetto[12]. Se si vuole invece assegnare a una variabile una *copia* di un oggetto esistente, occorre utilizzare un opportuno metodo:

```
r3= r1.copy();
```

Dopo quest'assegnazione, `r3` e `r1` non denotano lo stesso oggetto (non vale `r3 == r1`); comunque, se il metodo di copia è stato definito in accordo con la definizione data per l'uguaglianza, vale `r1.equals(r3)`.
Una possibile definizione del metodo `copy` che corrisponde alla definizione di `equals` data sopra è la seguente:

```
Rectangle copy () {
  Rectangle r = new Rectangle ();
  r.length = length;
  r.width = width;
  return r;
}
```

Una definizione più compatta può essere data utilizzando un *costruttore*, vedi Sez.2.2.7.
Nei linguaggi object-oriented esiste in genere un'operazione di copia predefinita. In Java per esempio `Object clone()` è un metodo della classe `Object` (la classe più generale) che effettua una *shallow copy* dell'oggetto su cui è invocato, vedi Sez.2.4.12: restituisce cioè un nuovo oggetto i cui campi denotano gli stessi oggetti denotati dai campi dell'oggetto ricevitore (ossia, una shallow copy è uguale nel senso (1) al ricevitore).
Una *deep copy* di un oggetto è invece un nuovo oggetto i cui campi sono (ricorsivamente) deep copy dei campi corrispondenti dell'oggetto ricevitore (ossia, una deep copy è uguale nel senso (2) al ricevitore). Naturalmente la deep copy coincide con la shallow copy nel caso di oggetti in cui tutti i campi sono di tipi primitivi, come le istanze della classe `Rectangle`. La deep copy, se desiderata, va in genere implementata dal programmatore; in Java, per esempio, ridefinendo opportunamente il metodo `clone`. Se si sceglie questa strada occorre ricordare però che il metodo `clone` restituisce un `Object` e quindi utilizzare il casting (vedi Sez.2.4.8) per ottenere oggetti del tipo desiderato.
Ricapitolando quanto visto finora, la nozione di classe nei linguaggi object-oriented può essere caratterizzata come segue.

- Una classe definisce uno schema di oggetti (detti istanze della classe).
- Gli oggetti della stessa classe condividono sia l'interfaccia operazionale, sia l'implementazione.
- Gli oggetti sono "valori di prima classe", cioè sono manipolabili come usuali valori.
- Le istanze di una classe sono create dinamicamente.

2.2.3 Ricorsione tra classi, confronto tra oggetti e puntatori

Si dice che una classe `A` è *cliente* di (o *usa*) una classe `B` se il codice di `A` utilizza `B` come tipo (quindi dichiara campi, variabili locali o parametri/risultato di metodi di tipo `B`) o come generatore di oggetti (quindi crea istanze di `B`).
Come già visto, un oggetto può infatti avere campi che sono a loro volta oggetti. Questo modella la situazione in cui un oggetto è composto da parti o più in generale una relazione tra oggetti.
In alcuni linguaggi la relazione di clientship va indicata esplicitamente dal programmatore (per esempio, tramite un operatore `use`); in Java è invece implicita (le classi usate sono quelle il cui nome compare nel codice).
La relazione di clientship può essere direttamente o mutuamente ricorsiva; in particolare, un oggetto di una certa classe può avere come sottocomponenti oggetti della stessa classe. Questo avviene tipicamente nel caso di implementazione di tipi di dato induttivi, cioè in cui gli insiemi di valori (*carrier* se modelliamo un tipo di dato come algebra eterogenea, vedi Def.A.8), sono insiemi definiti induttivamente, come liste e alberi. Notiamo che in un linguaggio class-based possiamo avere ricorsione di tre diversi tipi:

[12] Si noti che l'oggetto precedentemente denotato da `r3`, se non vi sono altri riferimenti a esso, potrà essere "rimosso" dal garbage collector automatico.

- ricorsione tra classi; la classe `A` usa la classe `B` che a sua volta usa `A`;
- (se c'è ricorsione tra classi) ricorsione tra oggetti; per esempio, se le classi `A` e `B` sono mutuamente clienti come descritto sopra, un oggetto *o* di classe `A` può avere una sottocomponente di tipo `B` che ha come sottocomponente l'oggetto *o*;
- ricorsione tra metodi.

Il fatto che gli oggetti possono avere campi che sono a loro volta oggetti può causare *sharing* (condivisione) di sottocomponenti, quindi *aliasing* (cioè, più nomi possono denotare lo stesso oggetto, con la conseguenza che ogni cambiamento effettuato attraverso un nome ha effetto anche sull'altro). In questo senso il paradigma ad oggetti è simile al paradigma imperativo con puntatori. Tuttavia, vi sono alcune differenze che fanno sì che la programmazione a oggetti possa essere considerata una sorta di "nobilitazione" della programmazione con puntatori.

- In un linguaggio a oggetti è possibile scrivere direttamente definizioni di tipo ricorsive (vedi l'esempio successivo).
- In un linguaggio imperativo si dichiara un tipo `P` puntatore a un tipo `T`, mentre in un linguaggio object-oriented si dichiara unicamente un tipo oggetto `P`. Di conseguenza, nel linguaggio imperativo è possibile, data un'espressione di tipo `P`, accedere al corrispondente valore di tipo `T` tramite un operatore di "dereferencing" (indicato con `*` in C). Per esempio, date due variabili `p,q` di tipo puntatore, è possibile effettuare un'assegnazione `*p = *q`. Invece in un linguaggio a oggetti lo stato dell'oggetto è accessibile solo indirettamente tramite i metodi (o eventualmente i campi visibili).
- In un linguaggio a oggetti le variabili di tipi oggetto sono in genere inizializzate automaticamente al valore "oggetto nullo" (`null` in Java). Inoltre, la deallocazione dinamica è effettuata automaticamente da un garbage collector. Queste due caratteristiche evitano alcuni errori di programmazione. È invece ancora affidata al programmatore, come già detto, la definizione di opportuni metodi di copia.

Vediamo ora un esempio di implementazione in Java di un tipo di dato induttivo, il tipo di dato delle liste di interi. L'insieme delle liste di interi può essere definito induttivamente nel modo seguente:

- la lista vuota è una lista di interi
- se z è un intero e l è una lista di interi, allora (z, l) è una lista di interi.

In un linguaggio object-oriented è possibile esprimere la ricorsione direttamente, come illustrato sotto.

```
class RecList {
  private int head;
  private RecList tail;
  boolean isEmpty () { return tail == null;}
  int head () {
    if (tail==null) ... errore
    else return head;
  }
  RecList tail () {
    if (tail==null) ... errore
    else return tail;
  }
  RecList cons(int e) {
    RecList rl = new RecList();
    rl.head = e;
    rl.tail = this;
    return rl;
  }
}
```

Abbiamo omesso la gestione delle situazioni di errore, da effettuare in Java tramite eccezioni, vedi Sez.2.4.9. Si noti che i metodi `head()` e `tail()` hanno lo stesso nome dei campi; questo non crea ambiguità poiché l'accesso a campo ha una sintassi diversa dall'invocazione di metodo, anzi è una convenzione sensata nel caso, come qui, che i metodi restituiscano proprio il valore dei campi corrispondenti.

Si noti che sono rappresentazioni della lista vuota tutti gli oggetti con `tail == null`. Invece sono rappresentazioni di una lista con un unico elemento tutti gli oggetti in cui `tail` è un oggetto che rappresenta la lista vuota. Si noti anche che il metodo `cons` non effettua una copia (si dia per esercizio una versione che effettua una deep copy).
L'implementazione delle liste data sopra *non* è quella più "naturale" per questo tipo di dato in un linguaggio object-oriented; un'implementazione migliore che sfrutta il meccanismo di inheritance sarà presentata in Sez.2.3.3.
Un ultimo aspetto importante della relazione di clientship è quello legato alla visibilità delle componenti di una classe. In tutti i linguaggi object-oriented vi sono almeno due livelli di visibilità di una componente, *pubblica* o *privata* a seconda che la componente sia visibile o meno alle classi clienti (faccia cioè parte dell'*interfaccia utente* o no). In alcuni linguaggi le regole che determinano la classificazione delle componenti in pubbliche o private sono prefissate (per esempio, in Smalltalk gli attributi sono privati e i metodi pubblici); in altri invece è il programmatore che si fa carico della distinzione (per esempio in Java, dove sia campi che metodi possono essere o no dichiarati `private`). Per esempio data la seguente definizione di classe Java

```
class Rectangle {
  private int length,width;
  void setLength (int l) { length = l;}
  void setWidth (int w) { width = w;}
  int area () { return width * length;}
  void print () {
    System.out.println("I am a rectangle:  = "+ length + ", " + width);
  }
}
```

i campi sono visibili solo dentro la classe `Rectangle`, mentre i metodi sono visibili anche all'esterno della classe.
Mantenere i campi privati è in linea di massima una buona norma di programmazione.
Si noti che vi sono due possibili modi di intendere il fatto che una componente sia privata, detti *per classe* e *per oggetto*. Si consideri il seguente esempio:

```
class Rectangle{ ...
  boolean equals (Rectangle r) {
    return length == r.length && width == r.width;
  }
}
```

L'invocazione `r.length` è corretta in Java e C++, ma non in Smalltalk. Visibilità per oggetto significa cioè che un oggetto ha accesso a tutte le proprie componenti, ma non può accedere a componenti private di oggetti della sua stessa classe. Questa nozione è quella che meglio corrisponde alla nozione di oggetto come entità manipolabile solo attraverso un'interfaccia operazionale.
Visibilità per classe significa invece che non vi è modo di nascondere alcuna componente tra oggetti della stessa classe.
La classificazione delle componenti in pubbliche e private è solo la più grossolana possibile: in molti linguaggi è adottata una classificazione più fine, che distingue per esempio un ulteriore livello di visibilità basato sulla relazione di inheritance, come vedremo nella sezione 2.3.5, o su altre considerazioni (come i quattro livelli di visibilità di Java basati anche sull'organizzazione delle classi in *package*, vedi Sez.2.4.10).

2.2.4 Uso di `this`

In tutti i linguaggi object-oriented esiste una particolare parola chiave (`this` in Java e C++, `self` in Smalltalk, `current` in Eiffel, etc.) utilizzata come un identificatore di costante che denota, durante ogni esecuzione di un metodo, l'oggetto su cui è stato invocato il metodo. In Java è possibile riferirsi a tale oggetto anche implicitamente: per esempio questa definizione di metodo

```
boolean equals (Rectangle r) {
  return length == r.length && width == r.width;
}
```

è equivalente alla seguente

```
boolean equals (Rectangle r) {
  return this.length == r.length && this.width == r.width;
}
```

In alcune situazioni tuttavia è necessario utilizzare esplicitamente `this`. Un primo caso è quello in cui ci sono delle variabili che mascherano i campi.

```
class Rectangle {
  int length;
  ...
  void setLength (int length) {
    this.length = length;
  }
}
```

Un utilizzo più sostanziale avviene quando si vuole passare l'oggetto corrente come parametro o restituirlo come risultato di un metodo. Per esempio supponiamo di voler aggiungere nella classe `Rectangle` un metodo che confronta il rettangolo corrente con un altro e restituisce quello dei due che ha area maggiore.

```
Rectangle greaterArea(Rectangle r){
  if (area() > r.area()) return this;
  else return r;
}
```

2.2.5 Campi e metodi di classe

In molti linguaggi object-oriented tra cui Smalltalk e Java è possibile definire campi e metodi cosiddetti *di classe*. In Java questi vengono anche chiamati campi e metodi *statici* per mantenere la terminologia del C (e anche perché per essi il binding è statico anziché *dinamico*, vedi Sez.2.3.2), tuttavia il significato è diverso. Introduciamo il concetto con un esempio: aggiungiamo alla classe `Rectangle` un campo che ha come valore il numero di istanze della classe esistenti in quel momento. È chiaro che questa non è una proprietà di un singolo oggetto della classe, ma della classe stessa.

```
class Rectangle{
  static int rectNum = 0;
  static void printNofRects(){
    System.out.println ("Alive rectangles: "+ rectNum);
  }
  Rectangle (int l, int w) {
    length = l;
    width = w;
    rectNum++;
}
  protected void finalize(){ rectNum--;}
  ...
}
```

Il campo di classe `rectNum` viene incrementato ogni volta che viene creato un nuovo rettangolo (invocando il costruttore `Rectangle (int, int)`, vedi Sez.2.2.7) e viene decrementato ogni volta che un rettangolo viene cancellato. Quest'ultimo effetto è ottenuto per mezzo del metodo predefinito `finalize`, che è una particolarità di Java: viene invocato automaticamente dal sistema immediatamente prima di distruggere un oggetto.
A differenza di quanto accade per le variabili di istanza, esiste un'unica copia delle variabili di classe, condivisa da tutte le istanze esistenti. Nell'esempio, esiste una sola copia di `rectNum`. Analogamente, un metodo di classe non viene invocato su un singolo oggetto, ma sulla classe stessa. Ciò è espresso dalla sintassi della chiamata.

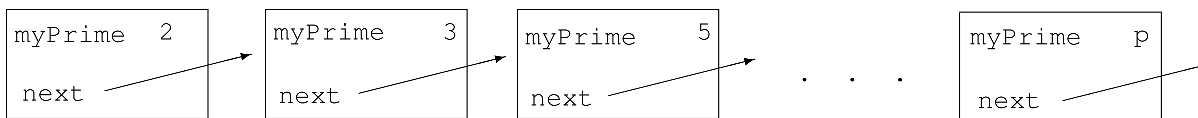
```
Rectangle.printNofRects(); /* preferibile */
r.printNofRects(); /* corretta, ma sconveniente */
```

Nel body di un metodo di classe, non è ovviamente possibile accedere a variabili di istanza, ma solo a variabili di classe, e non è possibile utilizzare implicitamente o esplicitamente `this`. Nel body di un metodo di istanza è possibile accedere sia alle variabili di istanza che a quelle di classe.
Le variabili di classe vengono allocate e inizializzate nel momento in cui il sistema carica una classe. Nel caso si richieda un'inizializzazione complessa, è possibile in Java utilizzare i cosiddetti *blocchi di inizializzazione statici*, vedi Sez.2.4.2 (in Java

1.1 sono stati aggiunti anche i blocchi di inizializzazione d'istanza, che non trattiamo in quanto risultano necessari solo all'interno di classi anonime, altro argomento non trattato).
I metodi di classe possono anche essere usati per simulare tipi di dato usuali, definendo classi prive di stato che non verranno mai istanziate, ma solo utilizzate come collezioni di operazioni; ad esempio la classe `Math` nel package `java.lang` definisce le usuali operazioni matematiche come metodi di classe.
Da quanto visto finora si comprende come, dal punto di vista del modello dell'esecuzione, le classi possano essere viste come particolari oggetti. In alcuni linguaggi questo punto di vista è portato alle estreme conseguenze; per esempio in Smalltalk le classi sono effettivamente oggetti come gli altri: di conseguenza (poichè ogni oggetto deve essere istanza di una classe) hanno una loro classe detta *metaclasse* (per esempio, `Rectangle class` è una metaclasse che ha un'unica istanza, la classe `Rectangle`), e i metodi di classe sono semplicemente i metodi della metaclasse corrispondente. In Java non c'è la nozione di metaclasse; tuttavia, esiste una classe `Class` le cui istanze sono rappresentazioni a run-time delle classi, vedi Sez.2.4.12.

2.2.6 Un algoritmo "a oggetti"

Mostriamo ora un esempio di come un algoritmo classico può essere espresso in maniera inusuale utilizzando il paradigma computazionale a oggetti. Il problema è quello di trovare tutti i numeri primi fino a un numero dato, e l'algoritmo è quello del crivello di Eratostene, che consiste nel cancellare via via dalla tabella di tutti i numeri quelli divisibili per due, poi quelli divisibili per tre, poi quelli divisibili per il successivo numero non cancellato, cioè cinque, e così via. Quando non ci sono più numeri per cui dividere, i numeri rimasti sono primi. Nella versione a oggetti, l'algoritmo è implementato utilizzando degli oggetti "crivello", ognuno dei quali rappresenta un numero primo. A ogni crivello arrivano dei numeri, e il compito del crivello che rappresenta il numero primo p è quello di filtrare questi numeri in arrivo lasciando uscire (quindi passando al crivello successivo) solo quelli non divisibili per p. La situazione è schematizzata nella figura.



Gli oggetti crivello vengono generati dinamicamente: infatti se un numero passa indenne attraverso tutti i crivelli esistenti, significa che si tratta di un nuovo numero primo trovato, quindi viene aggiunto in fondo un nuovo crivello corrispondente a questo numero. I numeri primi via via trovati vengono inseriti in uno stack (la cui implementazione assumiamo data altrove). Infine, un oggetto `PNG` (per Prime Numbers Generator) innesca l'algoritmo, creando il primo crivello[13] e generando via via i numeri da due al massimo assegnato in input.

```
class Sieve {
  int myPrime;
  Sieve next;
  static Stack myStack = new Stack();

  void filter (int i){
    if (myPrime == 0) {
      myPrime = i;
      next = new Sieve();
      myStack.push(i);
    }
    else if (i%myPrime!=0) next.filter(i);
  }
}

class PNG{
public static void main (String argv[]){
    Sieve first = new Sieve();
    final int n = Integer.parseInt(argv[0]) ;
    for (int i=2; i <= n ; i++)
      first.filter(i);
    Sieve.myStack.print();
```

[13] Utilizzando il *costruttore di default* che inizializza il campo `myPrime` a zero e `next` a `null`, come vedremo più in dettaglio in Sez.2.4.4.

```
  }
}
```

A scopo illustrativo, utilizziamo una variabile di classe `myStack` per raccogliere i numeri primi via via generati (si noti infatti che l'uso di una variabile di classe permette di modellare in modo naturale il fatto che lo stack che raccoglie i numeri primi è lo stesso per tutti i crivelli). Si noti anche che il modo in cui è implementato l'algoritmo rende possibile per i crivelli lavorare "in parallelo" (ciò è in effetti realizzabile utilizzando i *thread* in Java, che non sono trattati nel corso).

2.2.7 Inizializzazione e distruzione di oggetti

In tutti i linguaggi object-oriented deve essere fornito qualche meccanismo per inizializzare nel modo voluto gli oggetti al momento della creazione. Esistono vari tipi di soluzione. La soluzione adottata da Java (ereditata dal C++) prevede la possibilità di definire in una classe dei *costruttori* che vengono invocati al momento della creazione di un oggetto e il cui body consiste nell'inizializzazione dei campi. Vediamo un esempio.

```
class Rectangle {
  ...
  Rectangle (int l, int w) {
    length = l;width = w;
  }
}

Rectangle r = new Rectangle(2,3);
```

Come illustrato dall'esempio, un costruttore non ha un tipo di ritorno e il suo nome coincide con quello della classe; può avere dei parametri. Si possono definire diversi costruttori per una classe, ognuno identificato dalla lista dei tipi dei parametri; quindi i costruttori offrono un esempio di *overloading*. In esempi precedenti, abbiamo già visto invocazioni del costruttore di default associato implicitamente a una classe, quello senza parametri. Tuttavia, se si definisce un costruttore per una classe quello di default non è più implicitamente disponibile e, se si desidera averlo, occorre definirlo esplicitamente.
Per maggiori dettagli sul meccanismo dei costruttori in Java si veda Sez.2.4.4.
Per quanto riguarda la distruzione degli oggetti, come già detto nei linguaggi object-oriented puri questa è un'azione gestita dal sistema, mentre nei linguaggi "ibridi", come C++, viene gestita dal programmatore. Naturalmente la garbage collection automatica ha un costo in termini di tempo, anche se offre una maggiore affidabilità.

2.3 Inheritance e Sottotipo

2.3.1 Classi eredi

Una definizione di classe come la seguente

```
class Cuboid  extends Rectangle {
  int height;
  void setHeight (int h) { height = h;}
  int volume () { return area() * height;}
}
```

definisce la classe `Cuboid` (parallelepipedo) come classe *heir* (erede, sottoclasse) della classe `Rectangle`, che viene detta classe *parent* (genitore, superclasse). Si dice anche che `Cuboid` *eredita* da `Rectangle`. La definizione della classe `Cuboid` è ottenuta estendendo la definizione della classe `Rectangle` con un nuovo campo `height` e due nuovi metodi `setHeight` e `volume`, come avverrebbe ricopiando le definizioni delle componenti di `Rectangle` nella nuova classe. Intuitivamente, si intende definire un parallelepipedo come un caso particolare di rettangolo (o meglio, un oggetto che possiede tutte le caratteristiche di un rettangolo più altre aggiuntive).
La classe `Cuboid` riutilizza il codice di `Rectangle` e lo estende. Dal punto di vista implementativo, i metodi di `Rectangle` *non* sono duplicati in `Cuboid`, ma vengono ricercati nella classe parent al momento dell'invocazione. Si noti che questo vale anche in mancanza del codice sorgente di `Rectangle`; con riferimento a Java, è sufficiente che sia disponibile il file `Rectangle.class` ottenuto compilando la classe `Rectangle`.
Si noti che un metodo nuovo (per esempio `volume`) può utilizzare nel suo body componenti sia vecchie (`area()`) che nuove (`height`).
Le due classi `Rectangle` e `Cuboid` possono essere utilizzate nel modo seguente.

```
class RectCuboidTest {
  public static void main(String argv[]) {
    Rectangle r = new Rectangle();
    ...
    Cuboid c = new Cuboid();
    c.setLength(3); c.setWidth(3); c.setHeight(3);
    r = c;
    ...
  }
};
```

L'assegnazione `r = c` illustra un'importante caratteristica che nei linguaggi object-oriented si accompagna all'inheritance: un'espressione di tipo heir (*sottotipo*) può stare in un contesto di tipo parent (*supertipo*). Per esempio, un'espressione di tipo `Cuboid` può essere assegnata a una variabile di tipo `Rectangle`. Intuitivamente, un'assegnazione di questo tipo è infatti corretta poichè, dato che `Cuboid` come erede di `Rectangle` ne possiede tutti i metodi, un parallelepipedo sa rispondere ad almeno tutti i messaggi cui sa rispondere un rettangolo; un'assegnazione in senso inverso `c = r` invece potrebbe causare un errore a run-time, per esempio nel caso di una successiva invocazione `c.volume()`, e quindi è sensato che non sia permessa dal type system.
Si noti ancora che dopo l'assegnazione `r = c` in un certo senso si è "persa dell'informazione", più precisamente l'informazione di tipo che `r` denota un oggetto di classe `Cuboid`. Una successiva invocazione di metodo `r.volume ()` è infatti non corretta staticamente, anche se in realtà sarebbe sensata. Quindi quando viene utilizzato in un contesto di un tipo parent (supertipo) `P` un oggetto di un sottotipo `H`, quest'ultimo può essere manipolato solo attraverso l'interfaccia (meno ricca) offerta da `P`.
Il fatto che un'espressione di un certo tipo possa essere utilizzata anche in un contesto di un supertipo è espresso usualmente da una regola detta di *subsumption*, che può essere formulata informalmente nel modo seguente:

> se un'espressione e ha un certo tipo T e T è sottotipo di un certo tipo T', allora l'espressione e ha anche tipo T'.

In realtà, in Java la regola di subsumption nella forma generale data sopra non vale perché vi sono alcune situazioni in cui non è irrilevante considerare un'espressione di un tipo o di un supertipo (per esempio nella selezione dei campi, vedi Sez.2.4.5, e nella risoluzione dell'overloading, vedi Sez.2.4.6). Valgono invece regole che permettono di utilizzare un'espressione di un certo tipo in un contesto di un supertipo in alcune situazioni: le più importanti sono *l'assegnazione* (l'espressione deve avere un sottotipo del tipo della variabile) e l'invocazione di metodo o costruttore (l'argomento deve avere un sottotipo del tipo del parametro).
Si noti che la nozione di sottotipo e la regola di subsumption non sono un'esclusiva dei linguaggi object-oriented; per esempio , in Pascal un tipo intervallo ad esempio `type cif = 0 .. 9` è un sottotipo del tipo scalare corrispondente, in questo caso `integer`, e quindi un'espressione di tipo `cif` può apparire ovunque ci si aspetti un `integer`.

2.3.2 Overriding

Nell'esempio precedente la classe `Cuboid` era definita per semplice *estensione* della classe `Rectangle`, cioè ereditava i metodi di `Rectangle` senza modificarne la definizione. In generale però una classe heir può *ridefinire* i metodi della classe parent, come illustrato dal seguente esempio.

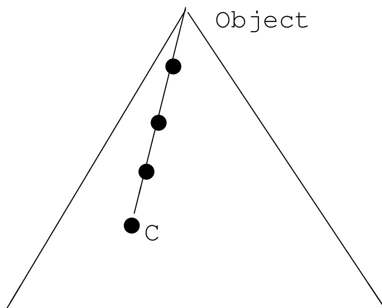
```
class Rectangle {
  ...
  void init () { length = 1; width = 1;}
  void print () {
    System.out.println("I am a rectangle:  = "+ length + ", " + width);
  }
}

class Cuboid  extends Rectangle {
  ...
  void init () { length = 1; width = 1; height = 1;}
  void print () {
    System.out.println("I am a cuboid:  " + length + ", " + width + ", " + height);
  }
}
```

Si dice che i metodi `init` e `print` sono *overriden* (ridefiniti). In questo caso ovviamente a livello implementativo le due classi non condivideranno il codice per questi metodi.
In caso di ridefinizione, si pone il problema, data una chiamata di metodo, di determinare la versione del metodo da invocare. Usualmente nei linguaggi object-oriented si ha il cosiddetto *late binding* (*dynamic binding*), cioè la versione del metodo da invocare è determinata dal *tipo dinamico* del ricevitore, cioè la classe di cui è istanza l'oggetto correntemente denotato (mentre il *tipo statico* è quello che può essere attribuito a un'espressione esaminando il codice). Quindi la decisione sulla versione del metodo da invocare non è presa a compile-time ma a run-time. Vediamo un esempio.

```
Rectangle r = new Rectangle(); ...
r.print();//viene invocato print di Rectangle
Cuboid c = new Cuboid(); ...
r = c;
r.print ();//viene invocato print di Cuboid
```

Nell'ultima linea `r` ha tipo statico `Rectangle` (il tipo con cui la variabile è stata dichiarata), tipo dinamico `Cuboid`. La verifica della correttezza statica del codice è fatta in base ai tipi statici; in particolare, la chiamata `r.print ()` è giudicata corretta perchè la classe `Rectangle` definisce effettivamente un metodo `print ()`. Tuttavia, al momento dell'esecuzione il metodo che viene effettivamente invocato è quello di `Cuboid`. Se la versione del metodo da invocare fosse determinata a compile-time (*binding statico*), nell'esempio verrebbe invocato il metodo della classe `Rectangle`.
Il tipo statico di un'espressione è quindi determinato a compile-time e di conseguenza non varia durante l'esecuzione, mentre il tipo dinamico può cambiare: in base alla regole viste prima però, *il tipo dinamico è sempre un sottotipo del tipo statico.*
A livello implementativo, la versione del metodo da invocare viene determinata nel modo seguente: data una chiamata `o.m(...)`, anzitutto si determina il tipo dinamico di `o`; sia una certa classe `C`. Se `C` contiene una definizione per `m`, la ricerca è terminata; altrimenti si va a cercare nella classe parent di `C`, e così via, finchè non si arriva alla classe `Object`, la classe "di tutti gli oggetti" che costituisce la radice della gerarchia (in molti linguaggi object-oriented esiste una classe analoga: se una classe non è dichiarata erede di un'altra, è per default sottoclasse di `Object`). Si noti che la gerarchia delle classi ha una struttura ad albero nel caso, che abbiamo considerato sinora, di inheritance singola, cioè quando ogni classe ha un'unica classe parent. La situazione è schematizzata nella figura sotto.



La ricerca sopra descritta può anche fallire, cioè si può arrivare fino ad `Object` e non trovare una definizione del metodo cercato. Per esempio, ciò può avvenire in un linguaggio non tipato staticamente[14], come Smalltalk, dove è possibile assegnare a una variabile qualunque oggetto. Quindi, data una chiamata di un metodo `m` dove il ricevitore è denotato da una variabile `x`, non vi è modo di sapere se al momento dell'esecuzione `x` denoterà un oggetto che sa rispondere al messaggio `m` (ossia, istanza di una classe che abbia tra i suoi metodi, propri o ereditati, `m`). Se la ricerca del metodo fallisce, si ha un errore detto di "messaggio non compreso". In un linguaggio tipato staticamente, le regole di tipo dovrebbero garantire che una tale situazione non si verifichi: si dice in tal caso che il type system è *safe*.
Java ha un type system safe; tuttavia, in alcuni casi vi sono controlli di tipo a run-time, come nel casting down, vedi Sez.2.4.8.
Si noti che, poichè le definizioni dei metodi in una classe possono essere mutuamente ricorsive, ridefinire un solo metodo ha come conseguenza che anche il comportamento di tutti gli altri metodi può potenzialmente cambiare.
La ridefinizione può essere *conservativa* o *non conservativa*. Per ridefinizione conservativa intendiamo una ridefinizione che preserva il comportamento originale, estendendolo (come quella di `init` nell'esempio precedente), mentre una ridefinizione non conservativa lo cambia (per esempio `void init () { length = 2; width = 2; height = 2;}`).

[14]Attenzione: vuol dire che le espressioni del linguaggio non hanno tipo, ma ogni oggetto è sempre istanza di un'unica classe.

2.3.3 Inheritance e programmazione "per casi"

Illustriamo ora come l'inheritance accoppiata al binding dinamico permetta, nel caso di implementazione di tipi "somma", di cambiare drasticamente l'organizzazione del codice e di ottenere una maggiore modularità ed estendibilità.
Diremo che T è un tipo *somma* o *unione disgiunta* se l'insieme degli elementi di tipo T è un'unione disgiunta di insiemi (cioè tutti gli insiemi che compongono l'unione sono a due a due disgiunti).
Supponiamo di dover implementare un tipo di dato "figura geometrica". Una figura geometrica può essere un rettangolo, un cerchio o un triangolo. Formalmente, si ha la seguente relazione tra gli insiemi di valori corrispondenti

Figure = Rectangle ∪ Circle ∪ Triangle

dove si noti che si tratta di unioni disgiunte.
Il modo standard in un linguaggio non object-oriented di implementare un tipo somma è quello di utilizzare dei tipi appositi, come i tipi record con variante del Pascal, i tipi `union` del C, o i tipi unione in Caml. Per esempio in Caml si avrà:

```
type figure = Rectangle of float * float | Circle of float | Triangle of float * float
```

dove le componenti di tipo `float` sono nel caso del rettangolo base e altezza, nel caso del cerchio il raggio e nel caso del triangolo base e altezza. Un'operazione che manipola figure, per esempio una funzione che restituisce l'area di una figura, viene quindi tipicamente implementata dando una definizione "per casi", utilizzando un costrutto apposito (`case` in Pascal, `switch` in C, definizioni per pattern-matching in Caml). Per esempio in Caml si avrà:

```
 let area = function Rectangle(l,w) -> l *. w
  | Circle(r) -> r *. r *. 3.14
  | Triangle(b,h) -> b *. h /. 2.0;;
```

In un linguaggio object-oriented, invece, l'implementazione standard di un tipo somma consiste nel definire una classe parent (astratta se possibile nel linguaggio, vedi Sez.2.3.6) corrispondente al tipo somma e tante classi heir quanti sono i casi possibili. Tutte le operazioni del tipo di dato la cui definizione è diversa nei vari casi verranno implementate con metodi non definiti nella classe parent e definiti di volta in volta nel modo appropriato nelle classi heir (naturalmente operazioni la cui definizione è invece uguale in tutti o un buon numero di casi verranno implementate con metodi definiti nella classe parent e ridefiniti solo in quelle sottoclassi corrispondenti a casi in cui il comportamento è diverso). Nel nostro esempio:

```
abstract class Figure {
  ...
  abstract double area ();
}

class Rectangle extends Figure {
  private double length;
  private double width;
  double area () {
    return length * width;
  }
}

class Circle extends Figure {
  static final double PI = 3.1416;
  // inserito per dare un esempio di campo statico: la costante pi-greco
  // e' predefinita e si trova nella classe Math (Math.PI)
  private double radius;
  double area () {
    return radius * radius * PI;
  }
}

class Triangle extends Figure {
  private double basis;
  private double height;
```

```
  double area () {
    return basis * height /2;
  }
}
```

In questo modo, è possibile per una classe cliente di quelle definite sopra utilizzare la classe astratta `Figure` come un tipo, e assegnare a variabili di questo tipo di volta in volta oggetti del sottotipo richiesto. Una chiamata del metodo `area ()` causerà l'invocazione della versione "giusta" in modo automatico, in base al binding dinamico.

```
class FigureTest {
  public static void main(String argv[]) {
    Figure[] r = new Figure[10];
    ...
    double areaTot = 0.0;
    for (int i = 0;i < r.length;i++)
      areaTot += r[i].area();
    ...
  }
}
```

Si noti che è possibile manipolare un array di figure eterogenee. Durante l'esecuzione del ciclo `for`, a ogni iterazione viene invocata la versione di metodo appropriata. Un utilizzatore esterno non deve preoccuparsi di quale versione del metodo viene invocata, anzi a priori non deve neppure sapere quali tipi di figure esistono.
Un'altra differenza tra questo approccio e quello "per casi", forse ancora più importante, è relativa al modo in cui viene gestita un'estensione del tipo originale. Supponiamo per esempio che, in un secondo momento, si voglia aggiungere una nuova categoria di figure geometriche, per esempio le ellissi. Nell'approccio tradizionale è chiaro che occorre cambiare la definizione del tipo `figure` e il codice di `area()`, come quello di tutte le operazioni definite per casi. Nell'approccio object-oriented, è sufficiente definire una nuova classe heir di `Figure`, `Ellipses`, con la sua propria definizione del metodo `area()`; non occorre modificare il codice precedente. È evidente il vantaggio enorme di questo tipo di soluzione. Questa caratteristica dell'approccio object-oriented (quella cioè di consentire di estendere il codice senza modificare quanto già scritto) è detta anche principio *open-closed*, in quanto in questo modo è possibile scrivere moduli software che sono *chiusi* nel senso di completi e utilizzabili, ma *aperti* nel senso che è sempre possibile aggiungere nuove funzionalità (e senza bisogno che il codice sorgente sia disponibile).
In [2] si propone un'interessante schematizzazione della differenza tra approccio "per casi" e approccio object-oriented che abbiamo illustrato finora. L'osservazione di partenza è che molto spesso in un tipo di dato è possibile classificare le operazioni in *costruttori*[15], cioè operazioni che servono per *costruire* i valori del tipo in esame, e *osservazioni*. Per esempio nel caso delle liste possiamo considerare come costruttori empty e cons, e come osservazioni $\mathit{isEmpty}$, head, tail. Il tipo di dato può essere allora sinteticamente specificato per mezzo di una tabella, che indica per ogni costruttore i valori corrispondenti delle osservazioni.
Per esempio per le liste la tabella avrà la forma seguente.

	costruttore di l	
osservazioni	*empty*	$\mathit{cons}(e, l')$
$\mathit{isEmpty}(l)$	*true*	*false*
$\mathit{head}(l)$	*indefinito*	e
$\mathit{tail}(l)$	*indefinito*	l'

Nel caso di osservazioni con più argomenti la tabella sarà pluridimensionale; per esempio vediamo la tabella corrispondente all'osservazione $\mathit{equals}: \mathit{list}\ \mathit{list} \to \mathit{bool}$:

	costruttore di l_1	
costruttore di l_2	*empty*	$\mathit{cons}(e_1, l'_1)$
empty	*true*	*false*
$\mathit{cons}(e_2, l'_2)$	*false*	$e_1 = e_2 \wedge \mathit{equals}(l'_1, l'_2)$

L'approccio "per casi" corrisponde allora a decomporre la tabella per righe; infatti viene definita una funzione (per casi) per ogni riga della tabella. L'approccio object-oriented corrisponde invece a una decomposizione per colonne; infatti viene definita

[15] Non si confonda con i costruttori nel senso di Java.

una classe per ogni costruttore, all'interno della quale si definiscono opportunamente le osservazioni. In entrambi gli approcci è possibile avere incapsulazione, nel senso che i dati possono essere manipolati da un cliente solo attraverso le osservazioni.
Per esempio, un'implementazione delle liste in Java che corrisponde alla prima tabella è data qui sotto.

```
 abstract class List {
   abstract boolean isEmpty();
   abstract int head();
   abstract List tail();
 }
class EmptyList extends List {
  boolean isEmpty () { return true;}
  int head () { ... errore }
  List tail () {... errore}
}

class NonEmptyList extends List {
  private int head;
  private List tail;
  NonEmptyList (int head, List tail) {
    this.head = head;
    if (tail.isEmpty()) this.tail = new EmptyList();
    else this.tail = new NonEmptyList(((NonEmptyList)tail).head,((NonEmptyList)tail).tail);
  }
  boolean isEmpty () { return false;}
  int head () { return head;}
  List tail () {return tail;}
}
```

Abbiamo omesso la gestione delle situazioni di errore, da effettuare in Java tramite eccezioni, vedi Sez.2.4.9.
Si noti nella definizione del costruttore `NonEmptyList(int,List)` l'uso del casting down, vedi Sez.2.4.8, per poter accedere ai campi `head` e `tail` di `tail`. Nella versione qui presentata il costruttore effettua una copia della lista argomento; una versione senza copia è la seguente:

```
  private NonEmptyList (int head, List tail) {
    this.head = head;
    this.tail = tail;
  }
```

Esaminiamo ora cosa succede in entrambi gli approcci in caso di aggiunta di nuove operazioni.

Aggiunta di costruttori Supponiamo per esempio di aggiungere un nuovo costruttore $interval$: *int int* $\rightarrow$ *list* che costruisce le liste corrispondenti agli intervalli $[n, m]$ di $\mathbb{Z}$ ($n \leq m$), in modo da poter ottimizzare la rappresentazione di queste particolari liste (basta infatti memorizzare i due estremi).
La tabella precedente deve quindi essere estesa nel modo seguente.

	costruttore di l		
osservazioni	*empty*	$cons(e, l')$	$interval(n, m)$
$isEmpty(l)$	*true*	*false*	*false*
$head(l)$	*indefinito*	e	n
$tail(l)$	*indefinito*	l'	*empty* se $m = n$, altrimenti $interval(n + 1, m)$

L'approccio object-oriented risulta, come già notato, molto più conveniente: infatti nell'approccio "per casi" è necessario cambiare la definizione del tipo e cambiare (aggiungendo un caso) la definizione di tutte le osservazioni. Nell'approccio object-oriented invece è sufficiente aggiungere una nuova classe corrispondente al nuovo costruttore con la propria definizione delle osservazioni. Per esempio in Java basta aggiungere la seguente definizione.

```
class IntervalList extends List {
  private int a,b;

  IntervalList(int a, int b) {
    if (b<a) ... errore
    else {
      this.a =a; this.b = b;
    }
  }
  boolean isEmpty() { return false;}
  int head() { return a;}
  List tail() {
    if (a<b) return new IntervalList(a+1,b);
    else return new EmptyList();
  }
}
```

Abbiamo omesso la gestione delle situazioni di errore, da effettuare in Java tramite eccezioni, vedi Sez.2.4.9.

Aggiunta di nuove osservazioni Supponiamo per esempio di aggiungere un'osservazione $\mathit{length}\colon \mathit{list} \rightarrow \mathit{int}$ che restituisce la lunghezza della lista.

	costruttore di l		
osservazioni	*empty*	$\mathit{cons}(e, l')$	$\mathit{interval}(n, m)$
$\mathit{isEmpty}(l)$	*true*	*false*	*false*
$\mathit{head}(l)$	*indefinito*	e	n
$\mathit{tail}(l)$	*indefinito*	l'	*empty* se $m = n$, altrimenti $\mathit{interval}(n+1, m)$
$\mathit{length}(l)$	0	$\mathit{length}(l') + 1$	$m - n + 1$

Anzitutto, a prescindere dall'approccio scelto, la nuova osservazione può essere definita come nuova primitiva, cioè effettivamente per casi (come illustrato sopra) oppure come *operazione derivata*, cioè in termini delle osservazioni definite precedentemente. Nel nostro esempio si avrebbe:

$\mathit{length}(l) = 0$ se $\mathit{isEmpty}(l)$, altrimenti $\mathit{length}(\mathit{tail}(l)) + 1$

Naturalmente definire l'osservazione come nuova primitiva offre spesso un vantaggio dal punto di vista dell'ottimizzazione; nell'esempio, nel caso delle liste intervallo. In compenso, definirla come operazione derivata ha il vantaggio che in questo modo si ha una definizione indipendente dall'implementazione, che resta quindi valida cambiando quest'ultima.
Nell'approccio "per casi" comunque l'aggiunta della nuova osservazione risulta semplice (basta aggiungere una nuova funzione definita per casi, o in maniera derivata).
Nel paradigma object-oriented ci sono invece due possibilità abbastanza diverse:

- aggiungere la definizione dell'operazione come derivata nella classe parent (si modifica quindi una sola classe, ma non si può ottimizzare);
- aggiungere l'opportuna definizione dell'operazione come primitiva in ogni classe heir, come illustrato sotto:

```
abstract class List {
  ...
  abstract int length();
}
class NonEmptyList extends List {
  ...
  int length () { return 1 + tail.length(); }
}
class EmptyList extends List {
  ...
  int length () { return 0; }
```

```
  }
  class IntervalList extends List {
    ...
    int length () { return b-a+1; }
  }
```

si modificano quindi tutte le classi, compresa la parent dove occorre aggiungere una dichiarazione di metodo astratto, ma si può ottimizzare.

Inoltre, è anche possibile adottare una soluzione intermedia (forse la strategia migliore), cioè dare la definizione derivata nella classe astratta e ridefinirla opportunamente solo in quelle classi heir dove effettivamente la ridefinizione offre dei vantaggi dal punto di vista dell'efficienza (per le liste intervallo nell'esempio).

Esercizio 1 Si aggiungano nell'implementazione delle liste vista sopra le seguenti nuove osservazioni, prima come operazioni derivate e poi come nuove primitive: uguaglianza $equals\colon$ *list list* $\rightarrow$ *bool*, concatenazione di due liste $append\colon$ *list list* $\rightarrow$ *list*, operazione che restituisce la lista rovesciata $reverse\colon$ *list* $\rightarrow$ *list*.

Soluzione Diamo la soluzione come operazioni derivate.

```
abstract class  List { ...
  boolean equals (List list) {
    if (isEmpty()) return list.isEmpty();
    if (list.isEmpty()) return false;
    return head()==list.head()&& tail().equals(list.tail());
  }

  List append (List list) {
    if (isEmpty()) return list;
    return new NonEmptyList(head(), tail().append(list));
  }

  List reverse () {
    if (isEmpty()) return new EmptyList();
    return tail().reverse().append(new NonEmptyList(head(), new EmptyList()));
  }
}
```

Esercizio 2 Si consideri la seguente implementazione in Java degli alberi binari con etichette intere (vedi Def.A.5) dove abbiamo omesso per semplicità il trattamento delle situazioni di errore tramite eccezioni.

```
abstract class BTree {
  abstract boolean isEmpty ();
  abstract int label ();
  abstract BTree left ();
  abstract BTree right ();
}

class Node extends BTree {
  private int label;
  private BTree left, right;
  Node (int label, BTree left, BTree right) {
    this.label = label; this.left = left; this.right = right;
  }
  boolean isEmpty () { return false;}
  int label () { return label; }
  BTree left () { return left;}
  BTree right () { return right;}
}
class Empty extends BTree {
  boolean isEmpty () { return true;}
  int label () { //error }
  BTree left () { //error}
  BTree right () { //error}
}
```

Attenzione: non si confonda l'albero vuoto (rappresentato da ogni istanza della classe `Empty`) con il valore `null`.

1. si aggiunga in `BTree` un metodo non astratto che restituisce l'altezza dell'albero (cioè, la lunghezza di un cammino massimale, 0 se l'albero è vuoto);
2. si aggiunga in `BTree` un metodo astratto che restituisce il numero di nodi dell'albero, e si implementi il metodo nelle sottoclassi;
3. si aggiunga una classe erede `Leaf` le cui istanze rappresentano *foglie* (alberi con la sola radice);
4. si definisca un metodo (attenzione: dove e di che tipo?) che dati due interi n e m restituisce un albero binario (preferibilmente di altezza minimale) che ha come etichette i numeri da n a m (quindi vuoto se $n > m$).

2.3.4 Esempi di classi wrapper

Come ulteriore illustrazione della relazione tra l'approccio "per casi" e quello object-oriented discussa sinora, mostriamo due esempi di classi le cui istanze "simulano" il comportamento di valori di tipi base. Tali classi vengono dette "wrapper".[16]
Come primo esempio definiamo un'interfaccia (vedi Sez.2.3.6) `Bool` i cui oggetti rispondono ai messaggi `not`, `and` e `or`.

```
interface Bool {
  Bool not ();
  Bool and (Bool b);
  Bool or (Bool b);
}

class True implements Bool {
  Bool not () {return new False();}
  Bool and (Bool b) {return b;}
  Bool or (Bool b) {return this;}
}

class False implements Bool {
  Bool not () { return new True(); }
  Bool and (Bool b) { return this;}
  Bool or (Bool b) { return b;}
}
```

Si noti che, mentre i "veri" valori booleani sono solo due, è possibile creare un numero arbitrario di istanze di `True` e `False`.
Analogamente, possiamo definire una classe `Nat` le cui istanze si comportano come numeri naturali. I numeri naturali possono essere definiti induttivamente come segue: lo zero è un numero naturale e il successore di un numero naturale è un numero naturale. Con la solita metodologia di implementazione dei tipi somma, si ottiene la seguente definizione.

```
abstract class Nat {
  abstract int val ();
  abstract Nat add (Nat n);
  abstract Nat mult (Nat n);
  abstract boolean isZero ();
  abstract boolean equals (Nat n);
  ...
}

class Zero extends Nat {
  int val () { return 0;}
  Nat add (Nat n) { return n;}
  Nat mult (Nat n) { return this;}
  boolean isZero () { return true; }
  boolean equals (Nat n) { return n.isZero();}
}

class Succ extends Nat {
```

[16] In Java esistono classi wrapper dei tipi primitivi predefinite, vedi Sez.2.4.12.

```
  private Nat pred;
  Succ (Nat n) { pred = n;}
  int val () { return pred.val() +1;}
  Nat add (Nat n) { return new Succ(pred.add(n));}
  Nat mult (Nat n) { return pred.mult(n).add(n);}
  boolean isZero () { return false; }
  boolean equals (Nat n) {
    if (n.isZero()) { return false;}
    else { return pred.equals(((Succ) n).pred);}
  }
}
```

Si noti nella definizione di `equals` l'uso del casting down, vedi Sez.2.4.8, per poter accedere al campo `pred` di `n`. Si noti che nella versione precedente le operazioni di somma e prodotto sono definite come primitive, cioè per casi nelle sottoclassi (si veda la discussione in Sez.2.3.3). Si dia per esercizio una versione in cui queste operazioni sono invece definite come derivate, utilizzando un metodo `Nat pred()` definito in `Nat`.

2.3.5 Inheritance e visibilità

In Sez.2.2.3 abbiamo visto che in tutti i linguaggi object-oriented le componenti di una classe possono essere pubbliche (cioè visibili ai clienti) o private (cioè visibili solo all'interno della classe stessa). In alcuni linguaggi esiste una ulteriore classificazione delle componenti non pubbliche (cioè non visibili ai clienti) in componenti visibili agli eredi e componenti private in senso stretto, cioè non visibili nemmeno alle classi eredi. Si dice in questo caso che una classe offre due diverse interfacce: l'*interfaccia client* e l'*interfaccia heir*. Usualmente, l'interfaccia client è inclusa (strettamente) nell'interfaccia heir. In Java, per esempio , la visibilità delle componenti di una classe è regolata dal programmatore attraverso i *modificatori di accesso* (*access control modifier*) `private`, `protected`, `public`. Le componenti che fanno parte dell'interfaccia heir sono dichiarate con modificatore `protected`; quelle che fanno parte dell'interfaccia client con modificatore `public`, mentre quelle visibili solo all'interno della classe stessa con modificatore `private`. In Java tuttavia le regole di visibilità sono ulteriormente complicate dall'esistenza di un ulteriore livello dovuto alla strutturazione del codice in *package*, peculiare di questo linguaggio e ortogonale alla relazione di inheritance tra classi; si rimanda quindi a Sez.2.4.11 per un'analisi più dettagliata.

2.3.6 Interfacce e classi astratte

Abbiamo visto precedentemente che una classe viene utilizzata sia come tipo che come schema per istanziare oggetti. In realtà, queste due nozioni possono essere completamente separate; il *tipo* di un oggetto corrisponde all'interfaccia operazionale, mentre la classe fornisce una particolare implementazione di tale interfaccia. È possibile quindi che più classi implementino lo stesso tipo.
Java accoglie parzialmente questo punto di vista: le classi sono tipi, ma è anche possibile definire dei "tipi puri", detti *interfacce*. Le interfacce contengono solo dichiarazioni di metodi (nome e funzionalità, ma senza body) e di costanti.
Per esempio, è possibile definire un'interfaccia corrispondente al tipo degli stack di interi con le usuali operazioni.

```
interface Stack {
  boolean isEmpty ();
  int pop();
  void push (int e);
  void print ();
}
```

Data questa interfaccia, è possibile definire diverse classi che la implementano; viceversa, una classe può implementare più interfacce.

```
class StackByArray implements Stack { ... };

class StackByList implements Stack{ ... };
```

Naturalmente una classe non astratta che implementa un'interfaccia deve fornire almeno tutti i metodi dichiarati nell'interfaccia. Non è possibile creare istanze di interfacce, ma è possibile dichiarare variabili di tipi interfaccia; è possibile assegnare a queste variabili istanze di classi che implementano l'interfaccia. Dal punto di vista metodologico, l'uso di interfacce permette di mescolare, per esempio, le due diverse implementazioni degli stack, che possono essere manipolate nello stesso modo senza preoccuparsi del fatto che l'implementazione è diversa.

```
class StackTest{
  static void main (String argv[]) {
    Stack s = new StackByList ();
    for (int i=1;i<20;i++) s.push(i);
    for (int i=1;i<5;i++) s.pop();
    s = new StackByArray ();
    for (int i=1;i<20;i++) s.push(i);
    for (int i=1;i<5;i++) s.pop();
}
}
```

La nozione di interfaccia permette di avere una forma limitata di *inheritance multipla*, vedi Sez.2.3.7; infatti una classe può avere un'unica classe parent, ma può implementare più interfacce (non sorgono infatti in questo caso problemi di conflitto).

```
interface Figure {
  void Move (double dx, double dy);
  void Draw ();
  void UnDraw();}

interface ColoredObject {
  void ChangeColor (color c);
}

class ColoredFigure implements Figure, ColoredObject { ... }
```

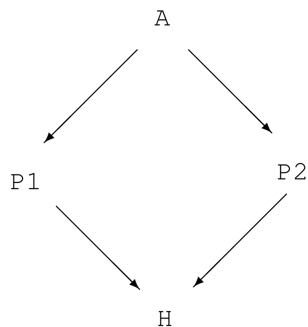
Classi e interfacce sono entrambe tipi; se una classe `C` implementa un'interfaccia `I`, `C` è sottotipo di `I`.
Una nozione intermedia tra quella di classe e quella di interfaccia è quella di *classe astratta* (presente, sotto varie forme, in quasi tutti i linguaggi object-oriented). Una classe astratta è una classe in cui alcuni metodi sono solo dichiarati e non definiti, come in un'interfaccia; tali metodi sono detti *abstract* (in Java; altri termini usati sono *deferred* e *pure virtual*). Quindi un'interfaccia può essere vista come un caso particolare di classe astratta che definisce solo metodi abstract.
Dal punto di vista metodologico, le classi astratte sono tipicamente utili quando si vogliono definire delle operazioni derivate da un nucleo di primitive di cui non si desidera fissare l'implementazione; il nucleo di primitive è dichiarato `abstract` (come nell'esempio dell'implementazione delle liste di interi dato in Sez.2.3.3).
In Java non è possibile creare istanze di classi astratte, esattamente come avviene per le interfacce. Altri linguaggi (per esempio Smalltalk) consentono di creare istanze di classi astratte, con l'idea che sia a carico del programmatore il garantire che queste siano manipolate solo attraverso chiamate ai metodi già definiti; un'invocazione di un metodo astratto provoca in questo caso un errore a run-time.

2.3.7 Varianti della nozione di inheritance

La forma di inheritance che abbiamo illustrato finora è detta *singola*, nel senso che ogni classe è erede di un'unica classe parent. Nel caso sia possibile per una classe avere più di una classe parent si parla di *inheritance multipla*. In questo caso la gerarchia delle classi non risulta più essere un albero, ma un grafo. L'inheritance multipla è tipicamente utile nei casi in cui si debbano implementare oggetti che riuniscono proprietà di due o più classi esistenti. Tuttavia la maggior parte dei linguaggi evita l'inheritance multipla, perchè sorge il problema di come gestire il conflitto quando un metodo è definito in due o più classi parent. Supponiamo per esempio di avere una classe `ColoredFigure` definita come erede sia di `Figure` che di `ColoredObject`. Il metodo `clear` potrebbe essere definito in entrambe le classi, ovviamente con diverso significato. Sorge quindi il problema di decidere quale versione del metodo deve essere ereditata. Ogni linguaggio che supporta inheritance multipla deve fornire una regola. Le soluzioni sono in genere di tre tipi:

- vietare situazioni di conflitto (ossia, in caso di conflitto si ha errore statico; in genere in questo caso il linguaggio fornisce anche la possibilità di *ridenominare* metodi per risolvere la situazione);
- fissare una regola di precedenza;
- permettere al programmatore di scegliere quale versione del metodo vuole ereditare.

Un problema ulteriore è il cosiddetto *problema del diamante*, illustrato nella figura sotto.



Nel caso due parent di una classe abbiano un antenato comune, e in entrambe le classi la versione di un certo metodo sia la stessa, ereditata dall'antenato comune, ovviamente non vi dovrà essere conflitto. Occorre quindi a livello implementativo un meccanismo non banale che rilevi questo tipo di situazioni e permetta alla classe heir di ereditare il metodo comune senza problemi.
Citiamo infine due altre varianti interessanti della nozione di inheritance.

Delegation La nozione di inheritance può essere introdotta in maniera indipendente da quella di classe, quindi anche in linguaggi object-based e non class-based. In questo caso, i singoli oggetti sono dei prototipi che possono essere duplicati con eventuali modifiche; quando si invoca un metodo di un oggetto, questo può *delegare* un altro oggetto a rispondere al messaggio.

Migrazione Un aspetto non considerato nei linguaggi, ma molto importante nelle basi di dati object-oriented, è la possibilità di avere oggetti che migrino dinamicamente da una classe all'altra. Un esempio classico è quello di uno studente che a un certo punto diventa laureato.

2.4 Il linguaggio Java

2.4.1 Generalità

Java è un linguaggio sviluppato dalla Sun Microsystems già nel 1991 (sotto il nome di Oak) con l'obiettivo di avere un linguaggio piccolo e facilmente portabile. È diventato di interesse generale quando è stato accoppiato con HotJava (1994), un World Wide Web browser in grado di caricare da un server *applet* Java cioè mini-applicazioni che possono essere eseguite sulla macchina client. HotJava ha fornito anche un esempio di applicazione complessa scritta in Java. Java, tuttavia, è anche un linguaggio general-purpose, le cui caratteristiche principali sono le seguenti:

- è un linguaggio object oriented "puro", con una ricca collezione di classi predefinite (Application Programming Interface);
- è progettato per aiutare a programmare correttamente (fornisce per esempio garbage collection automatica, controlli di tipo, eccezioni);
- è semplice[17] (cioè basato su pochi concetti ma chiari: questo è stato uno dei goal iniziali di design);
- è portabile, cioè indipendente dalla piattaforma;
- è "familiare", cioè simile a C, C++ e altri linguaggi object-oriented (Eiffel, Smalltalk).

La portabilità è garantita, tra le altre cose, dal fatto che l'implementazione è effettuata con una tecnica mista, in due passi: il codice Java è tradotto dal compilatore in *bytecode*, un codice intermedio che viene eseguito dall'interprete (macchina virtuale) Java. Quindi per eseguire le applicazioni Java è sufficiente che il sistema fornisca un'implementazione della macchina virtuale Java. Questo interprete è "built-in" nei browser abilitati Java. Naturalmente questo approccio sacrifica un poco la velocità di esecuzione. In Java si possono scrivere applet o applicazioni stand-alone. Un'applicazione consiste di una o più classi di cui almeno una contiene un metodo `static void main(String [] args)`. Nella fase di compilazione viene generato un file con estensione `class` per ogni classe definita nel codice sorgente. Una classe che definisce un metodo `main` è eseguibile, cioè è possibile invocare l'interprete sul corrispondente file `.class`. Un esempio di applicazione è il seguente:

[17] O meglio lo era inizialmente; questa caratteristica è decisamente andata persa nelle ultime versioni!

```
class HelloWorld {
    public static void main (String [] args) {
       System.out.println("Hello World!");
    }
}
```

Il file sorgente deve chiamarsi `HelloWorld.java`.
La sintassi di Java segue lo stile di C e C++.
I commenti sono gli stessi del C, più commenti speciali del tipo

```
/** commento **/
```

Questi commenti sono utilizzati da `javadoc` per produrre una versione documentata delle classi.
Come in C, il punto e virgola è un terminatore.
In Java esistono tre tipi di variabili: variabili di istanza, di classe e locali. Non esistono variabili globali. Le dichiarazioni di variabili locali possono apparire ovunque in una definizione di metodo. Le variabili locali vanno inizializzate, mentre quelle di istanza e di classe hanno valori di default dipendenti dal tipo (`null` per i tipi oggetto, `0` per i tipi numerici, `\0` per il tipo `char`, `false` per il tipo `boolean`). Le variabili possono essere dichiarate `final`, cioè costanti.
Java usa l'insieme di caratteri Unicode (a 16 bit), che include non solo i caratteri ASCII ma migliaia di altri caratteri per rappresentare la maggior parte degli alfabeti.
Un tipo è:

- uno degli otto tipi primitivi;
- un nome di classe o interfaccia;
- un tipo array;
- un tipo enumerativo, introdotto in J2SE 5 (e che non trattiamo in questo corso).

Gli otto tipi predefiniti sono built-in nel sistema quindi sono gestiti in modo più efficiente. Un aspetto importante, sempre ai fini della portabilità, è che i tipi predefiniti sono completamente specificati nel linguaggio, quindi completamente indipendenti dalla macchina. Ci sono quattro tipi interi (`byte`, `short`, `int`, `long`, rispettivamente a 8, 16, 32 e 64 bit), due tipi floating-point (`float` e `double`, rispettivamente a 32 e 64 bit), il tipo `char` (i caratteri Unicode, 16 bit) e il tipo `boolean` (i cui unici valori sono `true` e `false`). Non esiste un'istruzione simile a `typedef` del C.
Gli operatori utilizzati in Java sono quelli di C e C++. Ricordiamo i seguenti:

- operatori aritmetici `+`, `-`, `*`, `/`,`%`,
- operatori relazionali >, >=, <, <=, ==, `!=`,
- `&&` e `||` (il secondo operando non viene valutato se non è necessario),
- operatore condizionale `(be? e1: e2)`
- operatori di assegnazione =, `op`= con `op` operatore binario, per esempio `+`=, `*`=.

Anche gli statement sono quelli di C e C++; l'unica differenza è che non esiste `goto` ma solo la possibilità di utilizzare `break` e `continue` relativi a un loop più esterno etichettato:

- alcune espressioni (assegnazioni, espressioni con incremento e decremento `++` e `--`, chiamate di metodo e di costruttori) con `;` alla fine,
- dichiarazione di variabili con inizializzazione, per esempio `int x = 0;`,
- blocco { ...},
- `if (be) s1 else s2`, `if (be) s` (`else` è legato all'`if` più vicino),
- ```
  switch(ie) {
    case l1 : s1
    ...
    case lk : sk
    default : s
  }
  ```
  ```

- `break` esce dal blocco più interno,
- `while (be) s`
  `do s while (be)`
  `for (init-e; be; incr-e) s`
  (l'inizializzazione del `for` può anche consistere di più espressioni),
- `break label` esce dal blocco etichettato `label`,
- `continue` e `continue label` vanno alla fine di un loop,
- `return` e `return e` terminano l'esecuzione di un metodo, un costruttore, un inizializzatore statico.

Vediamo un esempio di loop etichettato.

```
        boolean found = false;
search: for (int y = 0; y < Matrix.length; y++) {
          for (int x = 0; x < Matrix[y].length; x++) {
            ...
            found = true;
            break search;
          }
        }
```

Secondo i progettisti di Java l'uso di `break` etichettati permette di trattare la grande maggioranza dei casi in cui i programmatori trovano utili il `goto`, ma al contrario di quest'ultimo non consente di cambiare in modo arbitrario il flusso dell'esecuzione.

2.4.2 Blocchi di inizializzazione statici

I blocchi di inizializzazione statici servono nel caso dei campi statici vadano inizializzati in maniera complessa. Supponiamo per esempio di aver bisogno in una classe di una variabile statica `primes` di tipo array di interi che vogliamo sia inizializzata con i primi 100 numeri primi. Ecco come potrebbe essere il codice.

```
class Primes  {
  static int pmax = 100;
  static int pi;
  static int[] primes = new int [pmax];
  static {
    Sieve first = new Sieve();
    int i = 2;
    while (i < pmax) {
      first.filter(i++);
    }
  }
}

class Sieve {
  int myPrime;
  Sieve next;
  void filter (int i) {
    if (myPrime == 0) {
      myPrime = i; Primes.primes[Primes.pi++] = i; next = new Sieve();
    }
    else if (i%myPrime!=0) next.filter(i);
  }
}

public class PNGTest {
  public static void main(String argv[]) {
  System.out.println (Primes.primes.length);
  for (int i = 0; i < Primes.primes.length; i++)
```

```
    System.out.println(Primes.primes[i]);
  }
}
```

Il blocco introdotto dalla parola chiave `static` viene eseguito quando la classe viene caricata.

2.4.3 Uso di `super`

Una caratteristica di Java ereditata da Smalltalk è che è possibile nel body di un metodo fare riferimento ai metodi della classe parent attraverso la parola chiave `super`. Questa parola chiave può essere utilizzata come ricevitore in un'invocazione di metodo, e l'effetto è lo stesso di un'invocazione con ricevitore `this`, tranne per il fatto che in questo caso la ricerca della versione di metodo da invocare inizia dalla classe parent. Come sempre, comunque, se non viene trovata una definizione del metodo si risale nella gerarchia delle classi. In questo modo è quindi possibile "recuperare" la versione del metodo della classe parent nel caso il metodo sia stato ridefinito. Tipicamente `super` viene utilizzato proprio quando si ridefinisce un metodo, in particolare nel caso di inheritance conservativa; infatti in questo caso è molto frequente che il nuovo body del metodo sia del tipo "invocazione di `super` seguita da ulteriore azione", perchè l'effetto che si vuole raggiungere con la nuova definizione è quello di arricchire l'effetto precedente con altre azioni presumibilmente relative ai nuovi campi. Vediamo un esempio.

```
class Cuboid  extends Rectangle {
  ...
  boolean equals (Cuboid c) {
    return super.equals(c) && height == c.height;
  }

  void init () {
    super.init ();
    height = 1;
  }
}
```

Vediamo ora alcune precisazioni sul significato di `super`.
Nel momento in cui si effettua una chiamata tramite `super` a un metodo della superclasse, supponendo che questo metodo richiami altri metodi e che questi siano a loro volta ridefiniti, si pone il problema di quali versioni di tali metodi vengono invocate: quelle della superclasse oppure quelle delle classe erede. La regola che vale è che la ricerca a partire dalla superclasse viene effettuata "una volta sola", quindi le ulteriori chiamate andranno a invocare i metodi della classe heir.
Questo è illustrato dal seguente esempio: la chiamata `o2.superResult`, in cui la classe del ricevitore è `Two`, stampa 2 perché il metodo `superResult` invoca il metodo `result` in `One`, che a sua volta effettua la chiamata `this.test`; la versione di `test` che viene invocata è quella della classe dell'oggetto corrente, cioè `Two`.

```
class One {
    int test () { return 1;}
    int result() { return this.test(); }
}

class Two extends One {
    int test () { return 2;}
    int superTest () { return super.test(); }
    int superResult () { return super.result();}
}

class Three extends Two {
    int test () { return 3;}
    int result () { return 3;}
}

class Four extends Three {
    int test () { return 4;}
}
```

```
public class SuperTest {
  public static void main(String[] argv) {
     One o1 = new One();
     Two o2 = new Two();
     Two o3 = new Three();
     Two o4 = new Four();
     System.out.println(o1.test());                    //1
     System.out.println(o1.result());                  //1
     System.out.println(o2.test());                    //2
     System.out.println(o2.result());                  //2
     System.out.println(o2.superTest());               //1
     System.out.println(o2.superResult());             //2
     System.out.println(o3.test());                    //3
     System.out.println(o3.result());                  //3
     System.out.println(o3.superTest());               //1
     System.out.println(o3.superResult());             //3
     System.out.println(o4.test());                    //4
     System.out.println(o4.result());                  //3
     System.out.println(o4.superTest());               //1
     System.out.println(o4.superResult());             //4
 }
}
```

L'esempio illustra anche un secondo fatto. Si consideri la chiamata `o4.superResult`, in cui la classe del ricevitore è `Four`. Viene invocato, come sopra, il metodo `superResult` in `Two`, il cui body consiste nella chiamata `super.Result`. Si noti che viene invocato il metodo `result` in `One`, cioè nella classe parent di `Two`, e non il metodo `result` in `Three`, cioè nella classe parent della classe dell'oggetto corrente.
Riassumento quanto detto, valgono le seguenti regole:

- il binding per `super` è sempre statico, cioè una chiamata `super.m(...)` invoca sempre la definizione di `m` nella superclasse della classe dove si trova la chiamata;
- tale binding statico vale solo al momento della chiamata di `super` e non influenza successive chiamate di metodo;
- infine, `super` va usato solo per invocare un metodo ridefinito o selezionare un campo in caso di hiding, vedi Sez.2.4.5 (va considerato a parte l'uso di questa parola chiave, come anche di `this`, all'interno del body di un costruttore, vedi la sezione successiva).

2.4.4 Costruttori

Abbiamo già introdotto in Sez.2.2.7 il meccanismo dei costruttori. Vediamo ora dettagliatamente le caratteristiche dei costruttori in Java.
All'inizio del body di un costruttore la parola chiave `this` assume un significato particolare: indica infatti l'invocazione di un altro costruttore, come illustrato dal seguente esempio.

```
class Rectangle{
  ...
  Rectangle (int l, int w) {
    length = l;
    width = w;
  }
  Rectangle () {
    this(1,1);
  }
  Rectangle (Rectangle r) {
    this(r.length,r.width);
  }
  ...
}
```

Se non vengono definiti costruttori, il linguaggio fornisce un costruttore senza argomenti predefinito `Rectangle ()`. La definizione del costruttore `Rectangle (int l,int w)` cancella il costruttore di default. In questo caso, se si vuole mantenere il costruttore costante bisogna definirlo esplicitamente, come nell'esempio.
L'invocazione di un altro costruttore tramite `this` può apparire solo come prima istruzione del corpo di un costruttore.
Vediamo ora qual è l'interazione tra costruttori e inheritance. Anzitutto, i costruttori non vengono ereditati, ma ogni classe definisce i propri. Tuttavia, in una classe è possibile fare riferimento ai costruttori della classe parent attraverso la parola chiave `super`. Vediamo un esempio.

```
class Cuboid  extends Rectangle {
  ...
  Cuboid () {
    super (); //superfluo
    height = 1;
  }
  Cuboid (int l, int w, int h) {
    super(l,w); height = h;
  }
  Cuboid (Rectangle r) {
    super(r); height = 1;
  }
  Cuboid (Cuboid c) {
    this(c.length,c.width,c.height);
  }
  ...
}
```

Si noti che il terzo costruttore nell'esempio corrisponde a un'operazione di conversione di tipo che permette di passare da un rettangolo a un parallelepipedo.
Il primo (e solo il primo) statement di un costruttore può quindi essere:

- una chiamata a un altro costruttore della stessa classe: `this (...)`
- una chiamata a un costruttore della superclasse: `super (...)`

Se non c'è una chiamata di costruttore esplicita come prima istruzione si assume una chiamata implicita `super ()`.
Il costruttore senza argomenti predefinito di una classe invoca semplicemente `super ()`, tranne quello della classe `Object` che non fa nulla.
I costruttori possono essere invocati esplicitamente in un'espressione `new C (...)`, da un altro costruttore attraverso le parole chiave `this` o `super`, oppure implicitamente come accade per la classe `String` (vedi Sez.2.4.12). Non essendo ereditati, i costruttori non possono essere dichiarati `abstract` o `final` (vedi Sez.2.4.11). Non possono neppure, per ovvi motivi, essere dichiarati `static`; infine, non sono ammesse chiamate ricorsive o mutuamente ricorsive di costruttori. Per quel che riguarda l'overloading di costruttori valgono le regole di risoluzione dell'overloading per i metodi (vedi Sez.2.4.6).
L'esecuzione di un costruttore consiste delle seguenti fasi:

1. esecuzione della chiamata iniziale (esplicita o implicita) a un altro costruttore;
2. se la precedente era una chiamata `super (...)`, esecuzione degli inizializzatori dei campi;
3. esecuzione degli altri statement.

Le regole precedenti implicano che quando si crea un'istanza di una classe si esegue un costruttore di tutte le superclassi (a partire da `Object`) e si inizializzano tutti i campi una volta sola.

2.4.5 Hiding di campi

Per quanto riguarda i campi non ha senso parlare di overriding, ma di *hiding*. Ossia, una classe heir può definire un campo con lo stesso nome di un campo della classe parent; l'effetto è che gli oggetti della classe heir hanno in realtà due campi, ma il campo della classe parent non può essere acceduto direttamente, ma solo attraverso `super` o un riferimento del tipo parent (anche ottenuto con un casting, vedi Sez.2.4.8). Si noti che il binding per i campi è sempre statico, come illustrato dal seguente esempio.

```
class One {
    String str = "One";
    void show () {
        System.out.println(str);
    }
}
class Two extends One {
    int str = 2;
    void show () {
        System.out.println(str);
    }
    void superShow () {
        System.out.println(super.str);
        super.show();
  }
}

public class HidingTest {
  public static void main (String[] args) {
    Two two = new Two();
    One one = two;
    one.show(); //2
    two.show();//2
    System.out.println(one.str);//One
    System.out.println(two.str);//2
    System.out.println(((One)two).str);//One
    two.superShow();//One, One
  }
}
```

L'hiding dei campi può quindi creare confusione, perchè, come illustrato dall'esempio, l'accesso attraverso un metodo si comporta diversamente dall'accesso tramite selezione diretta; tuttavia è permesso in Java per consentire di modificare una classe parent, aggiungendo un nuovo campo, dopo che una classe heir è stata definita senza rischiare di compromettere la correttezza del codice della classe heir. Si noti che adottando la soluzione "purista" che consiste nel dichiarare sempre privati i campi e accederli solo attraverso metodi di lettura e scrittura si evitano questi problemi.

2.4.6 Risoluzione dell'overloading

Abbiamo già visto che in Java è ammesso l'*overloading*, ossia una classe può avere più metodi (dichiarati direttamente o ereditati) con lo stesso nome che differiscono per la lista dei tipi dei parametri. Questi sono a tutti gli effetti *metodi diversi.* Analogamente una classe può dichiarare più costruttori che differiscono per la lista dei tipi dei parametri.

```
class C {
  void m() { ...}
  int m (int x) {...}
  C m (C x, Object y) {...}
}
```

Non è permesso invece che una classe possieda più metodi che differiscono solo per il tipo del risultato.
Quindi in una classe erede si avrà:

```
class D extends C {
  void m() { ...} // overriding
  int m (int x, int y) {...} // overloading
}
```

Come in tutti i linguaggi dove è permessa qualche forma di overloading, si ha il problema della *risoluzione*, cioè occorre definire delle regole per determinare quale metodo associare a una chiamata. Tale risoluzione è fatta staticamente.
La risoluzione è banale nel caso in cui vi sia nella classe del ricevitore un solo metodo *applicabile* alla chiamata (cioè, con il nome dato e con tipi dei parametri supertipi, rispettivamente, dei tipi degli argomenti).

```
class C {
  void m() { ...}
  int m (int x) {...}
  C m (C x, Object y) {...}
}
C c; ...
... c.m()... //scelgo il primo
... c.m(5)... //scelgo il secondo
... c.m(c,c)... //scelgo il terzo
```

Si noti che la risoluzione dell'overloading può essere vista come un preprocessing che sostituisce al nome un nome esteso con le informazioni di tipo. In questo senso è quindi profondamente diversa dalla scelta della versione di metodo da eseguire in base al binding dinamico descritta nella sezione 2.3.2 (vedi anche quanto detto nella sezione 2.1).
Nel caso invece vi siano più metodi applicabili la risoluzione è meno ovvia:

```
class D extends C { ... }

class C {
  void m( C x, C y) { ...}
  void m (C x, D y) {...}
  void m (D x, C y) {...}
}

C c; D d; ...
... c.m(c,d)... //scelgo il secondo
... c.m(d,c)... //scelgo il terzo
... c.m(d,d)... //ambigua
```

L' idea intuitiva è che viene scelta la versione "più specifica" tra quelle applicabili, e si ha una chiamata ambigua se non c'è una versione più specifica di ogni altra.
Vediamo un altro caso dubbio:

```
class C {
  void m( D x) { ...}
}
class D extends C {
  void m (C x) { ...}

C c; D d; ...
... d.m(d)... //scelgo il primo
```

In precedenti versioni di Java la chiamata era invece ambigua perché nella scelta della versione più specifica contava anche la classe in cui è dichiarato il metodo.
Diamo ora la regola precisa.
Sia `e_0.m(e_1, ..., e_n)` con `T_i` tipo (statico) di `e_i`

- Si cercano tutti i metodi *applicabili*, cioè: nome `m`, dichiarati in (supertipo di) `T_0` e tipi (**statici**) dei parametri `T'_i` compatibili, cioè `T_i` $\leq$ `T'_i` .
- Se non ci sono metodi applicabili si ha errore statico, se c'è un solo metodo applicabile lo si sceglie.
- Se c'è più di un metodo applicabile, si elimina via via ogni metodo *meno specifico* di un altro, dove:

  `m(T'_1, ..., T'_n)` meno specifico di
  `m(T''_1, ..., T''_n)`

  sse `T''_i` $\leq$ `T'_i` per ogni `i`.
- Se alla fine si resta con un solo metodo questo è quello da applicare, altrimenti si ha un errore statico (chiamata ambigua).

Si noti che, se si ha sia overloading che overriding, una volta stabilito *staticamente* il metodo da associare alla chiamata, a run-time per scegliere la versione da invocare interviene il binding dinamico:

```
class Parent {
  int m (Parent p) { return 0; }
  int m (Heir h) { return 1;}
}
class Heir extends Parent {
  int m (Parent p) { return 2; }
}
public class OverloadingTest {
  public static void main(String argv[]) {
    Parent p1 = new Parent ();
    Parent p2 = new Heir();
    Heir h = new Heir();
    System.out.println(p1.m(p1));//0
    System.out.println(p2.m(p1));//2
    System.out.println(p2.m(h));//1
    System.out.println(p1.m(p2));//0 (forse non proprio quello che vogliamo)
  }
}
```

Si noti che all'ultima riga il risultato ottenuto non corrisponde probabilmente a ciò che vorremmo (infatti l'argomento ha tipo dinamico `Heir`, quindi il metodo che ha un parametro di tipo `Heir` sembrerebbe il più adatto). Tuttavia questo è ciò che avviene in Java e in genere nei linguaggi object-oriented perché il binding dinamico è relativo solo al ricevitore e non agli argomenti. Potremmo ottenere di selezionare anche in questo caso il metodo `m(Heir)` in un linguaggio con *multimetodi*. È possibile ottenere anche in Java questo comportamento utilizzando l'operatore `instanceof` (vedi Sez.2.4.8) all'interno del metodo `int m(Parent)` in `Parent`.

2.4.7 Array

Gli array in Java sono considerati come particolari oggetti; la gestione è quindi diversa da quella del C e del C++. Vediamo un esempio.

```
int[] a = new int[3];
for (int i = 0;i < a.length;i++) a[i]=i;
```

Per ogni tipo `T`, esiste implicitamente il tipo `T[]` degli array con elementi di tipo `T`. Quindi per ogni classe `C` definita dall'utente è anche possibile dichiarare variabili di tipo `C[]`. Si può anche scrivere `int a[]`, ma la prima notazione va preferita in quanto suggerisce l'idea che `int[]` è un tipo.
Tutti i tipi array con elementi di tipi base sono sottotipi diretti della classe `Object`. Inoltre, se `Y` è sottotipo di `X`, `Y[]` sarà sottotipo di `X[]`.
I tipi array non possono essere estesi: non è possibile cioè dichiarare una classe erede di un tipo array.
La dimensione dell'array viene fissata al momento della creazione ed è accessibile tramite il campo `length`. Gli indici variano da `0` a `lenght - 1`. Se si tenta di accedere a un elemento di un array con un indice non compreso nel range valido viene sollevata un'eccezione `IndexOutOfBoundsException`.
Naturalmente è possibile definire array di array:

```
float[][] mat = new float[4][4];
```

o equivalentemente

```
float[][] mat = new float[4][];
for (int y = 0; y < mat.length; y++)
  mat[y] = new float[4];
```

L'inizializzazione viene fatta nel modo seguente:

```
int[][] a = {{1},{1,1},{1,2,1},{1,3,3,1}};
```

Si noti che ogni array annidato può avere una dimensione diversa.

2.4.8 Casting

Java è un linguaggio fortemente tipato nel senso che ogni espressione corretta ha un tipo, e il type-checking è effettuato prevalentemente a compile-time.
In alcuni casi tuttavia un'espressione può apparire correttamente in un contesto non del suo tipo. Si ha in questo caso una conversione di tipo. Java distingue cinque diversi contesti per la conversione di tipo: assegnazione, invocazione di metodo, casting, conversione a `String`, promozione numerica. In questa sezione tratteremo il casting tra tipi oggetto che è il contesto di conversione più interessante, in Sez.2.4.12 tratteremo la conversione a `String`, per gli altri contesti di conversione si rimanda a [4] Cap.5.
Un'espressione con casting è del tipo `(T)e` dove `T` è un tipo ed `e` è un'espressione. Il significato intuitivo dell'espressione `(T)e` è che l'espressione `e`, di un certo tipo `S` (per "source"), viene convertita al tipo `T`.
Consideriamo il caso in cui `T` e `S` sono due classi. Allora, perchè l'espressione sia corretta staticamente, `T` e `S` devono essere in relazione di inheritance (o essere la stessa classe), altrimenti si ha un errore statico.
Vi sono quindi due generi di conversione di tipo applicate in questo caso:

- casting da un tipo a un suo supertipo, detto *casting up*, *safe casting* o *widening*,
- casting da un tipo a un sottotipo, detto *casting down*, *unsafe casting* o *narrowing*.

Illustriamo ora la semantica dinamica del casting, cioè ciò che accade a run-time. Il casting up non implica alcuna azione a run-time (il suo unico effetto è quindi quello di cambiare il tipo statico dell'espressione). Il casting down implica invece un controllo a run-time:

- il casting ha successo (nel qual caso, non ha nessun effetto successivo) se e solo se il tipo dinamico `D` di `e` è sottotipo di `T`,
- altrimenti si ha un errore a run-time (più precisamente, viene sollevata l'eccezione `ClassCastException`).

Si noti che per il casting up non viene effettuato alcun controllo perché il successo a run-time è garantito. Infatti, $\mathtt{S} \leq \mathtt{T}$ e $\mathtt{D} \leq \mathtt{S}$ (infatti abbiamo visto che in base alle regole di tipo di Java il tipo di un'espressione durante l'esecuzione può unicamente diventare più specifico), quindi si ha $\mathtt{D} \leq \mathtt{T}$.
Si noti inoltre che la regola di semantica statica che richiede che `S` e `T` siano in relazione di inheritance è motivata dal fatto che, in caso contrario, non vi sarebbe alcuna possibilità di avere successo a run-time. Infatti, se `S` e `T` sono scorrelate, non è possibile che sia $\mathtt{D} \leq \mathtt{T}$ (poiché $\mathtt{D} \leq \mathtt{S}$ e l'inheritance è singola in Java). Se invece `S` e/o `T` è un'interfaccia, il caso `S` e `T` scorrelate è ammesso perché è possibile che, per qualche tipo dinamico `D`, sia $\mathtt{D} \leq \mathtt{T}$ e $\mathtt{D} \leq \mathtt{S}$.
Vediamo ora l'aspetto metodologico, ossia quando utilizzare i cast.
Il casting down è utile per accedere all'interfaccia più ricca del sottotipo in tutti i casi in cui ci aspettiamo che un'espressione con un certo tipo statico denoterà a run-time un oggetto di un sottotipo.

```
Rectangle r = new Cuboid();
int v = ((Cuboid) r).volume();
```

È possibile controllare prima la validità di un cast con l'operatore `instanceof`:

Sintassi e semantica statica `e instanceof T`, dove `T` è un tipo oggetto, `e` è un'espressione, è un'espressione di tipo `boolean`

Semantica dinamica Il valore di tale espressione è `true` se e solo se il tipo dinamico di `e` è sottotipo di `T`.

Un altro modo per accedere a run-time al tipo dinamico di un oggetto è fornito dal metodo `getClass()` della classe `Object` (vedi Sez.2.4.12)).
L'operatore `instanceof` può essere utilizzato per fare un'analisi per casi sul tipo dinamico degli argomenti di un metodo (in pratica, simulando ciò che avviene con i *multimetodi*, vedi alla fine di Sez.2.4.6).
Per esempio, una ridefinizione del metodo `equals` nella classe `Cuboid` potrebbe essere la seguente.

```
public boolean equals (Rectangle r) {
  if (r instanceof Cuboid) {
    return equals((Cuboid)r);
  }
  else return super.equals(r);
}
```

*NB: **In nessun caso** invece **l'operatore `instanceof` va utilizzato per fare un'analisi per casi** sul tipo dinamico del* ricevitore*, perché questo effetto in un linguaggio object-oriented può (**e deve!**) essere ottenuto automaticamente utilizzando una gerarchia di sottoclassi e il binding dinamico, come illustrato in Sez.2.3.3.*
L'utilità del casting up è meno evidente, dato che in genere in Java un'espressione di un sottotipo può comparire comunque in un contesto di un sopratipo. Tuttavia, ci sono alcune situazioni particolari in cui il tipo statico di un'espressione è rilevante ai fini del comportamento a run-time, e in questi casi un casting up può cambiare tale comportamento.
In particolare, il casting up può servire per:

- accedere a campi nascosti di una superclasse,
- accedere a campi e metodi privati di una superclasse (solo su oggetti di sottoclassi, dentro la superclasse),
- determinare una diversa risoluzione dell'overloading.

Ovviamente il casting up non ha alcuna influenza sul tipo *dinamico*.
Tutto questo è illustrato dal seguente esempio.

```
class One {
  String str = "One";
  void show () { System.out.println(str); }
}
class Two extends One {
  int str = 2;
  void show () { System.out.println(str);  }
}
public class CastingTest {
  static void test (One one) {
    System.out.println("test(One)");
  }
  static void test (Two two) {
    System.out.println("test(Two)");
  }
  public static void main (String[] args) {
    Two two = new Two();
    One one = two;
    System.out.println(((One)two).str);//One
    ((One)two).show();//2
    test(two);//test(Two)
    test((One)two);//test(One)
  }
}
```

2.4.9 Eccezioni

Le eccezioni sono uno strumento linguistico per la propagazione automatica e la gestione delle situazioni di errore. Gli ingredienti fondamentali di un meccanismo di eccezioni (presenti in molti linguaggi, anche in Caml) sono:

- Un costrutto linguistico per *sollevare un'eccezione*, in Java

  ```
  throw e ;
  ```

  dove `e` è un'espressione di tipo eccezione (in Java, una sottoclasse di `Exception`).

- Un costrutto linguistico per *catturare* un'eccezione, in Java

  ```
  try
    block
  catch (E_1 excp_1) block_1
  ...
  catch (E_n excp_n) block_n
  ```

La semantica dinamica di un blocco try è la seguente:

- Si esegue il blocco `block`.
- Se l'esecuzione di `block` termina normalmente l'esecuzione del try block termina normalmente.
- Altrimenti, cioè se l'esecuzione di `block` termina anormalmente sollevando un'eccezione di tipo `E`:
- Si controlla se `E` è sottotipo di $\mathtt{E}_i$ per qualche i.
- In caso positivo si esegue $\mathtt{block}_i$.
- Altrimenti (cioè se $\nexists i$ s.t. $\mathtt{E} \leq \mathtt{E}_i$) l'esecuzione del try block termina anormalmente sollevando l'eccezione `E`. In questo caso l'eccezione viene cioè *propagata* al blocco try più esterno se c'è oppure al chiamante del metodo, che a sua volta potrà catturarla o propagarla.
- Se nessun metodo cattura l'eccezione e si arriva al `main`, l'esecuzione del programma termina anormalmente sollevando l'eccezione `E`.

Si noti però che la descrizione data sopra non è precisa, perchè potrebbe essere $\mathtt{E} \leq \mathtt{E}_i$ per più di un i. In questo caso, si sceglie il primo i (in altri termini, le clausole `catch` vengono esaminate nell'ordine). Si ha quindi in particolare che un `catch` per un tipo dopo un `catch` per un suo supertipo non sarebbe mai eseguito; di conseguenza, questo è in Java un errore a compile-time (è infatti un caso particolare di *codice non raggiungibile*). La situazione opposta ha invece senso.
Oltre al meccanismo base per sollevare/catturare eccezioni comune a diversi linguaggi, Java offre in più uno strumento metodologicamente molto utile, cioè la possibilità di controllare staticamente quali eccezioni possono essere sollevate. Infatti in Java le eccezioni[18] sono *controllate* (*checked*). Ogni metodo deve dichiarare quali eccezioni possono essere sollevate dalla sua esecuzione, quindi un'intestazione di metodo prende la forma:

```
T m (T_1 x_1, ..., T_n x_n) throws E_1, ..., E_m { ... }
```

La clausola `throws` è un'indicazione per i clienti del metodo (li avverte che il metodo può potenzialmente causare certe situazioni di errore che andranno quindi gestite), e in più viene *controllata* dal compilatore, ossia:
Semantica statica: se il corpo del metodo può sollevare un'eccezione di tipo `E`, allora deve essere `E` sottotipo di $\mathtt{E}_i$ per qualche i.
Si noti che, quindi, una clausola `throws Exception` consentirebbe di poter sollevare qualunque eccezione nel corpo del metodo. Questo tuttavia vanificherebbe l'utilità del meccanismo delle eccezioni controllate, in quanto non verrebbe fornita alcuna informazione utile ai clienti del metodo (o, da un'altro punto di vista, li si obbligherebbe a gestire una situazione di errore assolutamente generica). In altre parole, è sempre metodologicamente opportuno scrivere una clausola `throws` il più specifica possibile, in modo da fornire maggiore informazione all'utente.
Come conseguenza della regola precedente, ogni metodo che invoca un metodo che dichiara un'eccezione nella sua `throws` clause deve fare una delle seguenti cose, altrimenti si ha un errore statico:

- catturare l'eccezione e gestirla;
- come caso particolare del precedente, catturare l'eccezione e "convertirla" in una propria eccezione (cioè dichiarata nella propria `throws` clause);
- dichiarare l'eccezione nella propria `throws` clause e propagarla.

Vediamo ora l'interazione tra clausole `throws` e overriding.

```
class Parent {
  ...
  T m (...) throws E_1, ..., E_n {...}
}
class Heir extends Parent {
  ...
  T m (...) throws E'_1, ..., E'_m {...}
}
```

[18]Più precisamente, i tipi eccezione dichiarati come sottoclassi di `Exception` e non di `RuntimeException`.

Se il metodo ridefinito dichiara nella propria clausola `throws` un certo tipo eccezione, allora il metodo nella classe parent deve dichiarare un supertipo, ossia:
$\forall i \in 1..m\ \exists j \in 1..n$ t.c. $\mathtt{E'}_i \leq \mathtt{E}_j$
(questo vale anche se la parent è una classe astratta che dichiara il metodo `abstract` o un'interfaccia).
Infatti, in caso contrario una chiamata del metodo potrebbe sollevare a run-time un'eccezione "non prevista". Si noti che non è richiesta alcuna relazione tra n e m: per esercizio si dia un esempio in cui n è maggiore di m e un altro in cui vale il viceversa.
Vediamo un esempio di implementazione del tipo di dato stack che utilizza gli array e gestisce le due possibili situazioni di errore in questo caso (pop sullo stack vuoto e push sullo stack pieno) con due eccezioni.

```
class StackByArray implements Stack {
  final static int maxLength = 15;
  int[] elems = new int[maxLength];
  int length = 0;
  public int pop () throws EmptyStackException {
    if (length==0) throw new EmptyStackException();
    return elems[--length];
  }
  public void push (int e)
    throws FullStackException {
    if (length == maxLength)
      throw new FullStackException();
    elems[length++]=e;
  }
}
```

Le eccezioni in Java sono oggetti, istanze di una classe heir di `Exception`. Quindi, nel programma occorrerà anche inserire le due seguenti dichiarazioni.

```
class EmptyStackException extends Exception{}
class FullStackException extends Exception{}

class stacksTest{
    public static void main (String[] args) throws EmptyStackException{
    Stack s = new StackByArray ();
    for (int i=1;i<20;i++)
      try { s.push(i);}
      catch (FullStackException excp) {
        System.out.println("Full Stack"); break;
      }
    for (int i=1;i<5;i++) { s.pop();}
  }
}
```

Nell'esempio sopra, si è optato, a scopo illustrativo, per una scelta diversa nei due casi: l'eccezione `FullStackException` è stata gestita con l'invio di un messaggio di errore definito dall'utente, mentre l'eccezione `EmptyStackException` non viene gestita, quindi va dichiarata nella clausola `throws` di `main`. In caso quest'ultima eccezione venga sollevata dall'esecuzione, vi sarà una gestione di default (in un programma "reale" va sempre prevista una gestione).
L'utilizzo di un meccanismo di eccezioni anzichè una propagazione e gestione delle situazioni di errore "a mano" offre grandi vantaggi in termini di modularità e chiarezza del codice. Elenchiamo qui alcuni principi metodologici.

- Ogni "tipo" di errore deve essere rappresentato con una diversa classe di eccezioni. Come gli altri oggetti, anche le eccezioni possono avere campi che servono a dare maggiori informazioni sull'errore avvenuto (per esempio la classe predefinita `Exception` ha un campo `String` utilizzato per memorizzare un messaggio di errore). Può essere utile definire una gerarchia di classi di eccezioni corrispondente a errori via via più specifici. In particolare, tutte le eccezioni relative a uno stesso modulo (per esempio package o singola classe) possono essere eredi di una stessa classe radice.

- Ogni modulo classe dichiara nelle clausole `throws` dei propri metodi le situazioni di errore che si potrebbero verificare nell'utilizzare il modulo. Le eccezioni fanno quindi parte della *specifica* del modulo.

- La gestione di tali errori è a carico dell'utilizzatore del modulo; la gestione della situazione di errore non va effettuata "prima del tempo" (per esempio, è inutile gestire venti volte una situazione di errore in venti metodi diversi, se basta gestirla una sola volta nello stesso modo nel metodo che li chiama).

- Infine, le eccezioni vanno utilizzate per effettive situazioni di errore, che non ci si aspetta si verifichino "normalmente"; non è buona programmazione utilizzarle per gestire situazioni che ci si aspetta si verifichino prima o poi, come per esempio per controllare la fine di uno stream di input o per interrompere una ricerca.

Le espressioni di inizializzazione dei campi non possono sollevare eccezioni controllate; se occorre sollevare un'eccezione si possono inizializzare i campi di istanza nei costruttori.
Infine, un blocco `try` può avere anche una parte `finally` che viene eseguita qualunque sia il modo in cui si esce dal blocco (normalmente o perchè è stata sollevata un'eccezione, eventualmente catturata da un `catch`). È un'altra caratteristica di Java che permette al programmatore di fare le stesse cose che si potrebbero fare con un `goto`, ma mantenendo una struttura a un ingresso e un'uscita.
La sintassi completa di un blocco `try` è quindi la seguente, dove le parentesi quadre denotano opzionalità:

```
try
  block
catch (E_1 excp_1) block_1
...
catch (E_n excp_n) block_n
[finally fblock] // obbligatorio solo se n==0
```

Diamo qui di seguito una versione dell'implementazione delle liste vista in Sez.2.3.3 arricchita con un'opportuna gestione degli errori tramite eccezioni.

```
class BadIntervalException extends Exception {
  BadIntervalException (int a, int b) {
   super(b+"<"+a);
  }
class EmptyListException extends Exception {}

class EmptyList extends List{
  boolean isEmpty () { return true;}
  int head () throws EmptyListException { throw new EmptyListException();}
  List tail () throws EmptyListException { throw new EmptyListException();}
  }
}

class IntervalList extends List {
  private int a,b;

  IntervalList(int a, int b) throws BadIntervalException {
    if (b<a) throw new BadIntervalException(a,b);
    this.a =a; this.b = b;
  }
  boolean isEmpty() { return false;}
  int head() { return a;}
  List tail() {
    if (b > a) {
      try {
         return new IntervalList(a + 1,b);
      }
      catch (BadIntervalException excp) { throw  new InternalError();};  //impossibile
    }
    else return new EmptyList();
  }
 }
}
class NonEmptyList extends List {
  private int head;
  private List tail;
  NonEmptyList(int elem, List list) {
      head = elem;
    if (list.isEmpty()) {
```

```
        tail = List.emptyList();
    }
    else
      try {
        tail = new NonEmptyList(list.head(),list.tail());
      }
      catch (EmptyListException excp) {throw  new InternalError();}; //impossibile
  }
  boolean isEmpty () { return false;}
  int head () { return head;}
  List tail () { return tail;}
   }
}

abstract class  List {
  abstract boolean isEmpty ();
  abstract int head() throws EmptyListException;
  abstract List tail() throws EmptyListException;
}
```

2.4.10 Package

Java permette di raccogliere diverse classi e interfacce in un *package*, che può essere composto da diversi file (unità di compilazione). L'utilità dal punto di vista metodologico è quella di raggruppare più classi e interfacce logicamente correlate, poter usare gli stessi nomi in più package e utilizzare nomi locali inaccessibili dall'esterno.
Una dichiarazione di package `P` all'inizio del file indica che le classi e le interfacce nel file appartengono al package `P`.
Per esempio:

```
package stacks;
public interface Stack {
  boolean isEmpty();
  int pop();
  void push (int e);
  void print ();
}
```

L'interfaccia `Stack` appartiene al package `stacks`.
Le classi e interfacce dichiarate `public` sono esportate all'esterno del package. Vale in genere la regola che ogni file possa contenere al più una componente pubblica e, se la contiene, il nome del file (senza l'estensione `.java`) deve coincidere col nome di tale componente. Inoltre, tutti i file di un package si devono trovare in una stessa directory che deve avere lo stesso nome del package. I sottopackage corrispondono quindi alle sottodirectory e il loro nome deve corrispondere al nome completo di path. Per esempio, `stacks.integer_stacks` è il sottopackage di `stacks` i cui files sono contenuti nella directory `stacks/integer_stacks`.
Vi sono due modi per utilizzare una componente `C` (`public`) di un package `P`:

- qualified notation: `P.C`
- import esplicito: `import P.C` (`import P.*` per importare tutte le componenti pubbliche - *non* i subpackage).

In caso di conflitto di nomi occorre usare il primo modo.
Le classi predefinite (Application Programming Interface) sono organizzate in una gerarchia di package con radice `java`. Le componenti del package `java.lang` sono importate automaticamente.
Se un file sorgente non contiene alcuna dichiarazione di package, tutto il suo contenuto fa parte di un unico *unnamed package*.
Si noti che il comando di `import` di Java è molto diverso da `#include` del C; quest'ultimo ha l'effetto di un'espansione del codice, mentre `import` si comporta come un linker, cioè indica al compilatore dove si trovano le componenti necessarie. Più precisamente, Java usa due cose per trovare una classe: il nome dei package e le directory elencate nella variabile d'ambiente `CLASSPATH`.

2.4.11 Visibilità e modificatori

Con l'uso di package, i livelli di visibilità di campi e metodi di una classe diventano quattro.

- `private`: visibili solo nella classe che li contiene
- `package` (ossia dichiarati senza alcun access modifier): visibili solo nel package che li contiene (sostituisce la nozione di visibilità dentro lo stesso file di testo del C);
- `protected`: visibili nel package che li contiene e in ogni sottoclasse `C` (solo attraverso oggetti almeno di tipo `C`);
- `public`: visibili ovunque la classe che li contiene sia visibile.

Come più volte ripetuto, una tecnica abbastanza standard nell'object-oriented è quella di dichiarare `private` le variabili di istanza e di definire dei metodi appositi per leggerle/aggiornarle (per esempio `getLength, setLength`). Questo permette di avere variabili read-only. Conviene usare una convenzione uniforme per i nomi di questi metodi (un'altra possibile è di usare lo stesso nome della variabile).
In generale, non si può ridurre la visibilità di un metodo, ma solo estenderla. Più in dettaglio:

- i metodi dichiarati `public` in una superclasse devono essere `public` nelle sottoclassi,
- i metodi dichiarati `protected` in una superclasse devono essere `protected` o `public` nelle sottoclassi,
- i metodi dichiarati `private` non sono visibili nelle sottoclassi, quindi non possono essere ridefiniti,
- i metodi dichiarati senza protezione (quindi `package`) non possono essere `private` nelle sottoclassi.

Chiariamo infine con un esempio il significato preciso del modificatore `protected`. Supponiamo che le classi `Rectangle` e `Cuboid` non facciano parte dello stesso package.

```
class Rectangle {
  protected int length , width;
  ...
}
class Cuboid  extends Rectangle {
  ...
  void test () {
  System.out.println(this.length);//ok
//  System.out.println(new Rectangle().length); //errore statico
  }
}
```

Riassumiamo ora brevemente quali sono tutti i *modificatori* che possono essere utilizzati. I modificatori relativi alla visibilità (*access modifiers*), `public`, `protected` e `private` sono stati descritti sopra.
Il modificatore `final` può essere utilizzato per variabili, metodi e classi. Una variabile `final` è una costante. Un metodo dichiarato `final` non può essere ridefinito nelle sottoclassi. Una classe dichiarata `final` non può essere estesa. Si noti che dichiarare una classe `final` è più forte che dichiarare tutti i metodi della classe `final`, perché in quest'ultimo caso la classe può ancora essere estesa con nuovi metodi.
Metodologicamente, l'uso di metodi e classi `final` ha essenzialmente due scopi:

- garantire che il comportamento di un metodo o classe non cambi;
- ottimizzazione del codice, infatti per un metodo `final` la versione del metodo da invocare è immediatamente determinabile senza bisogno del meccanismo di ricerca.

Quindi, dichiarare una classe o un metodo `final` aumenta l'efficienza; si noti però che in questo modo non si può più estendere la classe o ridefinire il metodo, quindi si perde il beneficio essenziale di poter riutilizzare software anche in assenza del codice sorgente. I metodi che leggono o aggiornano le variabili di istanza possono tipicamente essere `final`. I metodi privati e quelli dichiarati in classi `final` sono di fatto `final`; lo stesso vale per i metodi di classe.
Il modificatore `abstract` può essere utilizzato per classi e metodi (vedi Sez.2.3.6). Ricordiamo che le classi astratte non possono avere istanze.
L'ordine usuale dei modificatori è il seguente:
$\langle$ access $\rangle$ `static abstract final`
Dichiarare un metodo `private abstract` o `final abstract` produce un errore di compilazione.
NB: I modificatori, ad eccezione di `final`, ***non si applicano alle variabili locali!***

2.4.12 Cenni alle classi predefinite

Le classi predefinite costituiscono nel loro insieme la cosiddetta Application Programming Interface, e sono suddivise nei seguenti package.

java.lang Viene importato automaticamente. Comprende le classi `Object`, `String`, `Class`, le classi corrispondenti ai tipi primitivi, etc.

java.util Contiene `Date` e classi per collezioni come `Vector`, `Hashtable`, etc.

java.io Tutte le classi relative all'input/output.

java.net

java.awt

java.applet

La classe Object La classe `Object` è la radice della gerarchia delle classi, quindi tutti gli oggetti implementano i metodi di questa classe. Alcuni metodi interessanti forniti sono i seguenti.

boolean equals (Object) Controlla se due oggetti sono uguali. L'implementazione di default coincide con ==. Si veda la discussione sui vari tipi di uguaglianza tra oggetti in Sez.2.2.2. Questo metodo può essere sia overriden che overloaded.

int hashCode () Restituisce un hash code che può essere usato per memorizzare oggetti in una `Hashtable`. Va ridefinito in accordo con `equals`, nel senso che oggetti uguali nel senso di `equals` devono avere lo stesso `hashCode`.

protected Object clone () throws CloneNotSupportedException Restituisce una copia (di default la shallow copy, vedi Sez.2.2.2) del ricevitore. Ogni classe che ridefinisce `Object clone ()` deve implementare l'interfaccia `Cloneable`, altrimenti invocare questo metodo solleva un'eccezione. Si ricordi che, dato che il metodo `Object clone ()` restituisce un `Object`, va utilizzato un cast se si vuole ottenere un risultato di un tipo più specifico. Questo metodo non può infatti essere overloaded perchè non ha parametri.

final Class getClass() Restituisce un oggetto di tipo `Class` che rappresenta la classe dell'oggetto.

protected void finalize () throws Throwable Viene invocato dal garbage collector immediatamente prima di cancellare un oggetto. Di default non fa nulla.

String toString() Restituisce una rappresentazione come stringa del ricevitore. Di default restituisce il nome della classe + l'hashcode del ricevitore.

Tutti questi metodi sono in genere opportunamente ridefiniti nelle classi predefinite.

La classe String Le stringhe sono istanze di una classe predefinita `String`. Sono quindi oggetti e non valori. Possono essere create implicitamente scrivendo un literal:

```
String myname = "Mario";
```

L'operatore di concatenazione di stringhe si indica con +:

```
myname = myname + " Rossi";
```

Il metodo `length` restituisce la lunghezza. I caratteri che compongono la stringa sono indiciati da `0` a `length() -1` e possono essere ottenuti con il metodo `charAt(int)`. Il metodo `equals` è ridefinito e restituisce vero solo se due stringhe sono uguali.
Il metodo `toString` della classe `Object`, come visto sopra, restituisce una stringa corrispondente a una rappresentazione "testuale" di un oggetto della classe. Questo metodo viene invocato automaticamente quando un oggetto appare come operando dell'operatore + (oppure +=), ossia quando il contesto richiede un operando di tipo `String`. Gli oggetti di tipo `String` sono immutabili, cioè read-only. Stringhe modificabili sono fornite dalla classe `StringBuffer/StringBuilder`.

La classe `Class` La classe `Class` rende possibile esaminare a run-time la gerarchia di classi e interfacce di un programma. Le istanze di questa classe sono infatti rappresentazioni a run-time delle classi e interfacce. Vi sono due modi per ottenere un oggetto di tipo `Class`: attraverso il metodo `getClass()` di `Object` e attraverso il metodo statico `Class.forName (String)` che restituisce, se esiste, l'oggetto classe associato a un certo nome.
Esempi di metodi definiti in `Class` sono i seguenti:

- `boolean isInterface ()`, che controlla se si tratta di una classe o di un'interfaccia,
- `String getName ()`, che restituisce la stringa corrispondente al nome della classe/interfaccia
- `Class[] getInterfaces ()` , che restituisce le interfacce implementate,
- `getSuperclass ()`,
- `Object newInstance ()`, che invoca il costruttore senza argomenti; serve per scrivere codice indipendente dalla classe.

In J2SE 5 la classe `Class` è diventata una classe parametrica `Class<T>` (ossia per ogni classe `C` la sua rappresentazione a run-time è un'istanza di `Class<C>`).

Le classi wrapper dei tipi primitivi Queste classi (una per ogni tipo primitivo) servono per raggruppare metodi statici, costanti e metodi logicamente correlati e per manipolare come oggetti dei valori di tipi primitivi, per esempio per inserirli in una `Hashtable`. Ogni classe wrapper fornisce:

- un costruttore con argomento del tipo primitivo corrispondente, ad esempio `Integer (int)`,
- un costruttore con argomento `String`,
- un'implementazione opportuna del metodo `toString`,
- un metodo di conversione al tipo primitivo corrispondente, ad esempio `int intValue ()` in `Integer`,
- un'implementazione opportuna dei metodi `equals` e `hashCode`.

Inoltre ogni classe wrapper fornisce moltissimi altri metodi utili.
In Java 5 le operazioni di *boxing* e *unboxing*, cioè il passaggio da un valore di tipo primitivo all'oggetto wrapper corrispondente, per esempio da un `int` a un `Integer`, e viceversa, sono diventate automatiche. Ossia, è possibile omettere il costruttore e il metodo `intValue` e la conversione avviene automaticamente.

Le classi `Exception` ed `Error` La classe `Exception` è la classe di tutte le situazioni di errore che un'applicazione potrebbe voler gestire con `catch`.
Esempi di eccezioni predefinite sono `ClassNotFoundException`[19], `CloneNotSupportedException`, `IOException`.
Sono controllate, cioè devono essere dichiarate esplicitamente nell'intestazione dei metodi che possono sollevarle (tranne le `RuntimeException`, che possono essere sollevate da qualunque metodo, come `NegativeArraySizeException`, `ArithmeticException`, `ClassCastException`, `IndexOutOfBoundsException`, `NullPointerException`).
Le classi di eccezioni dichiarate dal programmatore, per utilizzare il meccanismo di controllo statico, devono essere sottoclassi di `Exception` che non siano sottoclassi di `RuntimeException`.
Invece `Error` è la classe delle situazioni di errore che un'applicazione normalmente non si occupa di gestire (per esempio `NoClassDefFoundError`[20], `OutOfMemoryError`, `StackOverflowError`).

Input-Output Vediamo un semplice esempio di programma che utilizza il pagkage `java.io`.

```
import java.io.*;
public class IOTest {
  public static void main(String[] args) throws IOException {
    InputStream in;
    if (args.length == 0) in = System.in;
    else in = new FileInputStream(args[0]);
    int ch;
```

[19]Sollevata se si cerca, per esempio tramite il metodo `Class forName (String)`, un oggetto classe non esistente.
[20]Sollevata dalla JVM se tenta di caricare una classe che non c'è più (per esempio perché il file `.class` è stato cancellato).

```
    int total;
    int spaces = 0;
    for (total = 0; (ch = in.read()) != -1; total++) {
      if (Character.isWhitespace((char)ch)) spaces++;
    }
    System.out.println(total + " chars, " + spaces + " spaces");
}
}
```

L'input-output in Java è basato sulla nozione di *stream* (sequenza di dati, in genere byte). L'input è gestito attraverso sottoclassi di `InputStream` e l'output attraverso sottoclassi di `OutputStream`. Una stream di input può essere ottenuta a partire da un file (come nell'esempio sopra), una stringa, un array di byte, una stream di output; analogamente per una stream di output.
Le istanze della classe `File` rappresentano nomi di file completi di path in un sistema operativo; le istanze della classe `FileDescriptor` forniscono una rappresentazione astratta dei file.
Vi è anche una classe `StreamTokenizer` che fornisce un semplice scanner adattabile (vedi Sez.1.7).
Per maggiori dettagli sull'input-output si rimanda alla documentazione on-line dell'API.

2.4.13 Riassunto, caratteristiche non trattate e confronto con C,C++

Riassumendo quanto visto, possiamo dire che Java è un linguaggio object-oriented puro con inheritance singola e garbage collector automatico. È un linguaggio con type-checking prevalentemente statico, ha un'implementazione mista (compilato in bytecode, poi interpretato). È indipendente dalla piattaforma: infatti il linguaggio intermedio (bytecode) è indipendente dall'architettura; inoltre vi sono specifiche fissate per i tipi base. Inoltre supporta *multiple threads* (un aspetto che non consideriamo nel corso) e compilazione separata (nel senso che una classe può essere compilata avendo a disposizione solo il bytecode, e non il codice sorgente, delle classi usate).
Oltre agli aspetti di concorrenza, non trattiamo nel corso tutte le caratteristiche di Java relative alle interfacce grafiche (package `java.awt/javax.swing`) e a Internet (`java.net`, `java.applet`). Rimandiamo alla documentazione on-line per approfondimenti relativi ai package `java.lang`, `java.util` e `java.io`.
Inoltre, non trattiamo alcune estensioni molto importanti e potenti introdotte successivamente nel linguaggio: le *classi inner* (possibilità di dichiarare in una classe componenti che sono a loro volta classi) e le classi generiche introdotte in J2SE 5.
I progettisti originali di Java hanno seguito due principi opposti: *simple*[21] e *familiar*. Hanno cercato, cioè,

- da un lato di rimuovere sistematicamente tutte le caratteristiche "superflue" o "dubbie" di C e C++;
- dall'altro di salvare "l'apparenza" simile a questi due linguaggi e a altri linguaggi noti come Eiffel, Smalltalk, Ada.

Riassumendo, le differenze fondamentali tra Java e C,C++ sono le seguenti:

- Java ha il tipo `boolean` che **non** può essere convertito a un altro tipo;
- gli array sono oggetti come gli altri e non puntatori;
- le stringhe sono oggetti (classe predefinita `String` e `StringBuffer`) e non array di caratteri;
- Java non ha `goto` ma ha `break` etichettati;
- Java ha garbage collection automatica (come ogni linguaggio object oriented "puro");
- Java **non ha puntatori** (come ogni linguaggio object oriented "puro"), nel senso discusso in Sez.2.2.3;
- Java non ha `typedef`: in molti casi in cui in `C` si utilizzerebbe `typedef` in Java si usa una dichiarazione di classe, anche se il meccanismo è completamente diverso perchè `typedef` è una semplice abbreviazione;
- Java non ha `define`: l'effetto può essere in genere ottenuto con una dichiarazione di costante; vale l'osservazione fatta al punto precedente;
- Java non ha `struct` e `union` (l'effetto si ottiene dichiarando una gerarchia di classi);
- Java non permette al programmatore di definire versioni overloaded dei simboli di operatore (come +), come avviene invece in C++, e in generale ha un severo controllo di tipo.

[21] Questa semplicità iniziale è decisamente andata persa nelle ultime versioni!

Per contro, vi sono le seguenti caratteristiche comuni:

- i tipi di dato primitivi sono essenzialmente gli stessi,
- gli statement sono essenzialmente gli stessi,
- la tecnica per inizializzare oggetti è sostanzialmente quella del C++ (costruttori).

Riassumendo potremmo dire che Java è una versione "ripulita" in senso object oriented puro di C++ (che è un linguaggio a paradigma misto), con alcune caratteristiche aggiuntive (eccezioni controllate, threads, ...); si può anche vederlo come un linguaggio object oriented che usa C come linguaggio in cui scrivere i body dei metodi.

3 Paradigma Funzionale

In questo capitolo illustreremo il paradigma di programmazione funzionale utilizzando come linguaggio di riferimento il linguaggio Caml, (`http://caml.inria.fr/`, sviluppato all'INRIA a partire dal 1985), che appartiene alla famiglia di linguaggi ML. Gli esempi di codice sono stati provati con la versione `Objective Caml 3.06`.

3.1 Concetti base

Il linguaggio Caml è un linguaggio funzionale higher-order, polimorfo, con inferenza di tipo. Vediamo il significato di questi termini.

Espressioni che denotano funzioni L'idea base del paradigma funzionale è quella di avere come modello le funzioni "matematiche". Si tratta quindi di un paradigma molto diverso da quello imperativo: in particolare, non vi è alcuna nozione di stato, variabile, assegnazione, esecuzione sequenziale di comandi.
I costrutti linguistici base sono quindi: definizione di funzioni (dichiarazione) e applicazione (chiamata).
Un'importante osservazione è quella che nel definire una funzione il nome è irrilevante: $f(x) = x+1$ e $g(x) = x+1$ definiscono la stessa funzione. Infatti talvolta in matematica si usa la notazione $x \mapsto x + 1$.
Una delle idee chiave del λ-*calcolo*, un formalismo introdotto negli anni '30 da Alonso Church e Stephen Cole Kleene, è stata appunto quella di poter scrivere espressioni corrispondenti a funzioni in modo indipendente dal nome, per esempio $\lambda x.x + 1$.
Il lambda-calcolo costituisce il modello computazionale dei linguaggi funzionali (in realtà tutti i linguaggi vi possono essere codificati).
In Caml è infatti possibile scrivere `function x -> x +1`. Questa è un'*espressione di tipo funzionale* che denota la funzione successore.
Una *definizione* di funzione ha la forma: `let succ = function x -> x+1` oppure `let succ(x) = x + 1`. In entrambi i casi, l'effetto della definizione è quello di introdurre un nuovo *nome* per la funzione successore, in questo caso `succ`.
In altri termini, dato che è possibile scrivere espressioni di tipo funzionale, è lecito passare da una definizione del tipo `let f(x) = e`, con `e` espressione che fa riferimento a `x`, a una del tipo `f = function x -> e`. Tale passaggio si chiama "astrazione funzionale"[22]. In particolare, una definizione del tipo `let f (x) = g(x)` può essere più sinteticamente scritta `let f = g`; infatti, se due funzioni sono uguali su ogni elemento, allora sono uguali.
L'*applicazione* di una funzione `f` a un argomento `e` può essere scritta `f (e)` come nell'uso matematico; tuttavia, le parentesi non sono necessarie e si può scrivere semplicemente `f e` (si usa questa notazione per evitare la proliferazione delle parentesi).
Per esempio le seguenti sono espressioni Caml corrette (avendo dichiarato precedentemente la funzione `succ` come sopra):

```
succ 2
(function x -> x +1) 2
```

Convenzioni sintattiche Si usano le seguenti convenzioni per disambiguare:

- `e_1 e_2 e_3` sta per `(e_1 e_2) e_3`
- `function x -> e_1 e_2` sta per `function x -> (e_1 e_2)`

Inoltre, nei tipi delle funzioni (vedi dopo), $T_1 \to T_2 \to T_3$ sta per $T_1 \to (T_2 \to T_3)$.

[22]Tuttavia, la seconda forma non permette ricorsione, vedi dopo.

Funzioni higher-order Le funzioni sono "valori di prima classe" ossia possono essere manipolate esattamente come gli altri valori (interi, booleani, etc). Quindi è possibile scrivere funzioni che prendono come argomento o restituiscono come risultato altre funzioni. Queste funzioni si dicono *di ordine superiore* (*higher-order*) o anche *funzionali*. Per esempio la composizione di due funzioni può essere definita in uno dei seguenti modi (tutti equivalenti):

```
let compose (f, g) x = f (g x)
let compose (f, g) = function x -> f (g x)
let compose = function (f, g) -> function x -> f (g x)
```

Analogamente, la seguente è una funzione che su ogni argomento restituisce il valore ottenuto applicando due volte una funzione data:

```
let double f x = f (f x)
```

Utilizzando funzioni higher-order è possibile "generalizzare" delle definizioni, ossia astrarre rispetto a un particolare valore dell'argomento. Per esempio, la seguente funzione calcola la somma dei quadrati dei numeri da uno a `n`:

```
let rec sumsquare n = if n<=0 then 0 else n*n + sumsquare (n-1)
```

Possiamo generalizzare la definizione a quella di una funzione che calcola la somma dei valori di una funzione data sui numeri da uno a `n`

```
let rec sum f n = if n<=0 then 0 else f n + sum f (n-1)
```

e poi definire `sumsquare` come caso particolare (*applicazione parziale*, vedi più avanti):

```
let sumsquare = sum (function x -> x*x)
```

Analogamente, possiamo definire la funzione costante 5:

```
let k5 x = 5
```

oppure ottenerla come caso particolare (`k 5`) della funzione higher order

```
let k x y = x
```

Uso del sistema L'utente interagisce con il sistema, inserendo nuove espressioni o definizioni. Il sistema stampa il prompt # prima di leggere l'input dell'utente, che deve terminare con `;;`. Negli esempi seguenti scriveremo esplicitamente prompt e terminatore per illustrare una sessione di lavoro e le corrispondenti risposte del sistema. Per esempio:

```
# 1+2;;
- : int = 3
```

L'interprete stampa il tipo dell'espressione (vedi dopo) e, nel caso di un'espressione di tipo non funzionale, il valore. Una definizione viene conservata per tutta la durata della sessione di lavoro (una nuova definizione per lo stesso nome cancella la vecchia).

```
# let succ x = x+1;;
val succ : int -> int = <fun>
# succ 2;;
- : int = 3
# let even n = (n mod 2)=0;;
val even : int -> bool = <fun>
```

Si noti che tuttavia altre definizioni che utilizzano un nome la cui definizione viene successivamente cambiata continuano a fare riferimento alla definizione vecchia, come illustrato dal seguente esempio.

```
# let y=5;;
val y : int = 5
#  let add_y= function x-> x + y;;
val add_y : int -> int = <fun>
# add_y 8;;
- : int = 13
# let y=10;;
val y : int = 10
# add_y 8;;
- : int = 13
# (function x->x+y) 8;;
- : int = 18
```

Questo è dovuto al fatto che nelle definizioni di funzione si utilizza un binding statico per i nomi globali, come avviene nella maggior parte dei linguaggi di programmazione. Nell'esempio, il valore che viene associato al nome `add_y` è la funzione `x -> x + 5`; più precisamente, il valore che viene associato è una cosiddetta "chiusura", cioè la tripla: $(\mathtt{x}, \mathtt{x+y}, [5/\mathtt{y}])$ (lista dei parametri, corpo della funzione, ambiente al momento della dichiarazione). Si confronti con la semantica formale del linguaggio didattico data nella prima parte delle dispense (Fig.7).

Funzioni polimorfe Una funzione *polimorfa* è applicabile ad argomenti di tipi diversi. Un esempio molto semplice è la funzione identità $f(x) = x$, che "ha senso" su qualunque tipo. Si noti la differenza con overloading/overriding: polimorfismo significa un'unica definizione valida per tipi diversi, mentre nel caso dell'overloading/overriding si ha lo stesso nome per diverse definizioni[23]. Nell'esempio seguente, la funzione `first`, che restituisce il primo elemento di una coppia, può essere applicata sia a due interi, che a due booleani, che a due funzioni. Analogamente, la funzione `compose` può essere utilizzata sia per comporre una funzione da interi in booleani e una da interi a interi, che per comporre due funzioni da interi a interi; in entrambi i casi infatti si ha che il dominio della prima funzione è uguale al codominio della seconda funzione.

```
# let first (x,y) = x;;
val first : 'a * 'b -> 'a = <fun>
# first (1,2);;
 - : int = 1
# first (true, false);;
 - : bool = true
# first (succ, even);;
int -> int = <fun>
# let compose (f, g) = function x -> f (g x);;
val compose : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
# compose (even, succ);;
 - : int -> bool = <fun>
# compose (succ, succ);;
 - : int -> int = <fun>
```

Il tipo di una funzione polimorfa è uno *schema di tipo*, cioè contiene delle *variabili di tipo* che indicano un tipo arbitrario. Indicheremo le variabili di tipo con le lettere greche $\alpha, \beta, \gamma, \delta, \ldots$. Come si vede dagli esempi precedenti, nel sistema Caml vengono utilizzate lettere minuscole precedute da un apice. Intuitivamente, lo schema di tipo "rappresenta" tutti i diversi tipi che la funzione può assumere quando viene applicata a degli argomenti; per questo motivo si chiama anche il tipo *più generale* (o *principale*) della funzione. Tutti i tipi più specifici possono essere ottenuti istanziando opportunamente le variabili di tipo. Per esempio, il tipo più generale della funzione `compose` è, come visto sopra:

$$(\alpha \to \beta) * (\gamma \to \alpha) \to \gamma \to \beta$$

Nell'espressione `compose (even, succ)` il tipo della funzione `compose` è il seguente:

$$(\mathtt{int} \to \mathtt{bool}) * (\mathtt{int} \to \mathtt{int}) \to \mathtt{int} \to \mathtt{bool}$$

che si ottiene istanziando α e γ con `int` e β con `bool`.

[23] Questa differenza fu descritta per la prima volta da Christopher Strachey nel 1967 con i termini di *polimorfismo parametrico* e *polimorfismo ad-hoc*.

Inferenza di tipo Possiamo notare negli esempi visti sopra che il programmatore non inserisce mai annotazioni di tipo esplicite nel codice, come avviene invece per esempio in C o in Java. Infatti, i tipi delle espressioni vengono *dedotti* automaticamente (non vedremo l'algoritmo preciso; si veda la Sez.3.7 per un'illustrazione informale di come si può dedurre il tipo di un'espressione).

Applicazione parziale Si consideri il seguente esempio.

```
# let sum (x,y) = x+y;;
val sum : int * int -> int = <fun>
# sum (2,5);;
- : int = 7
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 2 7;;
- : int = 9
# add 2;;
- : int -> int = <fun>
```

La funzione `add` calcola, come la funzione `sum`, la somma di due interi. Tuttavia, mentre `sum` prende come argomento una coppia di interi, la funzione `add` prende come argomento *un* intero e restituisce una *funzione* che, preso ancora un argomento intero, restituisce un intero. Le funzioni che prendono "un argomento per volta" sono dette *curried* (vedi sotto). Il vantaggio che offrono (per cui in genere sono preferite nel paradigma funzionale) è quello di poter effettuare un'*applicazione parziale*, ottenendo altre funzioni come casi particolari; nell'esempio sopra, l'espressione `add 2` denota la funzione che aggiunge due a un numero intero; è anche possibile dare un nome a questa espressione:

```
# let succsucc = add 2;;
val succsucc : int -> int = <fun>
```

Currying Chiamiamo *currying* (dal logico Haskell Curry) la trasformazione che permette di ottenere, data una funzione che prende come argomento una coppia $f\colon A \times B \to C$, una funzione $\tilde{f}\colon A \to (B \to C)$ (la sua versione "curried") così definita: per ogni $a \in A$,

$\tilde{f}(a)\colon B \to C$,
per ogni $b \in B$, $(\tilde{f}(a))(b) = f(a, b)$

È possibile anche dare la trasformazione inversa, cioè, data una funzione $g\colon A \to B \to C$, definirne la versone "uncurried" $\bar{g}\colon A \times B \to C$, come segue: per ogni $a \in A$, $b \in B$,

$$\bar{g}(a, b) = (g(a))(b)$$

Per esempio, `add` è la versione curried di `sum`.
Le definizioni sopra possono ovviamente essere estese al caso di funzioni che prendono come argomento una n-upla.
Nell'uso matematico non siamo molto abituati alle funzioni curried: vengono utilizzate, ma in genere gli argomenti sono scritti sotto forma di indici. Si consideri per esempio $\log_a b$: se fissiamo il primo argomento (la base), per esempio a dieci, otteniamo la funzione $\log_{10} b$.
Vediamo altri esempi:

- la versione curried della composizione funzionale:

  ```
  # let compose f g x = f (g x );;
  val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
  ```

- la funzione che costruisce la coppia formata da due elementi:

  ```
  # let pair x y = (x,y);;
  val pair : 'a -> 'b -> 'a * 'b = <fun>
  ```

- la funzione che controlla se un valore è minore di un altro (il simbolo < è predefinito in Caml e denota un operatore "minore di" applicabile a valori di diversi tipi):

```
# let lessthan x y = y < x;;
val lessthan : 'a -> 'a -> bool = <fun>
# lessthan 3 2;;
- : bool = true
```

(ossia, il predicato “essere minore di tre” vale sul due)

Si noti che le trasformazioni da una funzione non curried nella sua versione curried e viceversa viste sopra possono essere scritte in Caml come funzioni higher-order:

```
# let curry f = function a -> function b -> f(a,b);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# curry sum;;
- : int -> int -> int = <fun>
# let uncurry g = function (a,b) -> g a b;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
# uncurry add;;
- : int * int -> int = <fun>
```

Esercizio: dire se le seguenti espressioni sono tipabili, e se sì trovare i tipi (consideriamo da qui in poi `compose` nella versione curried).

```
uncurry curry sum
uncurry (curry sum)
uncurry compose
compose curry uncurry
compose uncurry curry
```

Altri costrutti sintattici In Caml è possibile definire funzioni ricorsive nel modo usuale, utilizzando la sintassi `let rec ...` oppure `let rec ... and ... and ...` nel caso di mutua ricorsione. Per esempio, la seguente è la definizione di una funzione `stringcopy` che, dati un intero n e una stringa s, restituisce la stringa ottenuta concatenando n copie di s. Possiamo scivere questa *specifica* come commento inseribile nel codice Caml, come mostrato sotto.

```
# (* stringcopy n s = s ... s (n volte) *)
let rec stringcopy n s = if n <= 0 then "" else s^stringcopy (n-1) s;;
val stringcopy : int -> string -> string = <fun>
```

Le definizioni che abbiamo visto finora sono *top level*, cioè il loro scope è la sessione di lavoro corrente. È anche possibile dare definizioni *locali* utilizzando la sintassi `let ... in ...`.
L'operatore condizionale ha la sintassi `if ... then ... else ...`. Per esempio:

```
# let min(x,y) = if x<y then x else y;;
val min : 'a * 'a -> 'a = <fun>
```

Le funzioni di più argomenti vengono viste come funzioni che prendono come argomento una n-upla (cioè un'espressione che ha un tipo prodotto). Per esempio:

```
# let evenprod(x,y) = even (x*y);;
val evenprod : int * int -> bool = <fun>
# let eithereven (x,y) = even x or even y;;
val eithereven : int * int -> bool = <fun>
# let fst (x,y) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd(x,y) = y;;
val snd : 'a * 'b -> 'b = <fun>
```

Si noti che `evenprod` e `eithereven` definiscono la stessa funzione; si dice che sono due diverse descrizioni *intensionali* che corrispondono alla stessa descrizione *estensionale* (cioè, lo stesso insieme di coppie).
Sui tipi base sono definite le operazioni usuali, per esempio: `+` e `*` su `int`, `+.` e `*.` su `float`, `&` e `or` (come al solito *non stretti*, che cioè valutano il secondo argomento solo se necessario) su `bool`.
Per utilizzare un operatore come espressione di tipo funzionale basta metterlo tra parentesi:

```
# (&);;
- : bool -> bool -> bool = <fun>
```

Gli operatori di confronto, come = e <, sono definiti su tutti i tipi; tuttavia se vengono invocati su argomenti di tipo funzionale danno luogo a un'eccezione[24]. Sui tipi prodotto e user-defined sono definiti componente per componente.

```
# (=);;
- : 'a -> 'a -> bool = <fun>
# let f x = x;;
val f : 'a -> 'a = <fun>
# let g x = x ;;
val g : 'a -> 'a = <fun>
# f = g;;
Exception: Invalid_argument "equal: functional value".
```

Un costrutto sintattico di Caml molto utile è il *pattern-matching*. In particolare, è possibile dare definizioni di funzioni "per casi" (separati dal simbolo `|`), dove ogni caso è specificato da un *pattern*, cioè un'espressione con variabili che descrive una possibile forma dell'argomento della funzione. Ossia, non tutti i valori sono istanziazioni del pattern, che agisce quindi come un filtro (a meno che non sia un pattern banale come `x`). Per esempio:

```
# let negate = function true -> false
  | false -> true;;
val negate : bool -> bool = <fun>
```

oppure

```
# let negate = function true -> false
  | _ -> true;;
val negate : bool -> bool = <fun>
```

Il simbolo _ (underscore) può essere utilizzato al posto di una variabile se non ci sono riferimenti a essa.
I casi vengono esaminati nell'ordine. Una definizione per pattern-matching non esaustiva genera un warning del compilatore:

```
#  let negate = function true -> false;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
false
val negate : bool -> bool = <fun>
```

Se si invoca una funzione su un argomento per il quale non è previsto un pattern, viene sollevata un'eccezione.

```
# negate false;;
Exception: Match_failure ("", 14, 36).
```

Vediamo un altro esempio: l'implicazione logica. Il modo più banale di definirla è utilizzando la sua tabella di verità.

```
let imply = function (true,true) -> true
  | (true, false) -> false
  | (false, true) -> true
  | (false, false) -> true;;
```

Possiamo però anche dare la seguente definizione più compatta:

```
let imply = function (true,x) -> x
  | (false, _) -> true
```

o anche

```
let imply = function (true,false) -> false
  | _ -> true
```

È vietato ripetere una variabile, per esempio:

```
# let f = function (x,x) -> true;;
This variable is bound several times in this matching
```

[24]Dato che per controllare che due funzioni sono uguali occorrerebbe controllare che diano lo stesso risultato su tutti gli (infiniti) argomenti. In altri linguaggi della famiglia ML si distingue invece tra tipi con uguaglianza e tipi senza uguaglianza.

3.2 Simulazione di algoritmi imperativi

È possibile simulare le strutture di controllo imperative modellando i comandi come funzioni da stato a stato. Illustriamo quest'idea su un esempio concreto. Si consideri il seguente comando espresso in una sintassi tipo C e Java:

```
z = x;
while (x <= y) {
  z = z * x;
  x++;
}
```

In questo caso, lo stato è costituito dalle tre variabili intere x, y e z. Possiamo quindi vedere un comando come una funzione

$$\texttt{int} * \texttt{int} * \texttt{int} \rightarrow \texttt{int} * \texttt{int} * \texttt{int}$$

che, dati tre valori iniziali per queste variabili, restituisce i loro valori finali. Abbreviamo int ∗ int ∗ int con state.
I comandi base utilizzati nel comando composto dato sopra possono quindi essere modellati con le seguenti funzioni di tipo state → state:

```
assignXtoZ (x,y,z) = (x,y,x)
multZbyX (x,y,z) = (x,y,z*x)
incrX (x,y,z) = (x+1,y,z)
```

Il test nel ciclo while sarà invece modellato con la seguente funzione di tipo state → bool:

```
testXleqY (x,y,z) = x <= y
```

Vediamo ora come simulare le strutture di controllo utilizzate nel comando composto dato sopra, cioè la sequenza di comandi e il ciclo while. La sequenza di comandi, dato che ogni comando è una funzione da stato a stato, corrisponde semplicemente alla composizione di funzioni, che in questo caso assume il tipo

$$(\texttt{state} \rightarrow \texttt{state}) \rightarrow (\texttt{state} \rightarrow \texttt{state}) \rightarrow (\texttt{state} \rightarrow \texttt{state})$$

Il ciclo while, che viene costruito a partire da un test e da un comando (corpo), può essere modellato con una funzione di tipo

$$(\texttt{state} \rightarrow \texttt{bool}) \rightarrow (\texttt{state} \rightarrow \texttt{state}) \rightarrow (\texttt{state} \rightarrow \texttt{state})$$

definita nel modo seguente:

```
let rec while_loop test body s =
  if test s then while_loop test body (body s )
  else s
```

Mettendo insieme le varie definizioni, il comando composto dato sopra può essere modellato come segue:

```
let com = compose (while_loop testXleqY (compose incrX multZbyX)) assignXtoZ;;
```

Per esempio, valutare la seguente espressione corrisponde a eseguire il comando in uno stato iniziale in cui x vale 1, y vale 5 e z vale zero:

```
# com(1,5,0);;
- : int * int * int = (6, 5, 120)
```

Per esercizio, si definisca una funzione higher-order corrispondente al ciclo for.
Una tecnica per simulare algoritmi iterativi con definizioni ricorsive è inoltre quella dei *parametri di accumulazione*, vedi Sez.3.4.

3.3 Liste

In Caml è possibile scrivere espressioni che denotano liste, con la seguente sintassi:

- `[]` denota la lista vuota (di tipo α `list`, cioè il tipo delle liste con elementi di un tipo arbitrario)
- `e::l`, dove `e` è un'espressione di un certo tipo `T` e `l` è un'espressione di tipo `T list`, denota la lista (di tipo `T list`) ottenuta concatenando l'elemento `e` in testa alla lista `l`.

Inoltre, `[e_1;...; e_n]` è un'abbreviazione per `e_1:: e_2 :: ...:: e_n::[]`. Notate che il separatore è un ";", l'espressione `[1,2]` è corretta *ma* rappresenta una lista di un elemento: la coppia (1,2).
Gli operatori `[]` e `_::_` sono polimorfi, con tipo α `list` e $\alpha \rightarrow \alpha$ `list` $\rightarrow \alpha$ `list`, rispettivamente. Possiamo quindi costruire liste di qualunque tipo, ma le liste sono omogenee, cioè tutti gli elementi devono avere lo stesso tipo, come illustrato dai seguenti esempi.

```
# [3;4;5];;
- : int list = [3; 4; 5]
# [true;false;true];;
- : bool list = [true; false; true]
# [add 1; add 2; add 3];;
- : (int -> int) list = [<fun>; <fun>; <fun>]
# [3;true];;
This expression has type bool but is here used with type int
```

Si noti che, a differenza di quanto avviene nel paradigma imperativo, non essendovi alcuna nozione di stato non vi è alcuna necessità per l'utente di gestire allocazione/deallocazione di memoria; questi aspetti sono gestiti automaticamente dal sistema e rimangono nascosti.
Vediamo ora alcune funzioni su liste.
La funzione che controlla se una lista è vuota:

```
# let is_empty l = (l=[]);;
val is_empty : 'a list -> bool = <fun>
```

oppure usando il pattern-matching

```
let is_empty = function [ ]->true
 | x::l -> false
```

oppure anche

```
let is_empty = function [] -> true
  | _ -> false
```

Le funzioni che restituiscono il primo elemento (testa) di una lista e la lista privata del primo elemento (coda):

```
# let head = function [] -> raise(Failure "head") | x::_ -> x;;
val head : 'a list -> 'a = <fun>
# let tail = function [] -> raise(Failure "tail") | _::l -> l;;
val tail : 'a list -> 'a list = <fun>
```

Queste due funzioni sollevano un'eccezione se invocate sulla lista vuota. Alternativamente, è possibile dare un pattern-matching non esaustivo.
La funzione che restituisce la lunghezza di una lista:

```
# let rec length = function [] -> 0
  | x::l -> 1 + length l;;
val length : 'a list -> int = <fun>
```

La funzione che concatena due liste:

```
# let rec append = function [],l -> l
  | x::l, l1 -> x::append (l,l1);;
val append : 'a list * 'a list -> 'a list = <fun>
```

Questa funzione in Caml è predefinita e si indica con l'operatore infisso @.
La funzione che data una lista restituisce la lista rovesciata:

```
# let rec reverse = function []-> []
  | x::l -> reverse l @ [x];;
```

La funzione che data una lista di interi restituisce la somma degli elementi:

```
# let rec sumlist = function []-> 0
  | x::l -> x + sumlist l;;
val sumlist : int list -> int = <fun>
```

Per esercizio dare il tipo e la definizione delle seguenti funzioni su liste:

- `upto n, m` = la lista degli interi da `n` a `m`
- `prodlist l` = il prodotto degli elementi di `l`
- `doublelist l` = la lista dei doppi degli elementi di `l`
- `copy n e` = la lista formata da `n` copie di `e`
- `isin l e` = controlla se `e` appartiene a `l`
- `flatten [l_1; ...; l_n] = l_1 @ ... @ l_n`
- `drop n l` = elimina i primi `n` elementi di `l`

Come illustrato dagli esempi precedenti, nel definire funzioni su liste è molto comune utilizzare il seguente pattern matching:

```
let rec f ... = function [] -> ...
  | x::l -> ...
```

Un esempio dove è necessario utilizzare un pattern matching più complesso è il seguente (funzione che data una lista restituisce la sottolista formata solo dagli elementi di posto pari):

```
# let rec alternate = function a::b::l-> b:: alternate l
  | _ -> [];;
```

3.4 Parametri di accumulazione

Un modo di simulare algoritmi iterativi nel paradigma funzionale è quello di utilizzare *parametri di accumulazione*. Un parametro di accumulazione è un parametro aggiuntivo passato a una funzione ricorsiva che simula lo "stato" (una o più variabili) che verrebbe utilizzato nel corrispondente algoritmo iterativo. In altre parole, un parametro di accumulazione memorizza la "porzione di risultato" ottenuta in una fase intermedia della computazione. In genere, la definizione finale della funzione che ci interessa chiamerà una funzione ausiliaria ricorsiva passandole un opportuno valore iniziale per il parametro di accumulazione.
Vediamo alcuni esempi di definizioni alternative, che fanno uso di parametri di accumulazione, di funzioni viste precedentemente.
La funzione `stringcopy` può essere definita utilizzando un parametro di accumulazione che corrisponde alla stringa ottenuta in una fase intermedia.

```
# let rec auxstringcopy res n s = if n <= 0 then res else auxstringcopy (s^res) (n-1) s;;
val auxstringcopy : string -> int -> string -> string = <fun>
# let stringcopy = auxstringcopy "";;
val stringcopy : int -> string -> string = <fun>
```

(è anche possibile nascondere la definizione ausiliaria dentro quella principale usando il costrutto `let`).
La funzione `sumlist` può essere definita utilizzando un parametro di accumulazione che corrisponde alla somma ottenuta in una fase intermedia.

```
# let rec auxsumlist sum = function []-> sum
  | x::l -> auxsumlist (x+sum) l
val auxsumlist : int -> int list -> int = <fun>
# let sumlist = auxsumlist 0;;
val sumlist : int list -> int = <fun>
```

La funzione `reverse` può essere definita utilizzando un parametro di accumulazione che corrisponde alla porzione di lista già rovesciata in una fase intermedia.

```
# let rec auxreverse rev = function [] -> rev
 | x::l -> auxreverse (x::rev) l;;
val auxreverse : 'a list -> 'a list -> 'a list = <fun>
# let reverse = auxreverse [];;
val reverse : '_a list -> '_a list = <fun>
```

L'utilizzo di parametri di accumulazione è vantaggioso in termini di efficienza. Infatti, confrontando le definizioni ricorsive date sopra con le versioni precedenti, si può osservare che nelle definizioni con parametri di accumulazione i risultati delle chiamate ricorsive non vengono ulteriormente elaborati, ma utilizzati direttamente come risultato della funzione (questa situazione si chiama *tail recursion*). Questo fa sì che, a livello di implementazione, non sia necessario memorizzare tutti i contesti di esecuzione lasciati in sospeso. Si può comprendere tale differenza confrontando le sequenze di esecuzione che si ottengono, per esempio, per la chiamata, `reverse[1;2;3]` applicando le due definizioni.

- definizione senza parametro di accumulazione

  ```
  reverse[1;2;3] ->
  (reverse [2;3]) @ [1] ->
  (reverse [3] @ [2]) @ [1] ->
  ((reverse [] @ [3]) @ [2]) @ [1] ->
  ([3] @ [2]) @ [1] ->
  [3;2]@[1] ->
   [3;2;1]
  ```

- definizione con parametro di accumulazione

  ```
  reverse[1;2;3] ->
  auxreverse [] [1;2;3] ->
  auxreverse [1] [2;3] ->
  auxreverse [2;1] [3] ->
  auxreverse [3;2;1] [] ->
  [3;2;1]
  ```

3.5 Funzioni higher-order su liste

In questa sezione presentiamo alcune funzioni higher-order su liste di particolare utilità.

Funzione map La funzione `map`, date una funzione e una lista di argomenti, restituisce la lista dei risultati, ossia la specifica di `map` è:

```
(* map f [a_1; ...; a_n] = [f a_1; ... ; f a_n] *)
```

La definizione e il tipo sono:

```
# let rec map f = function [] -> []
  | x::l -> f x :: map f l;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Utilizziamo per esempio `map` per produrre, data una lista di interi, la lista ottenuta incrementando di uno ogni elemento:

```
# let mapsucc = map (function x -> x+1);;
val mapsucc : int list -> int list = <fun>
# mapsucc [1;2;3;4];;
- : int list = [2; 3; 4; 5]
```

Il tipo di `map` in questa istanziazione è:

```
(int -> int) -> int list -> int list
```

Per esercizio, si definisca un'istanza di `map` che data una lista di liste restituisce la lista delle lunghezze, e si dica il tipo di `map` in questa istanziazione.

Funzioni che controllano predicati La funzione `filter`, dati un predicato e una lista, restituisce la sottolista formata solo dagli elementi che verificano il predicato.

```
# let rec filter p = function [] -> []
  | x::l -> if p x then x :: filter p l else filter p l;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

La funzione `exists`, dati un predicato e una lista, vale vero se e solo se il predicato vale su almeno un elemento.

```
(*  exists p [a_1; ...; a_n] = esiste i t.c. p a_i *)
# let rec exists p = function []-> false
  | x::l-> p x or exists p l;;
val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

La funzione `forall`, dati un predicato e una lista, vale vero se e solo se il predicato vale su tutti gli elementi.

```
(*  exists p [a_1; ...; a_n] = p a_1 & ... & p a_n *)
# let rec forall p = function []-> true
  | x::l-> p x & forall p l;;
val forall : ('a -> bool) -> 'a list -> bool = <fun>
```

Per esercizio, utilizzare le funzioni higher-order date sopra per definire: una funzione che restituisce tutti gli elementi di una lista di interi maggiori di un numero dato; una funzione che controlla se in una lista di liste non ci sono liste vuote; una funzione che prova ad applicare una lista di funzioni a un numero dato e controlla se tutti i risultati sono positivi. In ognuno dei casi, precisare il tipo della funzione richiesta e il tipo della funzione higher-order nell'istanziazione.

Iteratori su liste La funzione `itlist` (altri nomi `accumulate`, `foldleft`), dati una funzione un valore (intuitivamente, il valore iniziale di un parametro di accumulazione) e una lista, applica iterativamente la funzione al parametro di accumulazione e all'elemento corrente della lista, utilizzando ogni volta il risultato come valore successivo del parametro di accumulazione. Il risultato è il valore finale del parametro di accumulazione.
Ossia:

```
(* itlist f a [b_1; ...; b_n] = f (... (f (f a b_1) b_2) ...) b_n *)
```

In altri termini, la funzione `itlist` simula il seguente algoritmo iterativo:

```
A = a;
while (l != [ ]) {
  A = f A head(l);
  l = tail(l);
}
return A;
```

La definizione e il tipo sono:

```
# let rec itlist f a = function [] -> a
  | b :: l -> itlist f (f a b) l;;
val itlist : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Vediamo alcuni esempi di funzioni che abbiamo già definito in modo diretto, date invece come istanze di `itlist` (si noti che la definizione come istanza di `itlist` equivale a una definizione con parametro di accumulazione):

```
# let sumlist = itlist (+) 0;;
val sumlist : int list -> int = <fun>
# let prodlist = itlist (*) 1;;
val prodlist : int list -> int = <fun>
# let flatten = itlist (@) [];;
val flatten : '_a list list -> '_a list = <fun>
# let reverse = itlist (function rev -> function a -> a::rev) [];;
val reverse : '_a list -> '_a list = <fun>
```

Per esercizio, si dia il tipo di `itlist` in ognuna di queste istanze.

Esercizio 3

1. Si diano tipo e definizione della funzione `combine` con la seguente specifica:

   ```
   combine ([x_1; ...; x_n],[y_1;...;y_n]) = [(x_1,y_1);...; (x_n,y_n)]
   ```

2. Si diano tipo e definizione di una funzione `sublists` che restituisce la lista di tutte le sottoliste di una lista data (variante: solo le sottoliste formate da elementi contigui). Per esempio data la lista `[1;2;3;]` devo ottenere:
   `[[];[1];[2];[3];[1;2];[1;3];[2;3];[1;2;3]]`
   nella variante devo ottenere:
   `[[];[1];[2];[3];[1;2];[2;3];[1;2;3]].`

Vediamo ora alcuni ulteriori esempi che mostrano come definire in Caml classici algoritmi di ordinamento. Si noti che la definizione è data in modo parametrico rispetto a una generica relazione di ordinamento `leq`.

Esempio 3.1 [Insertsort]

```
# let rec insert leq a = function
  [] -> [a]
  | b :: l -> if leq b a then a :: b :: l else b :: insert leq a l;;
val insert : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>
#  let insertsort leq = itlist (function l -> function a -> insert leq a l) [];;
val insertsort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

Si noti che `leq b` è il predicato che vale vero se `a` è nella relazione `leq` con `b`.

Esempio 3.2 [Quicksort]

```
# let rec quicksort leq  = function
  []-> []
  | a :: l -> let low = filter (leq a) l
                              and high = filter (function b -> not (leq a b)) l
               in quicksort leq low @ [a] @ quicksort leq high;;
val quicksort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

Per esercizio, si ottimizzi la definizione di `quicksort` data sopra definendo e poi utilizzando una funzione higher-order

```
split: ('a -> bool) -> 'a list -> 'a list * 'a list
```

che, dati un predicato `p` e una lista, restituisce due liste, quella degli elementi su cui vale `p` e quella degli elementi su cui non vale.
Definiamo ora un altro iteratore su liste `listit` (altri nomi usati `reduce`, `foldright`) analogo a `itlist` ma che esegue l'iterazione in ordine inverso, cioè a partire dall'ultimo elemento. Il tipo è lo stesso di `itlist`.
Ossia:

```
(* listit f a [b_1; ...; b_n] = f b_1 (... (f b_n-1 (f b_n a)) ...) *)
```

```
# let rec listit f a = function [] -> a
  | b :: l -> f b (listit f a l);;
val listit : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

Possiamo per esempio utilizzare questo iteratore per definire una funzione che calcola la composizione di una lista di funzioni, cioè la cui specifica è `composelist [f_1; ...; f_n] = f_n ∘...∘ f_1`.

```
# let composelist = listit compose (function x -> x);;
val composelist : ('_a -> '_a) list -> '_a -> '_a = <fun>
```

Altri esempi di uso di questo iteratore:

```
# let append l1 l2 = listit (function x -> function l -> x::l) l2 l1;;
val append : 'a list -> 'a list -> 'a list = <fun>

# let insertsort leq = listit (insert leq) [];;
val insertsort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>

# let map f = listit (compose (function x-> function l-> x::l) f) [];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Le funzioni `itlist` e `listit` danno lo stesso risultato quando sono applicate a unoperazione associativa e al suo elemento identità; ma lefficienza (sia nel tempo sia nello spazio) può variare sensibilmente. Per esempio abbiamo visto prima che la funzione `reverse` definita usando un parametro di accumulazione (l'algoritmo è lo stesso che si ottiene usando `itlist`) è in "tempo lineare", mentre

```
let reverse = listit (function a -> function rev -> rev @ [a]) []
```

in "tempo quadratico"; infatti nella definizione di `listit` *non* si ha tail recursion.

3.6 Tipi user-defined

In Caml l'utente può definire nuovi tipi, in particolare tipi record e tipi somma (variant). Un nuovo tipo è introdotto dalla parola chiave `type`.
Un esempio di tipo prodotto è il seguente:

```
type person = {name: string; age : int ; job : string}
```

Avendo dichiarato questo tipo, è possibile scrivere espressioni come nell'esempio seguente:

```
# let mario = {name = "Mario"; job = "Studente"; age = 21};;
val mario : person = {name = "Mario"; age = 21; job = "Studente"}
# let age_of = function {name = _; job =_; age = x} -> x;;
val age_of : person -> int = <fun>
# mario.age;;
- : int = 21
```

È possibile definire tipi record parametrici:

```
type ('a, 'b) pair = { fst : 'a ; snd : 'b};;
```

Vediamo ora degli esempi di tipi somma.

```
type suit = Heart | Diamond | Club | Spade
type card = Ace of suit | King of suit | Queen of suit | Jack of suit | Plain of suit * int
```

Avendo dichiarato questo tipo, è possibile scrivere espressioni come nell'esempio seguente:

```
# let picture_cards_of s = [King s; Queen s; Jack s];;
val picture_cards_of : suit -> card list = <fun>
#  let number_cards_of s = map (function n -> Plain (s,n)) [1;2;3;4;5;6;7;8;9;10];;
val number_cards_of : suit -> card list = <fun>
```

Il tipo predefinito `bool` corrisponde alla seguente definizione:

```
type bool = false | true
```

Come sempre, è buona pratica di programmazione introdurre un nuovo tipo anche quando c'è un tipo predefinito che ne fornisce un'implementazione, per esempio:

```
type age = Age of int;;
type height = Height of int;;
```

altrimenti potrei sommare età e lunghezze.
È possibile definire tipi ricorsivi (tipicamente per implementare strutture dati induttive), come nel seguente esempio:

```
type exp = Num of int | Id of string | Sum of exp*exp | Prod of exp*exp
```

È possibile anche nel caso dei tipi somma definire tipi parametrici; per esempio la seguente dichiarazione definisce il tipo degli alberi binari con etichette di un tipo arbitrario.

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

Il tipo predefinito delle liste corrisponde alla definizione

```
type 'a list = [] | _::_ of 'a * 'a list
```

Vediamo qualche esempio di funzione su alberi binari. La funzione `insertBST` inserisce un elemento in un Binary Search Tree (BST); la funzione `constrBST` costruisce un BST a partire da una lista di elementi.

```
# let rec insertBST a = function
  Empty -> Node(a,Empty,Empty)
  | Node (b,left,right) -> if a < b then Node (b, insertBST a left, right)
                                    else if b < a then Node (b, left, insertBST a right)
                                    else Node(b,left,right);;
val insertBST : 'a -> 'a btree -> 'a btree = <fun>
# let constrBST = itlist (function bst -> function a -> insertBST a bst) Empty;;
val constrBST : '_a list -> '_a btree = <fun>
# let t = constrBST [1;3;4;1;7;9;0;2];;
val t : int btree =
  Node (1, Node (0, Empty, Empty),
   Node (3, Node (2, Empty, Empty),
    Node (4, Empty, Node (7, Empty, Node (9, Empty, Empty)))))
```

La funzione `inorder` restitusce la lista delle etichette di un albero binario effettuando una visita inorder (in modo analogo si possono definire preorder e postorder).

```
# let rec inorder = function
  Empty -> []
  | Node(b,left,right) -> inorder left @[b] @ inorder right;;
# inorder t;;
- : int list = [0; 1; 2; 3; 4; 7; 9]
```

La funzione `btree_in` astrae rispetto alla precedente, cioè definisce uno schema di visita inorder in cui, a partire da un certo valore iniziale `a`, per ogni nodo si effettua una certa azione `f`.

```
# let rec btree_in f a = function
  Empty -> a
  | Node(b,left,right) -> btree_in f (f (btree_in f a left) b) right;;
val btree_in : ('a -> 'b -> 'a) -> 'a -> 'b btree -> 'a = <fun>
```

Avendo a disposizione questa funzione, la funzione `inorder` può essere definita nel modo seguente:

```
let inorder = btree_in (function l -> function b -> l@[b]) []
```

3.7 Tipi Caml e inferenza di tipo

Riassumiamo la sintassi dei tipi Caml illustrati precedentemente:

- tipi base: `int`, `float`, `bool`, `char`, `string`
- variabili di tipo: α, β, γ, ...
- tipi funzionali, della forma `t -> t'` se `t`, `t'` sono due tipi
- tipi prodotto, della forma `t * t'` se `t`, `t'` sono due tipi (dato che il prodotto di tipi è associativo non è ambigua la notazione `t_1 * ...* t_n`)
- tipi lista, della forma `t list` se `t` è un tipo
- tipi user-defined record e somma

Come già detto, non daremo qui l'algoritmo di type inference, ma illustreremo come dedurre il tipo più generale di un'espressione (o concludere che questa non è tipabile) su alcuni esempi. Per cercare di assegnare un tipo a un'espressione Caml, utilizzeremo delle (ovvie) regole di tipo informali; le più importanti sono date sotto:

- un'applicazione `e e'` ha tipo `t''` se `e` ha un tipo funzionale `t' -> t''` ed `e'` ha tipo `t'`
- un'espressione di tipo funzionale `function x -> e` ha tipo `t -> t'` se `e` ha tipo `t'` assumendo che `x` abbia tipo `t`

Consideriamo ora i seguenti esempi:

```
let pippo f = function x -> (x f , f 1)
```

```
let pippo f = function x -> [x 1 ; f x]
```

e vediamo come dedurre il tipo più generale. Nel primo esempio, `pippo` prende come argomento `f` e restituisce una funzione che prende a sua volta come argomento `x` e restituisce la coppia `(x f , f 1)`. Quindi il tipo di `pippo` sarà della forma

```
tipo di f -> tipo di x -> (tipo di x f ) * (tipo di f 1)
```

Osserviamo ora che, poiché `f` viene applicata a `1`, il tipo di `f` deve essere della forma $\texttt{int} \to \alpha$. Inoltre, dato che `x` viene applicata a `f`, il tipo di `x` deve essere un tipo funzionale che prende come argomento il tipo di `f`, quindi della forma $(\texttt{int} \to \alpha) \to \beta$. Mettendo insieme queste osservazioni ricaviamo il tipo più generale di `pippo`, cioè:

$$(\texttt{int} \to \alpha) \to ((\texttt{int} \to \alpha) \to \beta) \to \beta * \alpha$$

Nel secondo esempio, `pippo` prende come argomento `f` e restituisce una funzione che prende a sua volta come argomento `x` e restituisce la lista `[x 1 ; f x]`. Quindi il tipo di `pippo` sarà della forma

```
tipo di f -> tipo di x -> t list
```

dove `t` è il tipo di `x 1` e `f x`. Osserviamo ora che, poiché `x` viene applicata a `1`, il tipo di `x` deve essere della forma $\texttt{int} \to \alpha$. Inoltre, dato che `f` viene applicata a `x`, il tipo di `f` deve essere un tipo funzionale che prende come argomento il tipo di `x`, quindi della forma $(\texttt{int} \to \alpha) \to \beta$. Infine, poiché il tipo di `x 1` e `f x` deve essere lo stesso, si ha che α e β devono coincidere. Mettendo insieme queste osservazioni ricaviamo il tipo più generale di `pippo`, cioè:

$$((\texttt{int} \to \alpha) \to \alpha) \to (\texttt{int} \to \alpha) \to \alpha\ \texttt{list}$$

3.8 Caratteristiche non trattate

Il linguaggio Caml fornisce anche un meccanismo di eccezioni simile a quello di Java ma senza il controllo, e dei costrutti imperativi. Inoltre, la versione Objective Caml ha anche costrutti object-oriented e un sistema di moduli.

Riferimenti bibliografici

[1] K. Arnold and J. Gosling. *The Java™ Programming Language, Fourth Edition*. Addison-Wesley, 2005. Terza edizione e prima edizione in italiano disponibili in biblioteca.

[2] W.R. Cook. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science 489, pag.151–178. Springer, 1990.

[3] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 1997. Prima edizione in italiano disponibile in biblioteca.

[4] J. Gosling, B. Joy, G. L. Steele, G. Bracha. *The Java language specification, Third Edition*. Addison Wesley, 2005. Disponibile in biblioteca.

[5] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1996. Seconda edizione, disponibile in biblioteca.

[6] P. Wegner. Dimensions of object based language design. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1987*, pag.168–182, 1987.

A Appendice Tecnica

A.1 Nozioni Base

Indicheremo con $\mathbb{N}$, $\mathbb{Z}$ e $\mathbb{B}$ rispettivamente gli insiemi dei numeri naturali, dei numeri interi e dei valori booleani *T* e *F*.
Useremo la notazione $[a, b]$ per indicare l'insieme dei numeri naturali da a a b cioè $\{k \mid a \leq k \leq b, k \in \mathbb{N}\}$. Ad esempio $[1, 4] = \{1, 2, 3, 4\}$. Si noti che in particolare se $a > b$, allora $[a, b] = \emptyset$.

Def. A.1 [Relazioni e funzioni parziali] Dati due insiemi A e B, una *relazione* R da A in B è un sottoinsieme di $A \times B$; una *relazione su* A è una relazione da A in A. Se $(a, b) \in R$ si scrive anche $a\ R\ b$. Una *funzione (parziale)* da A in B è una relazione f da A in B che gode della proprietà di univocità, cioè per ogni $a \in A$ esiste al più un $b \in B$ tale che $(a, b) \in R$; tale b, se esiste, è denotato con $f(a)$. Se f è una funzione da A in B si scrive $f\colon A \to B$; $A \to B$ denota l'insieme delle funzioni da A in B. Una funzione f da A in B si dice *totale* se $f(a)$ esiste per ogni $a \in A$.

Quando non specificato esplicitamente, per "funzione" intendiamo sempre funzione parziale (quindi non necessariamente totale). Si noti che quando si considera un uguaglianza $e_1 = e_2$ tra espressioni in cui compaiono funzioni parziali, non è scontato quale sia il significato dell'uguaglianza nel caso in cui e_1, e_2 o entrambe sono non definite. Noi assumiamo di interpretare tale uguaglianza nel senso cosiddetto *forte*, cioè $e_1 = e_2$ vale se e_1 ed e_2 sono entrambe definite e uguali, oppure sono entrambe indefinite.

Def. A.2 [Sostituzione] Data una funzione $f\colon A \to B$, due elementi $a \in A$, $b \in B$, $f[b/a]$ indica la funzione da A in B così definita:

$f[b/a](a) = b$, $f[b/a](a') = f(a')$ per $a' \neq a$.

Abbrevieremo $f[a_1/b_1] \ldots [a_n/b_n]$ con $f[a_1/b_1 \ldots a_n/b_n]$ e $\emptyset[a_1/b_1 \ldots a_n/b_n]$ con $[a_1/b_1 \ldots a_n/b_n]$.

Def. A.3 [Chiusura transitiva] Se R è una relazione su A, la *chiusura transitiva* di R è la relazione R^+ su A definita come segue:

per ogni $a, a' \in A$, aR^+a' sse $a = a_0Ra_1, a_1Ra_2, \ldots, a_{n-1}Ra_n = a'$, per qualche $a_1, \ldots, a_n$, $n \geq 1$.

La *chiusura riflessiva e transitiva* di R è la relazione $R^\star$ su A definita da: $R^\star = R^+ \cup \{(a, a) \mid a \in A\}$.
Equivalentemente, R^+ può essere definita induttivamente come segue:

- se aRa', allora aR^+a';
- se aR^+a' e $a'R^+a''$, allora aR^+a'';

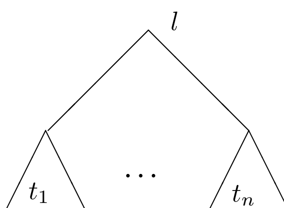
ed $R^\star$ può essere definita induttivamene dalle due metaregole precedenti più la seguente:

- $aR^\star a$.

Sia nel seguito L un insieme di *etichette*.

Def. A.4 [Albero etichettato] L'insieme degli *alberi (ordinati) etichettati in* L è definito induttivamente da:

se $t_1, \ldots, t_n$ sono alberi con etichette in L, $n \geq 0$, $l \in L$, allora

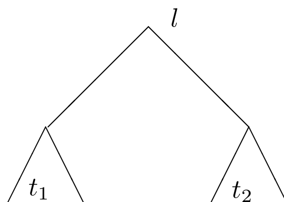


è un albero con etichette in L.

Si noti che nella definizione induttiva sopra la base è data dal caso $n = 0$ (albero formato da un unico nodo). Si noti anche che gli alberi che abbiamo definito sono *ordinati* (cioè i figli di un nodo sono considerati una sequenza e non un insieme).

Def. A.5 [Albero binario etichettato] L'insieme degli *alberi binari etichettati in* L è definito induttivamente da:

- l'albero vuoto è un albero binario con etichette in L;
- se t_1, t_2 sono alberi binari con etichette in L, $l \in L$, allora



è un albero binario con etichette in L.

Sia nel seguito S un insieme i cui elementi sono detti *indici* (o *sort* o *tipi*).

Def. A.6 [Famiglia di insiemi] Una *famiglia di insiemi* (o semplicemente *famiglia*) indiciata su S (o S-famiglia) è una funzione totale che associa a ogni $s \in S$ un insieme. Se A è una S-famiglia di insiemi, per ogni $s \in S$ l'insieme associato a s si indica A_s, e A si scrive anche $\{A\}_{s \in S}$.

Si noti la differenza tra una famiglia e un insieme; ad esempio la famiglia A indiciata su a, b, c definita da $A_a = \mathbb{Z}$, $A_b = \mathbb{B}$, $A_c = \mathbb{Z}$ è diversa dall'insieme $\{A_a, A_b, A_c\} = \{\mathbb{Z}, \mathbb{B}\}$.
Le usuali operazioni sugli insiemi (ad esempio unione, intersezione, differenza) si estendono alle famiglie indiciate di insiemi, componente per componente (è necessario che le famiglie coinvolte siano indiciate tutte sullo *stesso* insiemi di indici).
In modo esattamente analogo a una famiglia di insiemi si definisce una famiglia indiciata su S (o S-famiglia) di funzioni $f = \{f_s\}_{s \in S}$.

A.2 Algebre Eterogenee

Def. A.7 [Segnatura] Una *segnatura (eterogenea)* è una coppia (S, O) dove S è un insieme di simboli detti *sort* (o anche *tipi*, *indici*) e O è una $(S^\star \times S)$-famiglia di insiemi i cui elementi sono detti *simboli di operazione*. Per ogni $w \in S^\star, s \in S$, se $op \in O_{(w,s)}$ si scrive $op\colon w \to s$ e si dice che op ha *arità* w e *tipo* s; la coppia (w, s) si chiama *funzionalità* di op.

In informatica si è sempre interessati a segnature eterogenee, cioè in cui si considerano diversi tipi. Si noti che in una segnatura *omogenea* (cioè con un unico tipo) l'arità di un simbolo di operazione diventa semplicemente un numero naturale (il numero degli argomenti).
La nozione di segnatura esprime formalmente l'idea di "sintassi"o "interfaccia" di un tipo di dato, cioè specifica un insieme di nomi di tipo e di operazione senza indicarne una particolare implementazione (questo corrisponderà invece alla nozione di *algebra*, vedi sotto).
Ad esempio, una segnatura per il tipo di dato "liste di interi" potrebbe essere la seguente:

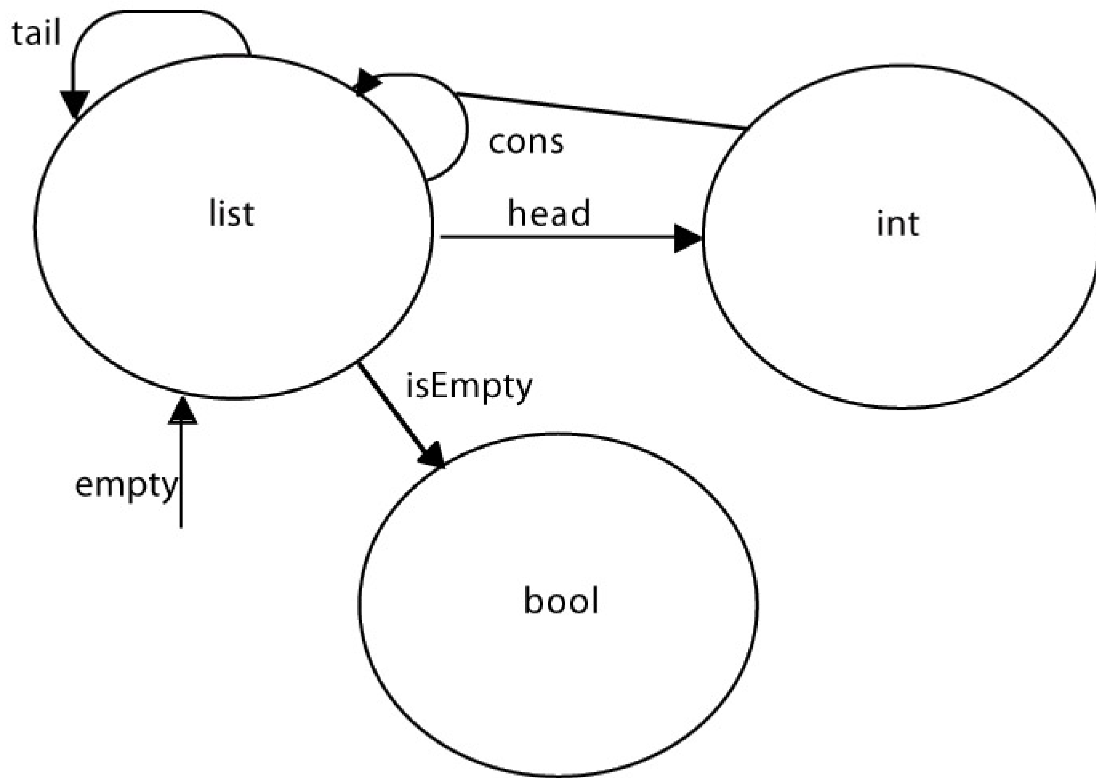


Figura 8: Segnatura del tipo di dato “liste di interi”

Sort int, $bool$, $list$

Simboli di operazione

$empty\colon \ \to list$
$cons\colon int\ list \to list$
$isEmpty\colon list \to bool$
$head\colon list \to int$
$tail\colon list \to list$

Per descrivere una segnatura si usa spesso la notazione grafica illustrata sull’esempio delle liste di interi in Fig.8.

Def. A.8 [Algebra] Sia $\Sigma = (S, O)$ una segnatura. Un’*algebra su* Σ o Σ-*algebra (parziale)* A consiste di

- per ogni $s \in S$, un insieme A_s detto *carrier* (*supporto*) di tipo s;
- per ogni simbolo di operazione $op\colon w \to s$ in O, $w = s_1 \ldots s_n$, una funzione $op^A\colon A_{s_1} \times \ldots \times A_{s_n} \to A_s$ detta l’*interpretazione* di op in A.

La nozione di algebra su una segnatura esprime formalmente l’idea di tipo di dato (o struttura dati) con una certa interfaccia. L’algebra specifica infatti una particolare implementazione di ogni tipo (un insieme di valori) e di ogni operazione (una funzione con argomenti e risultato dei tipi specificati nell’interfaccia).
Un’algebra su (S, O) si dice *totale* se l’interpretazione di ogni simbolo di operazione in O è una funzione totale. Quando non specificato esplicitamente, per “algebra” intenderemo sempre algebra parziale (quindi non necessariamente totale).

Def. A.9 [Omomorfismo] Siano A e B due algebre su una segnatura (S, O). Un *omomorfismo* da A in B è una S-famiglia f di funzioni tale che:

- per ogni $s \in S$, $f_s\colon A_s \to B_s$;

- per ogni $op\colon s_1 \ldots s_n \to s$ in O,

$$f_s(op^A(a_1, \ldots, a_n)) = op^B(f_{s_1}(a_1), \ldots, f_{s_n}(a_n)),$$

  per ogni $a_1 \in A_{s_1}, \ldots a_n \in A_{s_n}$.

La seconda condizione sopra viene anche espressa dicendo che f *rispetta le operazioni*. Ricordiamo che trattandosi di algebre parziali l'uguaglianza va intesa nel senso *forte* (cfr. commento dopo Def.A.1).

Def. A.10 [Isomorfismo di algebre] Siano A e B due algebre su una segnatura (S, O); si dice che A e B sono *isomorfe* se esiste un omomorfismo f da A in B che sia un *isomorfismo*, cioè tale che ogni componente di f sia una funzione bigettiva.