Capitolo 1

1. Nozione di macchina astratta

La macchina, intesa come calcolatore, è una macchina fisica che permette di eseguire algoritmi.
Intuitivamente quindi una macchina astratta non è altro che un'astrazione del concetto di calcolatore.
Per poter essere eseguiti, gli algoritmi devono essere opportunamente formalizzati tramite i costrutti di un linguaggio di programmazione L. L è costituito da una specifica sintassi che è un insieme finito di costrutti, e da una semantica.

Definizione 1: ***Macchina astratta:*** *supponiamo che sia dato un linguaggio di programmazione L. definiamo una macchina astratta per L, e la indichiamo con M_L, un qualsiasi insieme di strutture dati e di algoritmi che permettano di memorizzare ed eseguire scritti in L.*

1. L'interprete

Ovviamente alcune operazioni saranno specifiche a seconda del linguaggio, ma si possono ricavare dei tratti comuni:

1. Operazioni per l'elaborazione dei dati primitivi: con dati primitivi si intende dei dati che siano rappresentabili in modo diretto dalla macchina la quale li può riconoscere ed eseguire delle operazioni (si pensi agli interi e alle operazioni matematiche).
2. Operazioni e strutture dati per il controllo della sequenza di esecuzione delle operazioni: servono per gestire il flusso di esecuzione delle istruzioni presenti in un programma, in quanto la normale esecuzione sequenziale di un programma può dover essere modificata in base a delle condizioni. Per controllare queste condizioni sono necessarie operazioni particolare.
3. Operazioni e strutture dati per il controllo del trasferimento dei dati: servono per controllare come i dati vengono passati dalla memoria all'interprete e viceversa. Tali operazioni riguardano le varie modalità di indirizzamento e l'ordine degli operandi.
4. Operazioni e strutture dati per la gestione della memoria: sono tutte le operazioni relative all'allocazione di memoria per i dati e per i programmi. Se la macchina astratta è vicina a quella hardware allora la gestione della memoria sarà più semplice.
5. Operazioni e strutture dati per la gestione della memoria: sono tutte le operazioni relative all'allocazione di memoria per i dati e per i programmi.
   Se per macchine hardware e macchine fisiche a registri senza multiprogrammazione la gestione della memoria è al quanto facile, non si può dire lo stesso per quanto riguarda le macchine astratte. Infatti molti sono i costrutti che causano direttamente o indirettamente allocazione e deallocazione di memoria.

Il ciclo dell'interprete è il seguente:

1. Acquisizione dalla memoria della prossima istruzione da eseguire
2. Decodifica dell'istruzione per individuare operandi e tipo dell'istruzione
3. Vengono prelevati gli operandi come descritto dall'istruzione precedente.
   - Memorizzazione dell'eventuale risultato.
4. Se l'istruzione eseguita è di arresto si termina l'esecuzione del flusso[1], altrimenti si ricomincia dal punto 1.

Definizione 2: ***Linguaggio macchina:*** *Data una macchina astratta M_L il linguaggio L "compreso" dell'interprete di M_L è detto linguaggio macchina di M_L.*

[1] Non implica la fine del programma. Potrebbe essere la fine di una funzione.

I programmi scritti nel linguaggio macchina sono memorizzati nelle strutture di memoria in modo da essere distinti.
Tra i linguaggi di programmazione si distinguono principalmente due categorie:

1. Linguaggi di basso livello: sono più vicini alla macchina hardware. Sono stati i primi a essere impiegati (già dagli anni 40) per programmare i primi calcolatori. Sono estremamente scomodi poiché bisogna tenere conto delle caratteristiche fisiche della macchina. I linguaggi a basso livello sono detti anche assembly e vengono tradotti in codice macchina da un opportuno programma detto **assemblatore**.
2. Linguaggio di alto livello: presentano dei meccanismi di astrazione che permettono di non considerare le caratteristiche della macchina.

2. Un esempio di macchina astratta: la macchina hardware

Possiamo identificare i componenti di una macchina hardware come alcuni di quelli fondamentali per una macchina astratta:

1. Memoria: la macchina hardware è composta da vari livelli di memoria, i principali sono:
   - *Memoria principale*: è la memoria RAM, sono sequenze lineari di celle o parole.
   - *Memoria secondaria*: è il disco fisso, sono dispositivi magnetici o ottici.
   - *Cache*: registri dedicati alla CPU.

   L'alfabeto della memoria (qualsiasi) è la codifica binaria.
   I dati sono divisi in tipo di dato primitivo (interi, caratteri, ecc). Per la codifica dei caratteri si usa solitamente il codice ASCII o quello UNICODE, ma non ce n'è una universale.
   I tipi di dati primitivi possono essere manipolati da operazioni eseguite direttamente a livello fisico.
2. Il linguaggio della macchina hardware: un linguaggio di basso livello generalmente può eseguire operazioni molto semplici, di solito nella forma:

   CodiceOperativo Operando1 Operando2

   CodiceOperativo è un particolare codice che identifica un'operazione (ad esempio add indica la somma in quasi tutti i linguaggi assembly). Operando1 e Operando2 non sono sempre presenti entrambi. Vengono chiamati registri, e hanno anche loro dei codici a seconda del linguaggio.
   Sebbene l'insieme delle istruzioni dipenda da macchina a macchina, tuttavia si possono identificare due categorie:
   - *CISC* (Complex Instruction Set Computers): presentano molte istruzioni, anche complesse.
   - *RISC* (Reduced Instruction Set Computers): presentano meno istruzioni e più semplici.
3. Interprete: possiamo riconoscere le seguenti componenti:
   - *Operazioni per l'elaborazione dei dati primitivi*: usuali operazioni realizzate dalla ALU (Arithmetic Logic Unit)
   - *Controllo della sequenza di esecuzione*: struttura principale è il Program Counter che contiene l'indirizzo per la prossima istruzione. Le operazioni sono quelle di incremento e decremento che vengono eseguite su questa struttura.
   - *Controllo trasferimento dati*: le strutture sono registri della CPU che interfacciano la memoria principale:
     - MAR: registro indirizzo dei dati
     - MDR: registro dei dati

     Le operazioni sono quelle necessarie alla modifica di questi due registri, e quelle necessarie per accedere e modificare i registri interni della CPU.
   - *Gestione della memoria*: se non si considera una macchina senza multi-programmazione, la gestione della memoria è la più semplice: il programma è

caricato all'inizio dell'esecuzione e vi rimane fino alla sua terminazione. Tuttavia tutte le macchine di oggi usano metodi più sofisticati, come ad esempio dei livelli intermedi, con strutture dati e operazioni particolari.

Spesso poi avviene la multiprogrammazione ovvero nel caso in cui ci sia bisogno di ottimizzare delle prestazioni si sospende l'esecuzione di un programma per permettere alla CPU di portare a termine il lavoro. Questa tecnica è gestita dal sistema operativo e solitamente necessita di hardware particolare per la sua esecuzione.

Ci sono anche macchine che presentano strutture dati diverse dai registri, come ad esempio quelle che si basano sulle pile.

L'interprete è realizzato da un insieme di dispositivi fisici che costituiscono l'unità di controllo e che usando alcune operazioni particolari permettono di eseguire il ciclo fetch-decode-execute.

- *Fetch*: viene recuperata dalla memoria la prossima istruzione contenuta nel PC[2] e memorizzata nel registro istruzione.
- *Decode*: l'istruzione viene decodificata usando opportuni circuiti logici che permettono la corretta esecuzione dell'istruzione e scelta degli operandi
- *Execute*: viene eseguite l'operazione. Un eventuale risultato viene memorizzato dove specificato dall'istruzione.

Quindi si ricomincia il ciclo a meno che l'istruzione non fosse un'istruzione di terminazione.
Si noti che a livello fisico NON è vi è alcuna differenza tra dati e istruzioni. Tale distinzione avviene in base allo stato della CPU nella fase di fetch e decode.

2. Implementazione di un linguaggio

Una macchina astratta M_L è una macchina che permette di eseguire programmi scritti in L, quindi vi è un legame univoco: data una macchina corrisponde un solo linguaggio.
Dato invece un linguaggio, non vi è alcun legame univoco con una macchina, ma ve ne possono essere più di una che permettano di eseguire programmi in quel linguaggio. Queste macchine possono differire per quanto riguarda interprete e strutture dati, ma tutte hanno in comune L.
Implementare un linguaggio significa creare una macchina che abbia L come linguaggio macchina.

1. Realizzazione di una macchina astratta

Sebbene prima o poi le istruzioni debbano essere eseguite a livello fisico, tuttavia ci possono essere dei livelli intermedi a cui viene realizzata la macchina astratta. I livelli fondamentali sono:

1. **Realizzazione in hardware**: è in linea di principio sempre possibile e concettualmente semplice in quanto si tratta di realizzare mediante dispositivi fisici, una macchina tale che il suo linguaggio coincida con il linguaggio che si vuole implementare.

   Il vantaggio principale di questo tipo di macchina è la velocità d'esecuzione. Tuttavia a fronte di questo vantaggio, ci sono molti svantaggi. Primo fra tutti è legato al fatto che gli algoritmi usati con i linguaggi di alto livello sono molto difficili da trasformare in lavoro fisico. Ne segue quindi una fase di progettazione fisica molto complicata. Per seconda cosa una macchina di questo tipo risulterebbe immodificabile una volta completata, e quindi sarebbero precluse anche possibili modifiche al linguaggio.

   Solitamente quindi si tende a costruire una macchina hardware basandosi su un linguaggio di basso livello, o solamente per alcuni linguaggi dedicati. C'è da dire che alcuni linguaggi di alto livello hanno influenzato lo sviluppo hardware di alcune macchine per quanto riguarda la scelta di operazioni e strutture di dati primitive.

[2] Program Counter

2. **Simulazione mediante software**:
   Una soluzione alternativa è quella di scrivere le strutture dati e gli algoritmi per la macchina virtuale M_L con un altro linguaggio già implementato. In questo casi si avrà la massima flessibilità, ma tuttavia si sarà soggetti a prestazioni inferiori rispetto al caso precedente. Inoltre la macchina astratta su cui viene eseguito il linguaggio già implementato dovrà o essere la macchina hardware, oppure avere a sua volta una macchina astratta, e così via finché non si arriva appunto alla macchina hardware.
3. **Simulazione (emulazione) mediante firmware**:
   Questa possibilità è intermedia fra le due precedenti e consiste nella simulazione di strutture dati e algoritmi mediante microprogrammi.
   Concettualmente quindi appare simile alla soluzione mediante software: la macchina astratta viene gestita mediante programmi che sono poi eseguiti da una macchina fisica. La differenza fondamentale sta nei programmi: in questo caso non sono programmi di alto livello, ma microprogrammi. Questi usano uno speciale linguaggio di livello molto basso e invece che nella memoria principale risiedono in un'opportuna memoria di sola lettura per poter essere eseguiti dalla macchina fisica ad alta velocità.
   Come vantaggio è che ha una velocità di esecuzione superiore alla simulazione mediante software, ma tuttavia comporta un costo non del tutto indifferente nella modificazione di un microprogramma in quanto servono dispositivi appositi per modificare le memorie di sola lettura.

2. Implementazione: il caso ideale

Supponiamo di voler implementare un certo linguaggio L e di avere a disposizione una macchina astratta ospite (escludiamo quindi il caso della macchina hardware).
L'implementazione del linguaggio L sulla macchina ospite avviene tramire una traduzione di L nel linguaggio della macchiano ospite. Si possono distinguere due modalità:

1. Implementazione interpretativa pura: avviene tramite una traduzione "implicita", realizzata dalla simulazione dei costrutti della macchina astratta di L con programmi scritti nel linguaggio della macchina ospite. Ossia l'interprete per la macchina astratta di L, M_L, è scritto nel linguaggio ospite. Non avviene quindi una vera e propria traduzione, ma vi è solo una sorta di "decodifica" da parte dell'interprete che fa corrispondere a un'istruzione L un certo insieme di istruzioni del linguaggio ospite. La differenza con una traduzione è il fatto che il codice ottenuto non corrisponde all'uscita dell'interprete, ma viene prima eseguito dallo stesso.
2. Implementazione compilativa pura: avviene mediante una traduzione "esplicita", trasformando programmi di L in programmi del linguaggio della macchina ospite. La traduzione è eseguita da un opportuno programma detto compilatore, che traduce delle espressioni in un linguaggio sorgente (L) in un linguaggio oggetto. In questo caso la fase di traduzione e di esecuzione sono divise. Si noti che spesso il compilatore è scritto proprio nello stesso linguaggio del linguaggio oggetto (ma ciò non è valido per tutti i casi).

Lo svantaggio dell'implementazione puramente interpretativa risiede nella sua scarsa efficienza. Poiché non vi è una fase iniziale di traduzione l'interprete deve tradurre man mano che prosegue nel codice. Inoltre non ha "memoria" nel senso che ogni volta che incontra una certa istruzione che ha decodificato prima, non riprende lo stesso codice riadattandolo, ma ne decodifica di nuovo. Se ciò avviene ad esempio in un ciclo for, per ogni iterazione si dovrà rifare una decodifica ex-novo delle istruzioni in esso contenute.

Come spesso accade gli svantaggi in termini di efficienza sono compensati in termini di flessibilità, infatti decodificando durante l'esecuzione del flusso permette di interagire direttamente con il programma. Altri vantaggi sono:

1. La quantità di memoria impiegata, in quanto il programma viene memorizzato solo nel linguaggio *L*;
2. È più facile costruire un interprete che un compilatore;

I vantaggi e gli svantaggi dell'approccio compilativo sono duali rispetto a quelli precedenti.

3. Implementazione: il caso reale e la macchina intermedia

Solitamente quando si crea un linguaggio non lo si crea mai "puro", ovvero sono presenti degli aspetti sia di un tipo di implementazione che dell'altro. Ad esempio un compilatore simula alcuni costrutti complicati al posto di tradurli, mentre un interprete lavora su una rappresentazione interna dei programmi, che non coincide quasi mai con la rappresentazione esterna, ovvero avviene una sorta di traduzione dal linguaggio *L* a un linguaggio intermedio.
A seconda di quanto il linguaggio intermedio, presente in tutte e due le implementazioni sia spostato verso il livello sorgente o il livello ospite si avranno vari tipi di implementazione:

1. Se coincide con la macchina sorgente, allora si ha un'implementazione interpretativa pura;
3. Se coincide con la macchina ospite, allora si ha un'implementazione compilativa pura;
4. Se non coincide con nessuna delle due, allora si hanno due casi:
    - Se l'interprete della macchina intermedia è sostanzialmente diverso da quello della macchina ospite, allora siamo in presenza di un'implementazione di tipo interpretativo;
    - Se l'interprete della macchina intermedia è sostanzialmente diverso da quello della macchina sorgente, allora siamo in presenza di un'implementazione di tipo compilativo;

Nell'implementazione interpretativa "non pura" non tutti i costrutti vengono simulati, ma bensì alcun di essi vengono tradotti direttamente in quanto hanno direttamente un corrispettivo.
Nell'implementazione compilativa "non pura" può accadere che alcuni costrutti non trovino un proprio corrispettivo, e per questo motivo vengano simulati.
Anche se di solito la macchina intermedia è sempre presente, spesso non è resa esplicita.

3. Gerarchie di macchine astratte

Solitamente in calcolatori micro-programmati non si usa una macchina astratta, ma bensì una gerarchia di esse, in cui ogni macchina astratta ha un proprio linguaggio e un proprio insieme di funzionalità.
Ogni macchina astratta usa le funzionalità della macchina del livello sottostante e ne mette a disposizione di nuove. Le funzionalità messe a disposizione dalla macchina precedente, non sono tutte, ma solo quelle che il relativo linguaggio vuole mettere a disposizione.
Questo permette una certa indipendenza tra i livelli, nel senso che una qualche modifica a un livello non dovrebbe avere ripercussioni sugli altri livelli.

Capitolo 6

1. Nomi

Definizione 1: *Un nome è una sequenza (possibilmente significativa) di caratteri, usata per rappresentare qualche altra cosa e permette di astrarre sia aspetti relativi ai dati, sia relativi al controllo.*

Definizione 2: *Operazioni primitive: sono simboli quali + * che si riferiscono appunto a delle operazioni essenziali quali più e per.*

Osservazione 1. Il nome e l'oggetto da esso denotato NON sono la stessa cosa, infatti un oggetto può essere denotato da più nomi, e un nome può denotare oggetti diversi in momenti diversi dell'esecuzione.

Definizione 3: *Si parla di aliasing quando un oggetto ha più nomi.*

I nomi ci permettono di fare un'astrazione sui registri di memoria. Il computer sa che quando si scrive ad esempio:

```
int x=2;
```

il valore 2 deve essere memorizzato nel registro denotato dal nome x.

Definizione 4: *Si definisce **ambiente** quella parte dell'implementazione che è responsabile delle associazioni tra i nomi e gli oggetti che questi denotano*[3].

2. Oggetti denotabili

Definizione 5: *Si definisce oggetto denotabile un oggetto a cui può essere assegnato un nome, ad esempio:*

1. Oggetti i cui nomi sono definiti dall'utente
2. Oggetti i cui nomi sono definiti dal linguaggio di programmazione

Di seguito viene velocemente illustrato il processo che porta alla formazione degli oggetti denotabili:

1. **Progettazione del linguaggio:** in questa fase sono definite le associazioni fra nomi e costante primitive, tipi primitivi e operazioni primitive del linguaggio.
2. **Scrittura del programma:** il programmatore inizia a scegliere i nomi, è l'inizio della definizione di alcune associazioni che saranno completate più tardi dal compilatore.
3. **Compilazione:** infatti sebbene il programmatore scelga i nomi, è a compile-time, e successivamente a run-time, che si alloca effettivamente lo spazio e quindi si creano i legami. A compile-time ad esempio vengono istanziate le variabili globali.
4. **Esecuzione:** tutte le associazioni che non siano state precedentemente definite devono essere realizzate a tempo d'esecuzione.

Da questa prima schematica procedura (mancano infatti molte fasi) si possono notare principalmente due fasi che vengono denotate con i nomi di **statica** e **dinamica.**

Definizione 6: *Con statico ci si riferisce a tutto quello che avviene prima dell'esecuzione. La gestione della memoria statica avviene attraverso il compilatore*

Definizione 7: *Con dinamico ci si riferisce a tutto quello che avviene durante l'esecuzione. La gestione della memoria dinamica avviene attraverso opportune operazioni eseguite dalla macchina astratta a tempo d'esecuzione.*

Queste sono una prima definizione di quelli che verranno poi chiamati scope statici e dinamici.

[3] È una prima definizione di ambiente, vedremo in seguito la definizione vera e propria.

3. Blocchi

Definizione 8: *Un blocco è una regione testuale del programma identificata da un segnale di inizio e uno di fine, che può contenere dichiarazioni locali a quella regione.*

Si possono distinguere due casi principali:

1. Blocco associato ad una procedura: è il blocco associato alla dichiarazione di una procedura e che testualmente corrisponde al corpo della stessa, esteso con le dichiarazioni relative ai parametri.
2. Blocco in-line (o anonimo): è il blocco che non corrisponde ad una dichiarazione di procedura e che può pertanto comparite in una qualsiasi posizione dove sia richiesto un comando.

Definizione 9: *Si definiscono blocchi annidati, quei blocchi che contengono altri blocchi.*

Non è mai permessa la sovrapposizione di blocchi, ossia aprire un blocco prima di aver chiuso l'ultimo.

Definizione 10: *Regola di visibilità canonica: una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome. In tal caso, nel blocco in cui compare la ridefinizione la nuova dichiarazione nasconde (o maschera) la precedente.*

Si osservi che non c'è visibilità dai blocchi esterni verso quelli interni.

4. Ambiente

Come abbiamo detto non sempre un nome punta allo stesso oggetto per tutta la durata dell'esecuzione. Per questo c'è bisogno di una definizione di ambiente diversa:

Definizione 11: *Si definisce ambiente (referencing enviroment) l'insieme delle associazioni fra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma ed in uno specifico momento dell'esecuzione.*

L'ambiente è una componente della macchina astratta che, per ogni nome introdotto dal programmatore e in ogni punto del programma, permetter di determinare quale sia l'associazione corretta.

Definizione 12: *Si definisce dichiarazione, il costrutto che permette di introdurre un'associazione nell'ambiente.*

Un particolare tipo di dichiarazioni sono quelle implicite: il tipo dell'oggetto denotato viene dedotto dal contesto nel quale il nome è usato per la prima volta.

Definizione 13: *Si definisce alias la situazione per cui più nomi si riferiscono allo stesso oggetto.*

Questa situazione è comune quando si usano ad esempio dei riferimenti.

Tipi di ambiente

L'ambiente cambia solo all'entrata e all'uscita di un blocco.

Considerando quindi l'ambiente di un blocco, ovvero l'ambiente esistente nel momento in cui il blocco è eseguito. Questo ambiente è costituito in primo luogo dalle associazioni relative ai nomi dichiarati localmente al blocco stesso.

I principali 3 tipi sono:

1. Ambiente locale: quello costituito dall'insieme delle associazioni per nomi dichiarati localmente al blocco; si noti che anche i parametri appartengono a questo ambiente nel caso di procedure.
2. Ambiente non locale: quello costituito dalle associazioni relative ai nomi che sono visibili all'interno di un blocco ma che non sono stati dichiarati localmente.

3. Ambiente globale: quello costituito dalle associazioni create all'inizio dell'esecuzione del programma principale.

Si noti che i nomi che sono introdotti nell'ambiente locale possono essere gli stessi presenti nell'ambiente non locale: in tal caso la dichiarazione più interna (locale) nasconde quella più esterna.

Operazioni sull'ambiente

Come abbiamo detto l'ambiente subisce modifiche all'entrata e all'uscita da un blocco. In particolare in entrata si hanno le seguenti modifiche:

1. Create associazioni fra in nomi dichiarati localmente al blocco e gli oggetti;
2. Disattivate associazioni per i nomi già esistenti all'esterno del blocco che vengono però ridefiniti.

All'uscita invece:

3. Sono disattivate le associazioni fra i nomi dichiarati localmente al blocco e i relativi oggetti denotati;
4. Sono riattivate le associazioni per i nomi, già esistenti all'esterno del blocco che erano stati ridefiniti al suo interno.

In genere le operazioni che si possono fare sui nomi e sull'ambiente sono:

1. **Creazione di un'associazione fra nome ed oggetto denotato (naming)**
2. **Riferimento di un oggetto denotato mediante il suo nome (referencing)**
3. **Disattivazione di un'associazione fra il nome è l'oggetto denotato**: la vecchia associazione non viene distrutta, ma rimane inattiva nell'ambiente fino all'uscita del blocco.
4. **Riattivazione di un'associazione fra il nome e l'oggetto denotato**
5. **Distruzione di un'associazione fra il nome l'oggetto denotato (unnaming)**: quando si esce da un blocco locale (non entrando in un blocco annidato), allora i legami relativi al blocco in uscita sono distrutti.

Per quanto riguarda gli oggetto denotabili sono possibili le seguenti operazioni:

1. **Creazione di un oggetto denotabile**: quest'operazione avviene allocando la memoria necessaria a contenere l'oggetto.
2. **Accesso ad un oggetto denotabile**
3. **Modifica di un oggetto denotabile**: tramite il nome possiamo accedere all'oggetto e quindi modificare il valore in memoria.
4. **Distruzione di un oggetto denotabile**: un oggetto viene distrutto deallocando la memoria che era stata riservata per esso.

Osservazione 2. Non sempre il tempo di vita di un oggetto denotabile, ovvero il tempo che intercorre tra la sua creazione e distruzione, corrisponde al tempo di vita dell'associazione tra tale oggetto e nome.

Può accadere che il tempo di vita di un oggetto denotabile sia maggiore dell'associazione (cosa più probabile), sia che l'associazione duri di più, cosa più strana.

La prima situazione si ha ad esempio quando si fa un passaggio per riferimento a una procedura: anche quando poi si uscirà dalla procedura, l'oggetto continuerà ad esistere.

La seconda situazione si ha quando un nome permette di accedere a un oggetto che non esiste più. Una situazione di questo tipo viene chiamata "**dangling reference**" ed è sinonimo di un qualche errore.

5. Regole di scope

Si divide in scope dinamico e statico.

Scope Statico

Definizione 14: *L'ambiente esistente in un qualsiasi punto del programma ed in un qualsiasi momento dell'esecuzione dipende unicamente dalla struttura sintattica del programma stesso.*

Un tale ambiente può essere dunque determinato completamente dal compilatore, da cui il termine statico.

Ci sono più regole di scope statico, la più importante è quella dello scope annidato più vicino:

1. Le dichiarazioni locali di un blocco definiscono l'ambiente locale del blocco. Queste includo solo quelle presenti nel blocco e non quelle eventualmente presenti in blocchi annidati all'interno del blocco in questione.
2. Se si usa un nome all'interno di un blocco l'associazione valida per tale nome è quella presente nell'ambiente locale del blocco. Se non ne esiste alcuna, allora si considerano le associazioni esistenti nell'ambiente locale del blocco immediatamente sterno, se anche qui non se ne trovano si prosegue in quello più esterno, fin tanto che non si arriva a quello più esterno. Se anche in questo caso non c'è, allora si ha un errore.
3. Un blocco può avere un nome, nel qual caso tale nome fa parte dell'ambiente locale del blocco immediatamente esterno.

Definizione 15: *In un linguaggio con scope statico, si definisce scope[4] di una dichiarazione quella porzione di codice nella quale la dichiarazione è visibile secondo la definizione precedente.*

Osservazione 3. Lo scope statico permette di determinare tutti gli ambienti presenti nel programma semplicemente leggendone il codice. Ciò ha 2 conseguenze:

1. Il programmatore ha una migliore comprensione del programma.
2. Poiché anche il compilatore conosce tutti i collegamenti, ciò fa sì che ci possano essere un maggior numero di controlli di correttezza durante la compilazione e un maggior numero di ottimizzazioni di codice.

Scope dinamico

Semplifica la gestione a run-time dell'ambiente, infatti lo scope statico, a fronte dei vantaggi visti prima ha una gestione difficoltosa del run-time perché gli ambienti non locali evolvono in modo diverso dal normale flusso di attivazione e disattivazione dei blocchi.

Lo scope dinamico determina le associazioni tra nomi ed oggetti seguendo a ritroso l'esecuzione del programma, quindi usano una politica di tipo LIFO (Last In First Out), come ad esempio la pila.

Definizione 16: *Secondo la regola dello scope dinamico, l'associazione valida per un nome X, in un qualsiasi punto P di un programma, è la più recente (in senso temporale) associazione creata per X che sia ancora attiva quando il flusso di esecuzione arriva a P.*

Osservazione 4. La differenza tra scope statico e dinamico interviene solo per la gestione di ambienti che sono contemporaneamente non locali e globali, mentre per gli ambienti locali e quelli globali sono praticamente uguali.

Osservazione 5. Il vantaggio principale dello scope dinamico è di permettere la modifica del comportamento di procedura o di sottoprogrammi senza usare parametri espliciti ma solo ridefinendo delle variabili non locali usate dalla procedura. Questo tuttavia rende più difficile la comprensione del programma.

[4] Anche chiamato portata, campo d'azione, ambito o estensione.

Capitolo 7

Nel caso di una macchina astratta a basso livello, ad esempio una macchina hardware, la gestione memoria è molto semplice è può avvenire totalmente a livello statico. Prima dell'inizio dell'esecuzione, il programma in linguaggio macchina ed i relativi dati vengono disposti in opportune zone di memoria dove rimangono fino al termine.
Se il linguaggio permette la ricorsione però, allora l'allocazione statica non è più sufficiente, infatti il numero di chiamate di procedura attive contemporaneamente può dipendere dai parametri delle procedure o comunque da informazioni disponibili solo a run-time.
Quindi quando abbiamo ricorsione dobbiamo permettere operazioni di allocazione e deallocazione di memoria dinamiche, effettuate cioè durante l'esecuzione del programma. La struttura naturale per gestire questo tipo di memoria è la pila (politica LIFO). Tuttavia non sempre l'uso della pila basta. In alcuni casi torna utile la possibilità di allocare e deallocare manualmente la memoria, come accade ad esempio in C e C++. Per permettere questo si usa una particolare struttura di memoria detta heap.

1. Gestione statica della memoria

Definizione 1: *È la memoria allocata dal compilatore prima dell'esecuzione. Gli oggetti allocati staticamente risiedono in una zona fissa della memoria*[5] *per tutta la durata del programma.*

Osservazione 1. Tipici elementi allocati staticamente sono:

1. Variabili globali
2. Istruzioni del codice oggetto prodotto dal compilatore
3. Costanti
4. Varie tabelle prodotte dal compilatore necessarie per il supporto a run-time del linguaggio.

Se poi il linguaggio non supporta la ricorsione, allora tutti gli altri elementi possono essere gestiti staticamente.

Osservazione 2. Supponiamo che il linguaggio non supporti la ricorsione, il caso di più chiamate di una stessa procedura non comporta l'uso di aree di memoria diversa, in quanto queste chiamate devono avvenire in momenti diversi e quindi condividono la stessa area di memoria.

2. Gestione dinamica mediante pila

La maggior parte dei linguaggi moderni permette una struttura a blocchi, che segue la politica LIFO, infatti se si apre A e poi B non si può chiudere A, ma si deve prima chiudere B.
Supponiamo il seguente codice:

```
A:{
    int a=1;
    int b=0;
    B:{
        int c=3;
        int b=3;
    }
    b=a+1;
}
```

quando entriamo nel blocco A si deve allocare spazio per le variabili a e b, e quando si entra nel blocco B per le variabile b[6] e c. Le operazioni per le quali si alloca spazio per le variabili sulla

[5] Decisa dal compilatore

memoria vengono chiamate **push**. Quando poi si esce dal blocco B, allora la memoria va liberata. Quando ciò succede si parla di **pop**.

Definizione 2: *Lo spazio di memoria, allocato sulla pila, dedicato ad un blocco è detto record di attivazione (RdA) e anche frame.*

Osservazione 3. Nel caso di procedura, per sopperire al problema della ricorsione, l'RdA non può essere legato alla dichiarazione di procedura (poiché se no sarebbe unica), ma è legato alle attivazioni di procedura, infatti per diverse chiamate ci sono diverse aree di memoria.

Definizione 3: *Si definisce pila a run-time, o pila di sistema, la pila su cui sono memorizzati i record di attivazione.*

RdA PER BLOCCHI IN-LINE

Un generico record di attivazione per blocchi in-line è così formato:

1. **Risultati intermedi**: questo è presente soprattutto nel caso in cui si debbano fare calcoli, in quanto può essere necessario memorizzare alcuni valori intermedi.
2. **Variabili locali**: contiene tutte le variabili locali interne a un blocco. Le dimensioni di questo blocco dipenderanno dal numero e dal tipo di queste variabili. Queste informazioni di solito sono note al compilatore che potrà quindi creare correttamente questo blocco. Tuttavia ci sono delle eccezioni, ad esempio gli array dinamici, per i quali non si può conoscere
3. **Puntatore di catena dinamica**: memorizza il puntatore al precedente record di attivazione sulla pila. Viene anche chiamato link dinamico o link di controllo.

L'insieme dei puntatore di catena dinamica è detta catena dinamica.

RdA PER PROCEDURE

Simile al caso precedente, solo che bisogna memorizzare una maggiore quantità di informazioni per gestire correttamente il controllo, si ricordi infatti che, solitamente, una funzione quando termina restituisce un valore al chiamante.

1. Risultati intermedi
2. Variabili locali
3. Puntatore di catena dinamica
4. **Puntatore di catena statica**: serve per gestire le informazioni necessarie a realizzare le regole di scope statico
5. **Indirizzo di ritorno**: contiene l'indirizzo della prima istruzione da eseguire dopo l'esecuzione della funzione.
6. **Indirizzo del risultato**: contiene l'indirizzo di memoria in cui la funzione deposita il valore di ritorno. Questa zona di memoria è interna all'RdA del chiamante.
7. **Parametri**: sono memorizzati i valori dei parametri attuali usati nella chiamata della procedura.

Osservazione 4. Ogni puntatore di un RdA punta a una zona fissa di memoria (usualmente centrale) dello stesso RdA. Gli indirizzi dei vari campi si ottengono aggiungendo un offset positivo o negativo al valore del puntatore.

GESTIONE DELLA PILA

Sebbene la direzione di crescita della pila vari in base alle diverse implementazioni, solitamente questa cresce verso il basso.

Per gestire efficientemente la pila è necessario un puntatore esterno a questa che indica l'ultimo RdA inserito. Questo puntatore viene chiamato **puntatore al record di attivazione** (o anche frame pointer o untatore all'ambiente corrente).

[6] Sì ricordi che le variabili b del blocco A e b del blocco B non sono uguali.

Esiste anche un altro puntatore, detto stack pointer (puntatore alla pila) che indica la prima posizione di memoria libera. Questo puntatore può essere ommesso se punta sempre a una distanza prefissata dall'inizio della parte libera.
La gestione della pila, ovvero inserire e togliere gli elementi dalla pila viene fatto a run-time. Per fare ciò il compilatore inserisce frammenti di codice prima e dopo la chiamata di una procedura oppure di un blocco.
In particolare nel chiamante[7] viene aggiunta una parte di codice detta sequenza di chiamata che è eseguita in parte immediatamente prima della chiamata e in parte dopo la terminazione della chiamata; mentre nel chiamato[8] viene aggiunto un prologo da eseguire subito dopo la chiamata (quindi prima di eseguire la procedura) e un epilogo, da eseguire dopo.
Questi codici si dividono le operazioni necessarie a gestire gli RdA e le chiamate.
Per via di crescite incontrollate come la ricorsione, sebbene vari da implementazione a implementazione, è preferibile aggiungere la maggior parte del codice precedente nel chiamato, così che venga eseguito una sola volta invece che molte.

Alla chiamata si eseguono:

1. Modifica del valore del contatore del programma: il vecchio valore viene salvato per permettere poi il ritorno.
2. Allocazione dello spazio sulla pila: viene predisposto lo spazio per il nuovo RdA e quindi viene spostato il puntatore al successivo spazio vuoto.
3. Modifica del puntatore al RdA
4. Passaggio dei parametri
5. Salvataggio dei registri
6. Esecuzione del codice per l'inizializzazione.

Al momento del ritorno:

7. Ripristino del valore del contatore programma
8. Restituzione dei valori
9. Ripristino dei registri
10. Esecuzione del codice per la finalizzazione
11. Deallocazione dello spazio per la pila, con conseguente modica del puntatore statico.

3. Gestione dinamica mediante heap

La sola gestione tramite pila non è più sufficiente se ci sono comandi espliciti di allocazione e deallocazione di memoria.
Per questo è stata introdotta una speciale memoria detta **heap**. In informatica si riferisce anche ad una struttura dati definita tramite un albero binario a un vettore, usati per implementare code con priorità (heapsort). In questo caso il termine heap si riferisce semplicemente a una zona di memoria in cui i blocchi di memoria possono essere allocati e deallocati in modo relativamente libero.
I metodi di gestione si dividono in due a seconda dei blocchi usati:

1. Blocchi di dimensione fissa

In questo caso l'heap è diviso in una serie di blocchi di dimensione fissa collegati in una struttura a lista della **lista libera.**
Quando si chiede l'allocazione di un dato in memoria, allora viene restituito il puntatore al primo blocco di memoria libero, e successivamente viene aggiornato al successivo blocco libero. Quando invece viene chiesta la deallocazione di un elemento, allora questo viene tolto dalla lista risistemando il puntatore.

Blocchi di dimensione variabile

[7] Programma o procedura che effettua una chiamata di una procedura
[8] Procedura che viene chiamata

Se ad esempio dobbiamo allocare array con elementi di dimensione diversa, allora blocchi di dimensione uguale non vanno più bene, in quanto se abbiamo un elemento di dimensione maggiore dei blocchi non possiamo usare più blocchi per un unico elemento.
In questi bisogna stare molto attenti alla gestione della memoria che può portare a **frammentazione della memoria**. Il fenomeno di frammentazione si divide in due:

1. Frammentazione interna: fenomeno dovuto al fatto che allocando elementi di dimensione minore del blocco, c'è della memoria di ciascun blocco che rimane deallocata e che non viene utilizzata fino alla deallocazione dell'elemento.
2. Frammentazione esterna: si verifica quando la lista libera è composta da blocchi di dimensione piccola per cui, anche se la somma della memoria libera totale è sufficiente, non si riesce ad usare effettivamente la memoria.

Per evitare questi fenomeni, le tecniche di allocazione di memoria tendono a ricompattare la memoria libera unendo i blocchi liberi contigui (evitando frammentazione interna).
Ecco alcune tecniche:

1. Unica lista libera: è un unico blocco costituente l'intero heap. Questo quindi rispecchia il fatto che conviene avere tutti blocchi di dimensioni grandi. Quando viene richiesta l'allocazione di un blocco di n parole di memoria, le prime n parole sono allocate e il puntatore all'inizio dello heap avanza di n. I blocchi di memoria deallocati vengono collegati in una lista libera. Quando si raggiunge la fine dello spazio di memoria dedicato allo heap si dovrà passare ad utilizzare lo spazio di memoria deallocato, e questo può essere fatto in due modi:
    1. Utilizzo diretto della lista libera: viene mantenuta una lista di blocchi di dimensione variabile. Quando si richiede l'allocazione di un gruppo di n parole, allora si cerca un gruppo di dimensione uguale o maggiore di n. Se la differenza tra la dimensione del blocco e n è maggiore di una certa soglia, allora questo va a costituire un nuovo blocco, altrimenti è ammessa la frammentazione interna. La ricerca del blocco può avvenire secondo due politiche:
        - Best-fit: ovvero il blocco che meglio si addice alle dimensioni di memoria
        - First-fit: ovvero il blocco che per primo ha dimensione sufficiente.

        Per evitare invece la frammentazione interna, si controlla se blocchi contigui sono liberi, ed eventualmente vengono uniti. Questo tipo di compattazione è detta parziale.
    2. Compattazione della memoria libera: quando si raggiunge la fine dello heap si spostano tutti i blocchi ancora attivi, ossia tutti quelli che sono stati restituiti alla lista libera vengono spostati a un'estremità dello heap, lasciando così tutta la memoria libera in un unico blocco contiguo. Si aggiorna quindi il puntatore all'inizio di questo blocco di memoria e si riprende l'allocazione come prima.

        Osservazione 5. Non è sempre detto che i blocchi siano spostabili. Nei linguaggi in cui ciò accade non è possibile usare questo metodo.
2. Liste libere multiple: vengono usate più liste per blocchi di dimensione diversa. Quindi quando viene richiesta l'allocazione di un blocco di dimensione n, si cerca la lista con blocchi di dimensione maggiore o uguale a n e viene allocato un blocco di tale lista. Anche in questo caso i blocchi delle liste possono essere statici o dinamici. Nel caso di dimensioni dinamiche due sono i modi di gestione:
    1. Buddy system: le dimensioni dei blocchi sono a potenze di 2: 2^k. Supponiamo di voler allocare un elemento di dimensione n, e supponiamo che 2^k sia la dimensione minima tale che $2^k \geq n$, allora si prenderà un blocco di tale lista. Se tuttavia in quella lista non sono presenti blocchi disponibili, allora si cerca una

lista di dimensione 2^{k+1} e si divide un blocco in due così da formare due blocchi di dimensione 2^k. Uno dei due blocchi viene allocato, mentre l'altro viene spostato nella lita con dimensione 2^k. Questi due blocchi sono chiamati buddies (amici), in quanto quando il blocco occupato viene liberato, va in cerca del suo amico, e se anche questo è libero allora si riuniscono per formare un blocco di dimensione 2^{k+1} e tornare nella loro lista.

2. Heap di Fibonacci: funziona allo stesso modo del buddy-system, solo che al posto di potenze di 2, lavora sui numeri di Fibonacci, che crescono più lentamente di 2^k, e quindi si ha minore frammentazione interna. Poiché i numeri di Fibonacci sono formati dalla somma dei due precedenti numeri partendo da 1 1 (1, 1, 2, 3, 5, 8, 13, …), allora i blocchi avranno le dimensioni dei numeri di Fibonacci. Quando si cerca un blocco di dimensione n, allora si cerca la lista con il numero di Fibonacci che meglio si addice. Se non ci sono blocchi liberi, allora si prende quella con il numero successivo, dividendo il blocco come somma di due blocchi più piccoli secondo la regola di Fibonacci. Questi due blocchi poi si comportano come i buddies.

4. Implementazione delle regole di scope

Dato che gli RdA contengono lo spazio di memoria per i nomi locali, quando si incontra un nome non locale si dovranno esaminare i vari record di attivazione ancora attivi per trovare quello corrispondente al blocco dove il nome in questione è stato dichiarato.

1. Scope statico: catena statica:

L'ordine con cui vengono analizzati gli RdA per risolvere le associazioni è definito dall'ordine degli stessi nella pila, quindi non è detto che l'RdA direttamente collegato dal puntatore di catena dinamica sia necessariamente il primo RdA nel quale cercare di risolvere il riferimento non locale, ma tale primo RdA sarà definito dalla struttura testuale del programma.

Prendiamo il seguente esempio:

```
A:{
      int y=0;
      B:{
            int x=0;
            void pippo (int n){
                  x=n+1;
                  y=n+2;
            }
            C:{
                  int x=1;
                  pippo(2);
                  write (x);
            }
      }
      write(y);
}
```

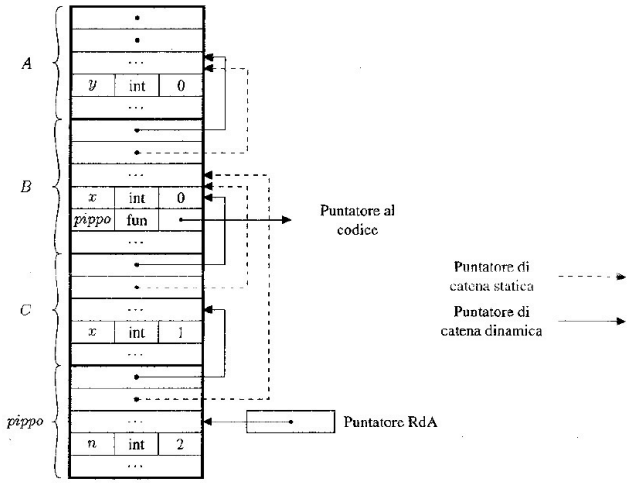


Figura 7. 1 Pila dei RdA con catena statica

Il primo record di attivazione sulla pila (quello più in alto) è relativo al blocco esterno; il secondo è quello per il blocco B, il terzo è per e ed infine il quarto è il record di attivazione per la chiamata di procedura. La variabile non locale x usata nella procedura pippo, come sappiamo dalla regola di scope statico, non è quella dichiarata nel blocco C ma è quella dichiarata nel blocco B. Per poter reperire correttamente questa informazione a tempo di esecuzione, il record di attivazione della chiamata di procedura è collegato da un opportuno puntatore, detto puntatore di catena statica e disegnato tratteggiato in Figura 7.12, al record del blocco contenente la dichiarazione della variabile. Tale record è collegato, a sua volta, da un puntatore dì catena statica al record del blocco A in quanto questo blocco, essendo il primo immediatamente esterno a B, è il primo blocco da considerare per risolvere i riferimenti non locali di B. Quando all'interno della chiamata di procedura pippo si usano le variabili x e y, per reperire la zona di memoria dove queste sono memorizzate si seguono i puntatori di catena statica a partire dal record di attivazione di pippo arrivando così al RdA di B, per la x, e a quello di A, per la y.
Generalizzando dal precedente esempio possiamo dire che, per gestire a run-time la regola di scope statico, il record di attivazione del generico blocco B è collegato dal puntatore di catena statica al record del blocco immediatamente esterno a B (ossia al più vicino blocco esterno che contiene B). Si noti che nel caso in cui B sia il blocco di una chiamata procedura, il blocco immediatamente esterno a B è quello che contiene la dichiarazione della procedura stessa. Inoltre se B è attivo, ossia se il suo RdA è sulla pila, allora anche i blocchi esterni a B che lo contengono devono essere attivi, e quindi si trovano sulla pila.
Quindi affianco alla catena dinamica, costituita dai record presenti sulla pila di sistema, esiste una catena statica costituita dai vari puntatori di catena statica usata per rappresentare la struttura statica di annidamento dei vari blocchi del programma.
La gestione della catena statica a run-time rientra tra le funzioni svolte dalla sequenza di chiamata dal prologo e dall'epilogo visti in precedenza. Secondo l'approccio più comune il chiamante calcola il puntatore di catena statico del chiamato e quindi passa tale puntatore al chiamato. Ci possono essere due casi:

1. **Il chiamato si trova all'esterno del chiamante**: in base alle regole di visibilità definite dallo scope statico, affinché il chiamato sia visibile si deve trovare in un blocco esterno che includa anche il blocco del chiamante. Ciò è come dire che l'RdA di tale blocco si deve trovare già sulla pila. Supponendo che ci siano k livelli di annidamento tra chiamante e chiamato. Questo valore è determinabile dal compilatore dato che dipende unicamente dalla struttura statica del programma.
2. **Il chiamato si trova all'interno del chiamante**: in questo caso le regole di visibilità assicurano che il chiamato è dichiarato immediatamente nel blocco in cui avviene la

chiamata e quindi il primo blocco esterno al chiamato è proprio il blocco in cui avviene la chiamata, e quindi il primo blocco esterno al chiamato è proprio quello del chiamante.

Una volta che il chiamato ha ricevuto il puntatore di catena statico deve soltanto memorizzarlo in un'apposita area del proprio record di attivazione (eseguita al prologo).

Definizione 4: *Si definisce **tabella dei simboli** una tabella creata dal compilatore per tenere traccia del libello di annidamento delle varie chiamate di procedura. Questa tabella contiene i nomi usati nel programma e tutte le informazioni necessarie per gestire gli oggetti denotati.*

In particolare nella tabella vengono identificati e numerati i vari scope in base al livello di annidamento e quindi ad ogni riferimento viene anche associato un numero che indica lo scope contenente il nome e l'associazione. In base a tale numero si può calcolare a tempo di compilazione la distanza fra chiamante e chiamato.

Osservazione 6. Questa distanza permette anche di risolvere a run-time i riferimenti non locali senza dover effettuare alcuna ricerca negli RdA. Basterà infatti scorrere la catena statica di n posizioni pari alla distanza, partendo dall'RdA che contiene il riferimento.

Comunque il compilatore non può risolvere completamente in modo statico un riferimento a un nome non locale. Questo perché non è possibile conoscere staticamente il numero di RdA presenti sulla pila.

Scope statico: display

La realizzazione dello scope statico mediante catena statica ha un inconveniente: se dobbiamo usare un nome non locale dichiarato in un blocco esterno di k livelli, allora sarà necessario fare un accesso alla memoria per k volte, il che è molto costoso in termini di tempo[9].

Il metodo del display permette di ridurre k a un valore pari a 2.

Definizione 5: *Il display è un vettore contenente tanti elementi quanti sono i livelli di annidamento dei blocchi presenti nel programma, dove l'elemento k-esimo è l'RdA correntemente attivo.*

Quando ci si riferisce ad un oggetto non locale, dichiarato in un blocco esterno di livello n, si può reperire l'RdA contenente tale riferimento semplicemente accedendo alla k-esima posizione del vettore e seguendo il puntatore presente in quella posizione.

La gestione del display è leggermente più costosa in quanto quando si entra e si esce da un ambiente bisogna aggiornare il valore del puntatore del vettore e salvare il vecchio valore.

Scope dinamico: lista di associazioni e CRT

È concettualmente molto più semplice come implementazione rispetto allo scope statico. Questo perché gli ambienti non locali si considerano nell'ordine con cui sono attivati a run-time. Quindi teoricamente, per risolvere un riferimento non locale basterà ripercorrere a ritroso la lista.

Le varie associazioni fra nome e oggetti denotati che costituiscono gli ambienti locali possono essere memorizzate direttamente nei record di attivazione. Se questo è il caso, è sufficiente quanto detto prima per poter risolvere i riferimenti.

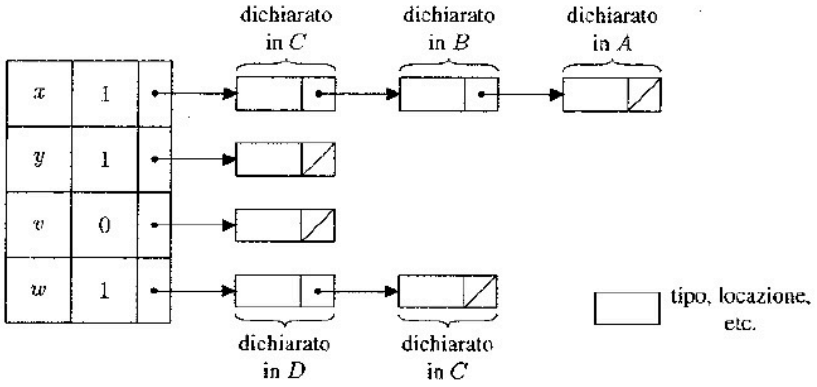
Ci sono però delle varianti. Una di queste è l'uso delle A-lists, ovvero una lista di associazioni gestita come fosse una pila. Quando l'esecuzione di un programma entra in un nuovo ambiente le nuove associazioni locali vengono inserite nella A-list, mentre quando si esce da un ambiente vengono rimosse.

In tutti e due i metodi usati si hanno degli inconvenienti, ovvero:

1. I nomi devono essere memorizzati in strutture presenti a tempo di esecuzione.
2. Inefficienza della ricerca a run-time del nome sulla pila o tra gli RdA.

[9] Questa situazione non è così tragica in quanto di solito non sono molti i livelli di annidamento

Per evitare questi problemi si usa un altro metodo detto **tabella centrale dell'ambiente**. Questo metodo ha una minore efficiente nelle operazioni di entrata ed uscita da un blocco, ma evita gli altri due problemi.
Secondo questo metodo tutti i blocchi del programma fanno riferimento ad un'unica tabella centrale (**CRT** "Central Referencing Table"). In questa tabella sono presenti tutti i nomi usati nel programma e per ogni nome è presente un flag che indica se l'associazione per quel nome è attiva o meno. Se assumiamo che tutti gli identificatori usati nel programma siano noti a tempo di compilazione, ogni nome può avere una posizione fissa nella tabella: A run-time l'accesso alla tabella può quindi avvenire in tempo costante, sommando all'indirizzo di memoria dell'inizio della tabella un offset relativo alla posizione del nome che ci interessa. Se invece non tutti i nomi sono noti a tempo di compilazione, la ricerca della posizione di un nome nella tabella può essere fatta efficientemente a run-time usando tecniche di hashing.
Come detto però, le operazioni di entrata e uscita da un blocco sono più difficoltose. La struttura usata ancora una volta è la pila: quando si entra in un blocco si modificano le associazioni e in caso deattivando quelle per le quali sono fatte delle nuove associazioni. Queste tuttavia vanno salvate in quanto dovranno successivamente essere ripristinate. Per fare questo nel modo più efficiente, si usano delle pile dentro la pila: quando viene creata una nuova associazione, si aggiunge alla pila della vecchia associazione quella nuova, così che quella deattiva non venga vista. Quando poi si esce dal blocco si toglie l'elemento "coprente" e quindi si ritorna all'associazione precedente.



Si possono anche usare delle pile nascoste: quando un elemento viene deattivato, viene spostato in una pila esterna. La tabella quindi conterrà oltre al nome e a un flag, anche un puntatore per poi riottenere il valore disattivato.

Capitolo 8

1. Espressioni

Definizione 1: *Un'espressione è un'entità la cui valutazione produce un valore oppure non termina, nel qual caso l'espressione è indefinita.*

La differenza tra un'espressione e un comando è appunto che l'espressione produce un valore.
Un'espressione è solitamente composta da un'entità singola, quali una variabile o una costante, oppure da un operatore applicato a un certo numero di operandi (a loro volta espressioni).
Per rappresentare quest'ultimo tipo di espressioni spesso si usano espressioni ad albero. Questo però non risulta comodo nel momento della scrittura del codice di un programma, ma si preferisce usare una scrittura lineare.

1. Notazioni

Le principali notazioni usate sono 3:

1. **Notazione infissa**: il simbolo di un operatore binario è posto fra le espressioni che rappresentano i due operandi (2+2). Per evitare ambiguità nell'applicazione degli operatori agli operandi è necessario l'uso delle parentesi, e di opportune regole di precedenza. Questo tipo di notazione, per il fatto che è la più simile a quella matematica è quella che viene maggiormente usata nei linguaggi di programmazione.
2. **Notazione** (polacca) **prefissa**: il simbolo che rappresenta l'operatore precede i simboli che rappresentano gli operandi, da sinistra a destra (+ 2 2). Usando questo tipo di notazione non sono necessarie parentesi o regole di precedenza fra gli operatori, purché sia noto il numero di operandi che ogni operatore vuole (**arietà**).
   La maggior parte dei linguaggi usa la notazione prefissa per gli operatori unari e per le funzioni definite dall'utente
   a. LISP usa per le funzioni una variante che è chiamata **polacca di Cambridge** che include anche gli operatori fra parentesi
3. **Notazione postfissa**[10]: è uguale alla notazione prefissa con la differenza che il simbolo segue gli operandi (2 2 +). Questo tipo di notazione è per lo più usato dai compilatori

Notazione infissa	(a+b)*(c+d)
Notazione prefissa	* + a b + c d * (+ a b) (+ c d)
Polacca di Cambridge	(* (+a b) (+ c d))
Notazione postfissa	a b + c d + * (a b +) (c d +)*

I vantaggi di usare una notazione polacca in confronto a una notazione infissa sono:

1. Uso di parentesi superfluo (vantaggio anche nei confronti della polacca di Cambridge).
2. Possibilità di avere operatori che necessitano di un maggior numero di operandi senza la necessità di ricorrere a operatori ausiliari.
3. Permettono di valutare un'espressione in modo molto semplice.

2. Semantica delle espressioni

Di seguito le regole usate nelle varie notazioni.

1. **Notazione infissa**: la facilità d'uso si paga con meccanismi di valutazione delle espressioni più complicati. Infatti se non si usano estensivamente le parentesi è necessario chiarire le precedenze fra i vari operatori. Se prendiamo un caso della matematica di tutti i giorni il

[10] Anche chiamata polacca inversa

risultato è banale: 3-2*2=3-(2*2) e non a (3-2)*2. Tuttavia se prendessimo ad esempio il Pascal e scrivessimo: x=4 and y=5, il risultato non sarebbe più banale. Infatti in Pascal si è deciso che l'operatore and ha precedenza sull'assegnazione, e per tanto il risultato che si otterrebbe sarebbe x=(4 and y)=5, ovvero un errore.
Come si ha capito i vari linguaggi differiscono per le regole di precedenza adottate, anche se solitamente vengono mantenute valide le regole matematiche.
Un secondo problema degli operatori è la loro associatività. Sebbene solitamente gli operatori associno da sinistra a destra, come ad esempio il - , tuttavia ci sono degli operatori che associano da destra a sinistra, come ad esempio l'esponenziale. Basti pensare infatti a: $5^{2^3} = 5^6$.
La difficoltà di creare un algoritmo per questo tipo di notazione è che non può essere effettuata un'unica scansione, infatti in alcuni casi è necessario prima valutare la parte successiva di un'espressione per poi valutare quella precedente.

2. **Notazione prefissa**: in questo caso l'algoritmo è molto più semplice. Infatti basta una sola scansione e l'uso di una pila. L'unico problema che si può incontrare è che bisogna conoscere preventivamente il numero di operandi sulla pila. Inoltre bisogna controllare che sulla pila ci siano abbastanza operandi per l'operatore più in alto presente sulla pila stessa.
3. **Notazione postfissa**: questo tipo di notazione non ha neanche bisogno del controllo degli operandi per l'ultimo operatore, in quanto venendo prima gli operandi e poi l'operatore, questi verranno automaticamente associati all'operatore. Anche in questo caso tuttavia è necessario conoscere preventivamente il numero di operandi per ogni operatore.

Come accennato prima ogni espressione può essere vista come un albero dove ogni nodo che non è una foglia è un operatore, ogni nodo che è una foglia p etichettato con una costante, una variabile o un altro operatore elementare, e ogni sottoalbero che ha come radice un figlio di un nodo N costituisce un operando per l'operatore associato a N. poiché i sottoalberi più in basso costituiscono gli operandi per le espressioni più in alto, vanno valutati prima quelli in basso.
La rappresentazione ad albero di un'espressione chiarisce precedenza e associatività degli operatori.
Detto questo sorge un altro problema nella notazione ad albero. Infatti bisogna creare delle regole per definire l'ordine in cui i sottoalberi dello stesso livello di uno stesso operatore debbano essere eseguiti. Se questa cosa non influenza i calcoli dal punto di vista matematico, ciò non è vero in informatica. Infatti uno dei problemi più grandi legato a questo aspetto è quello che viene chiamato **side-effects** (**effetti collaterali**). Nei linguaggi di tipo imperativo la valutazione di un'espressione può modificare il valore di alcune variabili mediante appunto gli effetti collaterali. Per questo motivo alcuni linguaggi vitano completamente alcuni effetti collaterali, altri, ed è il caso più diffuso, li permettono ma specificano chiaramente nella definizione del linguaggio, l'ordine degli operandi.
Un altro grande problema è dovuto all'**aritmetica finita** dei calcolatori. Infatti i numeri che un computer può rappresentare sono finiti[11]. Supponiamo ad esempio 3 numeri: a, b e c, dove a è il numero massimo rappresentabile e b>c. Prendiamo quindi l'espressione a-b+c. Se il linguaggio definisce che tale espressione associa da sinistra non ci saranno problemi, infatti avremmo un valore minore del massimo rappresentabile, mentre se associa da destra, allora avremmo il cosiddetto fenomeni di overflow.
Si possono poi verificare dei casi in cui gli operandi non sono definiti. Questo si basa più che altro sul tipo di strategia di valutazione che si usa. Le principali strategie sono 2:

1. **Eager**: si valutano prima tutti gli operandi e poi si passa ad applicare l'operatore ai valori ottenuto dalla valutazione degli operandi.

[11] Si intende anche la precisione con cui i calcolatori possono rappresentare numeri in virgola, limitazione dovuta alla base 2.

2. **Lazy**: consiste nel non valutare gli operandi prima dell'operatore, ma di passarli direttamente all'operatore, che deciderà poi quali sono quelli effettivamente necessari valutando solo quelli.

Consideriamo ad esempio un'espressione condizionale del tipo:

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

Se usassimo una strategia di tipo eager si otterrebbe un errore. Infatti valutando prima gli operandi si valuterebbe anche b/a, ma nel caso in cui a=0 si finirebbe in errore.
Con la strategia lazy invece, si valuterebbe prima se a=0, e in tal caso si restituirebbe b, in caso contrario b/a.
Poiché questa tecnica è molto più costosa da implementare, la maggior parte dei linguaggi imperativi adotta la strategia di tipo eager. Vi sono ovviamente anche i soliti mix, come quello usato da ALGOL.
Il problema presentato precedentemente si ripresenta nelle espressioni booleane in maniera ancora più decisa. Consideriamo ad esempio l'espressione:

```
a==0||b/a>2
```

Se la prima condizione fosse verificata, essendo poi seguita da un or, non ha senso verificare anche la seconda (o tutte le altre) in quanto l'espressione valuterebbe direttamente a true. Allo stesso modo, in negativo, se avessimo l'espressione:

```
a==0&&b/a>2
```

con a≠0, allora non sarebbe necessario valutare le espressioni seguenti in quanto l'end logico valuterebbe sempre a false.
Questo tipo di valutazione, molto importante, è detto **valutazione con corto-circuito**.
Si noti che non tutti i linguaggi mettono a disposizione questo tipo di valutazione, e che alcuni linguaggi mettono a disposizione operatori particolari nel caso in cui si voglia valutazione con corto-circuito o meno.
Per problemi di ottimizzazione alcuni compilatori possono cambiare l'ordine degli operandi nelle espressioni per ottenere codice più efficiente, ma semanticamente equivalente. Questo provoca in alcuni casi degli errori la cui origina è di difficile identificazione.

2. Comandi

Definizione 2: *Un comando è una entità sintattica la cui valutazione non necessariamente restituisce un valore, ma può avere un effetto collaterale.*

Un comando ha un effetto collaterale se esso influenza il risultato della computazione senza che la sua valutazione restituisca alcun valore.

1. La variabile

La definizione di variabile varia molto da paradigma imperativo a quello funzionale. Il paradigma imperativo classico usa la variabile modificabile, vista come una sorta di contenitore al quale si può dare un nome e che può contenere dei valori. Questi valori possono cambiare nel tempo in seguito all'esecuzione di comandi.
In alcuni linguaggi imperativi la variabile non è vista come il contenitore di un valore, ma come un riferimento a un valore (solitamente) sullo heap.
I linguaggi funzionali puri usano una nozione di variabile più simile a quella matematica: una variabile non è altro che un identificatore che denota un valore. Una volta creato un legame fra identificatore di una variabile e un valore, tale legame non può essere eliminato. Vi è però il modo, pur non modificando il legame, di modificare il valore associato ad una variabile.

2. Assegnamento

È il comando di base che permette di modificare il valore delle variabili modificabili (e quindi dello stato) in un linguaggio imperativo.

L'assegnamento è un comando con effetto collaterale perché non ha di per sé restituito nulla.
Si presti attenzione all'espressioni come:

```
x=x+1;
```

Poiché le 2 x non hanno lo stesso significato. Infatti molti linguaggi imperativi le differenziano parlando di l-value (left value, ovvero che stanno a sinistra) per la prima x a cui viene assegnato il valore e r-value (right value) per la seconda x, di cui si prende il valore per sommarlo a 1.
Quello che fa l'operatore di assegnazione quindi è quello di calcolare l' l-value dell'operando di sinistra, calcolare il valore dell'espressione dell'operando di destra e assegnare al contenitore denotato dall'l-valore l'espressione ottenuta.
Ci sono poi delle ottimizzazioni agli operatori di assegnazioni, quali il +=, -=, *=, x++, ++x (questi tipici del C).

3. Comandi per il controllo di sequenza

Questi comandi servono per specificare l'ordine con cui le modifiche di stato espresse dagli assegnamenti devono essere effettuate. Questi comandi possono essere divisi in 3 categorie:

1. Comandi per il controllo di sequenza esplicito
2. Comandi condizionali
3. Comandi iterativi

1. Comandi per il controllo di sequenza esplicito

Si divide a sua volta in:

1. Comando sequenziale: è il comando che indica dove termina un comando. Solitamente viene indicato con un ;
2. Comando composto: è un raggruppamento di comandi in un blocco.
3. Goto: è presente sin dai primi linguaggi di programmazione, ed è ispirato direttamente dalle istruzioni dei linguaggi assembly.
   L'esecuzione del comando
   ```
   goto A
   ```
   trasferisce il controllo al punto del programma nel quale è presente l'etichetta A.
   Nonostante l'apparente semplicità e naturalezza, il comando goto è stato al centro di un acceso dibattito a partire dagli anni '70 e, dopo circa 30 anni di discussioni, possiamo dire che i detrattori hanno avuto la meglio sui sostenitori di questo comando.
   Questo dibattito era per prima cosa incentrato sul fatto che il goto non è essenziale per l'espressività di un linguaggio di programmazione: un teorema di Böhm e Jacopini infatti dimostra che un qualsiasi programma può essere "tradotto" in uno, equivalente, che non usi il goto. Il nocciolo della questione in realtà non è di tipo teorico ma è di natura pragmatica. Usando il goto si può facilmente scrivere codice che diventa ben presto incomprensibile e che rimane tale anche in caso di una eventuale eliminazione successiva del goto. Si pensi, ad esempio, ad un programma di una certa dimensione dove siano inseriti dei salti fra punti distanti qualche migliaio di linee di codice. Oppure si pensi all'uscita da sottoprogrammi realizzata, mediante goto, in punti diversi a seconda delle condizioni che si verifichino.
   Questi ed altri usi arbitrari di questo costrutto rendono il codice difficilmente comprensibile, e quindi di difficile modifica, correzione e manutenzione, con ovvie conseguenze negative anche in termini di costi. A tutto ciò si aggiunge che il goto, col suo modo primitivo di trasferire il controllo, mal si accorda con molti altri meccanismi presentì nei linguaggi di alto livello.
4. Altri: poiché in alcune situazioni può tornare utile l'uso di un goto, ma senza tutta la sua "forza", alcuni linguaggi di programmazione mettono a disposizione dei costrutti di salto

più limitati, quali **break**, **continue**, o **return**. Il primo costrutto entra in gioco in cicli iterativi o nei case dello switch; il continue serve per terminare l'esecuzione di un'iterazione e iniziare con quella successiva e il return viene solitamente impiegato per terminare l'esecuzione di una funzione e ritornare il controllo al chiamante, con, solitamente, anche un valore.

Un'ultima possibilità è l'uso di eccezioni, che verrà trattato nel capitolo successivo.

2. Comandi condizionali

Esprimono un'alternativa fra due o più possibilità. Possono essere divisi in 2 grandi gruppi:

1. if: introdotto per la prima volta in ALGOL 60, è presente in quasi tutti i linguaggi imperativi e dichiarativi. Le varie forme sintattiche si possono ricondurre alla forma:

```
if Bexp
   then C1
   else C2
```

cha sta a significare: se l'espressione Bexp è true, allora esegui il comando C1, altrimenti il comando C2. Spesso si ommette il ramo dell'else nel qual caso, se Bexp dovesse risultare falsa, non si eseguirà nessun commando. Ci possono essere degli if annidati, nella forma

```
if Bexp
   then if Bexp
              then C1
              else C2
   else C3
```

O alcuni linguaggi ammettono anche la forma semplificata:

```
if Bexp
   then C1
   elseif C2
   else C3
```

Solitamente la valutazione dell'espressione Bexp avviene per mezzo della tecnica del corto-circuito discussa in precedenza.

2. case: è una specializzazione del comando if con più rami. Nella sua forma più semplice può essere scritto come:

```
case Exp of
    label1 : C1;
    label2 : C2;
    ...
    labeln : Cn;
   else C(n+1)
```

Dove Exp deve essere un'espressione di tipo compatibile con quello di label1, label2…labeln. In pratica si valuta Exp, e si trova il valore uguale tra i label, quindi si esegue il comando corrispondente. Il comando C(n+1) viene eseguito come default nel caso in cui Exp non corrisponda a nessuno dei label.

3. Comandi iterativi

Un linguaggio con i comandi finora elencati non sarebbe di certo Turing completo perché di limitata espressività. Infatti non abbiamo visto comandi che permettano di ripetere finitamente, o meno, parti di codice.

Nei linguaggi di basso livello si usano degli appositi algoritmi per fare dei salti "all'indietro" e ripetere del codice.

Nei linguaggi di alto livello invece, per evitare comandi di salto, si usano due meccanismi che sono l'**iterazione** e la **ricorsione**.

Il primo è più utilizzato nei linguaggi imperativi, mentre il secondo è predominante in quelli funzionali e logici.

- **Iterazione**: si divide in:
  - **Iterazione indeterminata**: è realizzata da costrutti linguistici costituiti da due parti: condizione (o guardia) del ciclo e corpo. La guardia ha il compito di verificare una condizione, mentre il corpo contiene un comando (spesso composto) che viene ripetuto fin tanto che la condizione della guardia non diventa vera (o falsa a seconda dei costrutti).
    Vediamo solo due casi molto noti: il ciclo while:
    ```
    while Bexp
        do Cmd
    ```
    implementato per la prima volta con ALGOL sta a significare: finché la condizione Bexp è verificata esegui Cmd.
    Una variante è quella in cui la condizione è verificata alla fine del ciclo (do while). Questo costrutto è molto comodo soprattutto per gli input, e tutti i costrutti che hanno questa forma vengono chiamati repeat until.
  - **Iterazione determinata**: realizzata da costrutti linguistici più complessi di quelli per l'iterazione indeterminata. In genere la forma che assumono questi costrutti può essere riassunta con la seguente formula:
    ```
    for i=inizio to fine by passo do
        corpo
    ```
    ovvero partendo da un valore i iniziale, ogni "passo" esegui il corpo, fintanto che non arrivi a una condizione falsa.
    La variabile i viene detta indice o contatore. I valori inizio e fine devono essere di tipo discreto. Passo invece è una costante intera diversa da zero.
    Per prima cosa con questo ciclo vengono valutate le espressioni di inizio e fine, che vengono in un certo senso congelate. Questo è molto importante perché permette di valutare facilmente il numero di cicli che verranno eseguiti come:
    $$ic = \left\lceil \frac{(fine - inizio + passo)}{passo} \right\rceil$$
    Poiché è stato passo≠0, si ha che ic non può essere infinito.
    Si noti che questo costrutto non corrisponde al costrutto del for tipico del C (e di Java), che verrà visto successivamente.
    I linguaggi variano sotto molti aspetti la semantica del loro for, in particolare:
    1. Numero di iterazioni: in alcuni casi il test viene eseguito dopo il corpo, per tanto anche se si è in una situazione iniziale in cui fine<inizio, si esegue un giro.
    2. Passo: la richiesta che passo sia una costante è necessaria per calcolare staticamente il segno, per generare il codice opportuno (il valore di i deve essere minore uguale del valore finale nel caso in cui sia positivo, oppure maggiore uguale nel caso in cui sia negativo). Alcuni linguaggi per ovviare a questi problemi hanno sostituito il to con altre sintassi per specificare se il passo è negativo. Dall'uso di questa tecnica deriva il nome di iterazione controllata numericamente.
    3. Variabile finale dell'indice: in molti linguaggi la variabile i è visibile anche fuori dal ciclo. Questo apre un problema su quale valore assuma, se l'ultimo valore prima di non essere più valida per il ciclo, oppure il valore successivo. Alcuni linguaggi come FORTRAN e Pascal lasciano indefinito questo punto, generando un sacco di problemi per quanto riguarda la portabilità del programma. La soluzione più semplice è quella di decretare i come variabile interna e quindi non visibile esternamente.

4. Salto nel ciclo: alcuni linguaggi permettono di eseguire salti all'interno del for per mezzo del comando goto.

Si può facilmente notare che un costrutto di iterazione determinata (for) può essere trasformato in uno di iterazione indeterminata (while), mentre non vale il contrario. Si dice quindi che l'iterazione indeterminata ha una maggiore espressività.
Ci si può quindi chiedere perché utilizzare un ciclo for al posto che uno while. La risposta sta nel fatto che il ciclo for, mettendo gli elementi essenziali (inizio, fine e step) sulla stessa riga dà una maggiore chiarezza al codice. Inoltre alcuni linguaggi implementano il ciclo for in modo più efficiente.
Vediamo ora il for in C. La versione generale è:

```
for (exp1, exp2, exp3)
      corpo
```

Tradotto quello che avviene è:

a. Si valuta exp1
b. Si valuta exp2, se falsa allora si ferma il for
c. Si esegue il corpo
d. Si valuta exp3 e si riprende dal punto b.

Come si può notare non c'è tentativo di congelare il valore delle espressioni di controllo, né vi è alcun vincolo sulla possibilità di modificare il valore dell'indice.
Si nota subito che in C il for è in tutto e per tutto un ciclo while.
Un altro tipo di ciclo for è quello che viene chiamato for-each. Il for-each è fondamentale per costrutti come liste o array. Permette di scorrere tutta una lista (o array) prendendo ciascun elemento ed eseguendo un determinato comando.

4. Programmazione strutturata

il punto di partenza di tutto fu la "rivolta" contro il goto degli anni 70. Sebbene la cosa più conosciuta, era anche solo la punta dell'iceberg. Da quel momento andò sviluppandosi un processo molto più ampio che portò a una serie di "prescrizioni" volte a permettere uno sviluppo il più possibile strutturato del codice e corrispondentemente del flusso di controllo. Alcuni punti salienti erano i seguenti:

1. **Progettazione del programma top-down o comunque gerarchica**: il programma è sviluppato per raffinamenti successivi, ovvero si parte da una specie di bozza per migliorare man mano il codice.
2. **Modularizzazione del codice**: è opportuno raggruppare i comandi che si riferiscono a una specifica funzione dell'algoritmo.
3. **Uso di nomi significativi**: per quanto banale, usare dei nomi significativi per le varie variabili rende il codice molto più leggibile. Basta immaginarsi programmi molto pesanti, anche di centinaia di righe solo per una funzione, quando bisogna controllarla o migliorarla si esce di testa se le variabili sono messe casualmente.
4. **Uso estensivo dei commenti**: non ci possiamo leggere il pensiero, né tanto meno attraverso del codice, quindi è bene spiegare ogni funzione e anche alcuni passaggi per mezzo di commenti.
5. **Uso di tipi di dato strutturato**: se il linguaggio mette a disposizione dei costrutti per raggruppare informazioni, e nel codice vediamo che alcune informazioni stanno bene raggruppate è sempre bene farlo. Mantiene il codice più leggibile e più facile da mantenere.
6. **Uso di costrutti strutturati per il controllo**: usare dei costrutti che abbiano un solo punto di ingresso e un solo punto di uscita. Niente goto. Pochi break. Evitare return multipli.

Se l'ultimo punto è violato si arriva a un tipo di codice che viene detto "codice spaghetto".

Si noti che non sempre l'uso di break/goto/continue porta a una ramificazione del codice. Ad esempio, se volessimo elaborare tutti gli elementi di un file che leggiamo dall'esterno potremmo usare un codice della forma

```
while true do{
    read(X) ;
    if X = end_of_file then goto fine;
    elabora ( X ) ;
}
fine: ...
```

Si osservi che quest'uso di goto non viola il principio "un solo ingresso e una sola uscita", perché il salto non fa che anticipare l'uscita, che avviene comunque in un unico punto per tutto il costrutto.
Il comando strutturato break (o suoi analoghi) è la forma canonica di questo "salto alla fine di un ciclo": sostituito al posto di goto fine rende più chiaro il programma ed evita l'introduzione di un'etichetta.

5. Ricorsione

È un meccanismo, alternativo all'iterazione, per ottenere linguaggi di programmazione Turing equivalenti.
Una funzione ricorsiva può essere definita come una funzione nel cui corpo contiene dei richiami a sé stessa. Si definiscono poi funzioni mutualmente ricorsive due funzioni che si richiamano a vicenda.
Le funzioni ricorsive derivano direttamente dal principio di induzione matematico. Se il dominio su cui f è tale da non ammettere catene infinite di elementi sempre più piccoli, allora diminuendo man mano il valore siamo sicuri che dopo un numero finito di applicazioni della funzione f, si arriva a un caso terminale, dal quale possiamo ricostruire il valore di f applicata al valore iniziale.
Vi è tuttavia una differenza fondamentale con l'induzione matematica: in matematica non tutte le possibili definizioni di funzioni in termini di sé stessa vanno bene.
Si noti che questo tipo di ricorsione può provocare una crescita incontrollata dello stack.

1. Ricorsione in coda

Si può però adottare uno stratagemma per evitare un'esplosione dello stack. Infatti se al posto che utilizzare sempre nuovo spazio si riutilizzasse la stessa cella, lo spazio necessario sarebbe molto ridotto.

Definizione 3: *Si definisce chiamata in coda (tail recursion) la chiamata di una funzione f a una funzione g (uguale o diversa da f), se la funzione f restituisce direttamente il valore di g senza dover fare ulteriori computazioni.*

Definizione 4: *La funzione si dice ricorsiva in coda (tail recursive) se tutte le sue chiamate ricorsive presenti in f sono chiamate in coda.*

Solitamente è sempre possibile passare da una funzione ricorsiva a una funzione ricorsiva in coda. Questo può essere fatta ponendo la parte di lavoro prima della chiamata ricorsiva (ove possibile) o come parametro nel caso in cui non sia possibile.
Tutto ciò non ha senso se si parla di funzioni di ordine superiore.

2. Ricorsione o iterazione?

Per prima cosa va ricordato che sono metodi alternativi per esprimere lo stesso metodo espressivo.
Molto spesso la scelta dipende, oltre dal piacere del programmatore, da quello che bisogna fare: in casi in cui si lavora con matrici o cose del genere viene più naturale l'uso dell'iterazione, invece in casi come lettura o scrittura di alberi, viene più immediato l'uso della ricorsione.
Sebbene molte funzioni ricorsive possano essere inefficienti, allo stesso tempo possono essere trasformate in tail recursive rendendole molto più efficienti.

Capitolo 9

In un linguaggio di programmazione solitamente si dividono due tipi di astrazione:

1. Sul controllo: permette di nascondere dettagli procedurali
2. Sui dati: permette di utilizzare tipi di dati complessi senza far riferimento a come questi siano implementati.

1. Astrazione funzionale

Si parla di astrazione funzionale quando le azioni di specifica, implementazione ed uso di una funzione avvengono in modo indipendente l'una dall'altra e senza conoscere il contesto nel quale le altre azioni avverranno.
Ossia l'astrazione funzionale permette di mettere a disposizione della gente funzioni senza far conoscere come lavorano, perché non è quello che interessa.
L'astrazione funzionale è tanto più garantita quanto più l'interpretazione tra componenti è limitata al comportamento esterno espresso dall'intestazione delle funzioni.

2. Passaggio dei parametri

Definizione 1: *Si parla di modalità di passaggio dei parametri quando ci si riferisce alle modalità con cui i parametri attuali sono accoppiati con quelli formali.*

La modalità è fissata al momento della definizione della funzione, può essere distinta da parametro a parametro e si applica a tutti gli usi della funzione.
Un parametro può essere

1. Di ingresso: permette una comunicazione unidirezionale dal chiamante alla funzione
2. Di uscita: permette una comunicazione unidirezionale dal chiamato al chiamante
3. Dia d'ingresso che d'uscita: permette una comunicazione bidirezionale.

Ecco alcune modalità di passaggio dei parametri:

1. **Passaggio per valore**: il passaggio per valore è una modalità che corrisponde ad un parametro di ingresso. L'ambiente locale della procedura è esteso con un'associazione tra il parametro formale ed una nuova variabile. Al termine della procedura questo viene eliminato come tutto l'ambiente locale. Durante l'esecuzione del corpo non c'è alcun legame tra il parametro formale e quello attuale (tra la variabile che viene passata e il valore all'interno della funzione).
   Si noti come si tratta di una modalità costosa nel caso in cui il parametro per valore corrisponda ad una struttura dati di grandi dimensioni.
   Come vantaggio ha che l'accesso al parametro formale è minimo dal momento che coincide con il costo di accesso ad una variabile locale.
2. **Passaggio per riferimento**: il parametro è sia d'ingresso che d'uscita. Il parametro attuale deve essere un'espressione dotata di l-value. Al momento della chiamata viene valutato questo valore che viene associato a un parametro formale estendendo l'ambiente locale.
   Al termine della procedura viene distrutto sia l'ambiente locale che l'associazione.
   Poiché ogni volta che si accede al parametro formale, si accede alla zona di memoria del parametro attuale, allora ogni modifica che avviene sul parametro formale avviene anche su quello attuale.
   Si tratta di una modalità di passaggio dal costo molto contenuto in quanto si deve solo memorizzare un indirizzo. Anche il costo di accesso a questo indirizzo è relativamente basso in quanto si tratta di un accesso indiretto che può essere realizzato a basso costo su molte macchine.
3. **Passaggio per costante**: prende il meglio del passaggio per riferimento, e il fatto che il passaggio per valore sia solamente in ingresso. Ossia, tramite il passaggio per costante si

esegue un passaggio per riferimento, ma il parametro formale non può essere modificato, e così facendo non viene modificato neanche quello attuale.

4. **Passaggio per risultato**: è l'esatto duale del passaggio per valore, ossia è di sola uscita. I parametri passati non vengono usati per leggere il valore, ma solo per passare alla fine il valore, quindi il parametro attuale deve avere l-value.
5. **Passaggio per valore-risultato**: combinazione del passaggio per risultato e per valore.
6. **Passaggio per nome**: fu introdotto in ALGOL e oggi giorno non è più utilizzato. I progettisti di ALGOL lo definirono come segue:

Definizione 2: *Sia f una funzione con parametro attuale x e sia a un'espressione compatibile col tipo di x. Una chiamata a f con parametro attuale a è semanticamente equivalente all'esecuzione del corpo di f nel quale tutto le occorrenze del parametro formale z sono sostituite da a.*

Sembrerebbe quindi trattarsi di una semplice sostituzione, ma è facile osservare come è più complicato di così. Si prenda il seguente esempio:

```
int x=0
int foo(name int y){
    int x=2;
    return x+y;
}
foo (x+1);
```

in questo caso ad esempio se facessimo una semplice sostituzione si otterrebbe return x+(x+1), ovvero 2+2+1 quindi 5. Ma così non è, il risultato corretto è 3. Questo tipo di sostituzione viene chiamato **senza cattura**. Si può intendere il passaggio per nome come il passaggio del parametro formale con il rispettivo ambiente, infatti si richiede che il parametro formale, anche dopo la sostituzione, venga valutato nell'ambiente del chiamante e non in quello del chiamato.

Osservazione 7. Il parametro attuale viene valutato ogni volta che il formale viene incontrato durante l'esecuzione, con le relative conseguenze in presenza di eventuale effetti collaterali. Ad esempio:

```
int i=2;
int fie (name int y){
    return y+y;
}
int a=fie(i++);
```

dà come risultato a=5 e i=4, in quanto avvengono i seguenti passaggi:
Si assegna i, quindi il primo y=2
Si incrementa i, quindi i=3
Si assegna i, quindi il secondo y=3
Si incrementa i, quindi i=4
Sommo, quindi y=5

Definizione 3: *Si chiama **chiusura** la coppia (espressione, ambiente) nella quale l'ambiente comprende (almeno) tutte le variabili libere presenti nell'espressione.*

Riassumiamo quindi quello che sappiamo sul passaggio per nome:

- È una modalità che corrisponde ad un parametro sia di ingresso che di uscita.
- Il parametro formale non corrisponde ad una variabile locale alla procedura
- Il parametro attuale può essere una generica espressione, ma deve valutare ad un l-valore se il formale compare a sinistra di un assegnamento.
- Semantica stabilita dalla regola di copia
- Implementazione canonica fornita dalla chiusura.

- Ogni accesso al formale viene risolto mediante una valutazione ex novo del parametro attuale nell'ambiente fornito dalla chiusura.
- Modalità di passaggio molto costosa, sia per la necessità di passare una struttura complessa, sia per la valutazione ripetuta del parametro attuale in un ambiente diverso da quello corrente.

2. Funzioni di ordine superiore

Definizione 4: *Una funzione viene detta di ordine superiore se ha come parametro, o restituisce come risultato un'altra funzione.*

1. Funzioni come parametro

Il problema di passare una funzione come parametro è che si viene a creare un nuovo ambiente. Per definire questo ambiente si usano due politiche:

1. **Deep binding:** si usa l'ambiente attivo al momento della creazione del legame tra la funzione passata come parametro attuale e quella formale.
2. **Shallow binding:** si usa l'ambiente attivo al momento della chiamata della funzione.

A prima vista sembra che tra shallow e deep binding ci sia la stessa divisione che c'è per scope statico e dinamico. Tuttavia non è del tutto così, infatti se lo scope statico può adottare solo una politica di deep binding, non è lo stesso per quanto riguarda lo scope dinamico che può scegliere.

2. Implementazione shallow binding

Non pone ulteriori problemi implementativi rispetto alle tecniche utilizzate per lo scope dinamico: basta ricercare per ogni nome la sua ultima associazione presente.
Mentre si è appena detto che lo scope statico usa solo deep binding.

3. Implementazione deep binding

Richiede strutture dati ausiliarie rispetto all'ordinaria catena statica o dinamica.
Prendiamo il caso di scope statico: sappiamo già che quando viene chiamata una certa funzione f, viene associato un valore che indica la profondità di annidamento di tale funzione. Tuttavia nel caso in cui la funzione venga invocata tramite un parametro formale (h), alla chiamata non può essere associata alcuna informazione, perché il parametro formale può essere associato a diverse funzioni in base all'attivazione della procedura in cui si trova. È quindi evidente che con deep binding l'informazione relativa al puntatore di catena statica deve essere determinata al momento dell'associazione tra il parametro formale e il parametro attuale. Al formale h deve essere associato non solo il codice di f, ma anche l'ambiente non locale nel quale il corpo di f deve essere valutato.
Al parametro formale funzionale è associata una chiusura: al momento in cui il formale è usato per invocare una funzione, la macchina astratta trova il codice a cui trasferire il controllo nella prima componente della chiusura, e assegna il contenuto della seconda componete al puntatore di catena statica del record di attivazione della nuova invocazione.
Questo problema però si risolve in linguaggi come il C in cui non c'è ambiente non locale e quindi non c'è questo problema.
Per quanto riguarda lo scope dinamico, anche in questo caso si adottano le chiusure.

4. Politiche di binding e scope statico

A prima vista sembrerebbe che deep binding e shallow non facciano molta differenza e che per determinare gli ambienti bastino le regole di scope. Tuttavia questo non è vero nel caso in cui su una pila ci siano più RdA corrispondenti alla stessa funzione, come ad esempio le funzioni ricorsive.
Prendiamo ad esempio il seguente codice:

```
{
      void foo (int f (), int x){
            int fie (){
                  return x;
```

```
                }
                int z;
                if (x==0)
                      z==f();
                else
                      foo (fie, 0);
          }
          int g(){
                return 1;
          }
          foo (g, 1);
    }
```

Il problema sta nella x all'interno di fie. Le regole di scope infatti dicono che l'associazione della x è da trovarsi tra i parametri di foo, ma quando si cerca il legame della x ci sono due istanze di foo. Perciò le regole di scope non sono sufficienti a individuare l'associazione corretta.
Se venisse usato deep binding, allora l'ambiente sarebbe stabilito al momento della creazione dell'associazione tra fie e f, cioè mentre x è associato al valore 1.
In caso invece di shallow binding, l'ambiente sarebbe stato determinato al momento dell'invocazione di f: a z sarebbe stato assegnato lo 0.
Si può quindi concludere l'elenco degli oggetti che determinano un ambiente:

1. Regole di visibilità
2. Eccezioni alle regole di visibilità
3. Regole di scope
4. Regole relative alle modalità di passaggio dei parametri
5. Politica di binding

5. Funzioni come risultato

La funzione restituita come risultato non potrà essere rappresentata a tempo d'esecuzione dal suo solo codice, ma sarà necessario anche l'ambiente nel quale tale funzione dovrà essere valutata.
Pertanto quando una funzione restituisce un'altra funzione, tale risultato è una chiusura.
Tuttavia, affinché possa essere mantenuto tale risultato, il RdA che viene inserito non potrà poi essere rimosso dalla pila, perdendo quindi, in teoria, il significato stesso di pila.
La soluzione più comune è quella di allocare tutti gli RdA sullo heap e lasciare a un garbage collector il lavoro sporco.

3. Eccezioni

Definizione 5: *Un'eccezione è un evento particolare che si verifica durante l'esecuzione di un programma e che non deve (o non può) essere gestito dal normale flusso del controllo.*

Per gestire correttamente le eccezioni un linguaggio deve, almeno:

1. Quali eccezioni sono gestibili e come possono essere definite; alcuni linguaggi permettono eccezioni definite anche dall'utente, oltre che dalla macchina astratta.
2. Specificare come un'eccezione può essere sollevata, cioè con quali meccanismi si provoca la terminazione eccezionale di una computazione; ad esempio, essa può venire generata implicitamente se si tratta di un'eccezione della macchina astratta, oppure esplicitamente dal programmatore, con un apposito costrutto.
3. Specificare come un'eccezione possa essere gestita, cioè quali sono le azioni da compiere al verificarsi di un'eccezione e dove deve riprendere l'esecuzione del programma. Solitamente i costrutti usati sono due:
    a. Un meccanismo per definire una capsula attorno ad una porzione di codice con lo scopo di intercettare le eccezioni che dovessero verificare all'interno della capsula stessa (il try-catch di Java).

b. La definizione di un gestore dell'eccezione, in genere legato staticamente ad un blocco protetto al quale trasferire il controllo quando la capsula intercetta un'eccezione (ad esempio implementando una classe come Throwable in Java).

Per quanto queste questioni siano fortemente dipendenti dal linguaggio, osserviamo due aspetti importanti:

1. Un'eccezione non è un evento anonimo: ha un nome che viene esplicitamente menzionato nel costrutto throw e che viene usato dai costrutti di tipo try-catch per intrappolare una specifica classi di eccezioni
2. L'eccezione potrebbe inglobare un valore, che il costrutto che solleva l'eccezione passa in tal modo al gestore come argomento su cui operare per "reagire" all'avvenuta eccezione.

Se non viene creato alcun gestore per tale eccezione, allora l'eccezione risalirà tutta la catena di chiamate fintanto che non possa incontrare un gestore che la soddisfi, o fintanto da arrivare al livello più alto che fornirà un gestore di default.
Si presti molta attenzione che le eccezioni si propagano lunga la catena dinamica anche se il gestore è associato staticamente al blocco protetto. Ossia, un'eccezione viene gestita dall'ultimo gestore che sia stato posto sulla pila d'esecuzione, così da intrappolare l'eccezione il più vicino possibile a dove si è verificata.

Implementare le eccezioni

Il modo più semplice per gestire le eccezioni da parte della macchina astratta è quello di sfruttare la pila degli RdA. Ogni volta che si entra in un blocco protetto, viene inserito un puntatore al gestore corrispondente. Quando si esce da un blocco, senza generare errore, viene tolto dalla pila il riferimento al gestore. Se invece viene sollevata un'eccezione, la macchina cerca un gestore in quel record di attivazione, se non lo trova usa le informazioni del record di attivazione per ripristinare lo stato della macchina, toglie il record dalla pila e risolleva l'eccezione.
Sebbene questa sia una soluzione concettualmente semplice, non lo è dal punto di vista dell'esecuzione in quanto la macchina deve lavorare sulla pila ogni qual volta entri od esca da un blocco.
Una soluzione migliore a tempo d'esecuzione è la seguente: per prima cosa ad ogni procedura viene associato un blocco protetto nascosto costituito da tutto il corpo della procedura e il cui gestore nascosto è responsabile solo del ripristino dello stato e del risollevamento dell'eccezione. Il compilatore prepara poi una tabella EH in cui, per ogni blocco protetto, inserisce due indirizzo (ip e ig) che corrispondono all'inizio del blocco protetto e all'inizio del corrispondente gestore.
In questo modo quando si entra e si esce da un blocco protetto non si deve far niente, mentre quando viene sollevata un'eccezione, si cerca nella tabella una coppia di indirizzi tali che ip + il massimo presente in Eh e per cui ip<pc<ig.

Capitolo 10

Il sistema dei tipi di un linguaggio è fondamentale. I linguaggi di alto livello mettono a disposizione algoritmi e strutture dati per creare dei nuovi tipi di dato.

1. I tipi di dato

Definizione 1: ***Tipo di dato*** *Un tipo di dato è una collezione di valori omogenei ed effettivamente presentati, dotata di un insieme di operazioni che manipolano tali valori.*

Il termine omogenei suggerisce che i valori debbano condividere qualche proprietà. Tali valori inoltre vengono presentati insieme alle operazioni che permettono di manipolarli.
Con "effettivamente presentati" si intendono i valori che possano essere rappresentati. Ad esempio non tutti i numeri reali sono rappresentabili in quanto infiniti. Di conseguenza quello che viene presentato da un linguaggio di programmazione è solo un loro sottoinsieme che contiene le varie approssimazioni.
I tipi di dato sono fondamentali per 3 motivi:

5. Supporto all'organizzazione concettuale (una parola è diversa da un numero). Spesso questo si rispecchia soprattutto quando i tipi sono quelli definiti dall'utente, che definisce dei tipi in base al loro "scopo". Per esempio nella lettura di un programma per la gestione di un hotel, apparirà subito che il tipo "camera" è diverso dal tipo "prezzo".
6. Supporto alla correttezza di un programma (non si può fare la radice quadrata di una parola). Un esempio fondamentale è quello dell'assegnamento: affinché il comando x=exp sia corretto, la maggior parte dei linguaggi richiede che x e exp siano compatibili[12]. Questi vincoli sono fondamentali per evitare errori hardware a runtime, e errori logici che violano la regola di tipo.
   Il punto cruciale è che molti linguaggi verificano che i vincoli di tipo siano tutti soddisfatti prima di procedere all'esecuzione del programma. Questo ruolo è svolto dal type checker (controllore dei tipi) durante il controllo (da parte del compilatore) della semantica statica.
   Sebbene queste regole siano efficaci, possono risultare al tempo stesso minimali ed eccessive. Minimali perché non implicano che un programma corretto dal punto di vista dei tipi sia logicamente corretto, eccessive perché un programmatore abituato a lavorare direttamente sui puntatori potrebbe ritenere restrittive politiche tipo quella di Java sui puntatori.
   Sono poi presenti regole di tipo più sofisticate, che permettono ad esempio di scrivere funzioni parametriche nel tipo.
   I linguaggi di programmazione pertanto sono classificati tra sicuri e insicuri rispetto ai tipi, proprio in base alla possibilità che vi possano essere durante l'esecuzione delle violazioni dei vincoli di tipo non rilevate dalla macchina astratta.
7. Supporto all'implementazione durante la traduzione: forniscono importanti informazioni alla macchina astratta. Per prima cosa la dimensione che devono occupare in memoria. Questa informazione, per quanto riguarda i tipi, è disponibile staticamente e non cambia durante l'esecuzione. Questo permette un accesso alla memoria migliore, in quanto permette un offset (rispetto al puntatore di RdA) diretto, senza bisogno di alcuna ricerca a tempo d'esecuzione.

2. Sistema di tipi

[12] Spesso si richiede che siano proprio dello stesso tipo.

Un sistema di tipi è costituito da:

8. Insieme dei tipi predefiniti dal linguaggio;
9. I meccanismi che permettono di definire nuovi tipi;
10. Meccanismi relativi al controllo dei tipi, tra i quali distingueremo:
    - Le regole di equivalenza, che specificano quando due tipi formalmente diversi corrispondono allo stesso tipo;
    - Le regole di compatibilità, che specificano quando un valore di un certo tipo può essere utilizzato in un contesto nel quale sarebbe richiesto un tipo diverso;
    - Le regole e le tecniche di inferenza dei tipi, che specificano come il linguaggio attribuisce un tipo ad un'espressione complessa, partendo dalle informazioni delle sue componenti;
11. La specifica se i vincoli siano da controllare staticamente o dinamicamente.

Si dice che un sistema di tipi è type safe (sicuro rispetto ai tipi) quando nessun programma può violare le distinzioni tra i tipi definite da quel linguaggio.
Fissato un linguaggio di programmazione, possiamo classificare i suoi tipi a seconda di come i suoi valori possono venir manipolati e del genere di entità sintattiche che corrispondono a tali valori. Si possono avere valori:

- denotabili, se possono essere associati ad un nome;
- esprimibili, se possono essere il risultato di un 'espressione complessa (cioè diversa da un semplice nome);
- memorizzabili, se possono essere memorizzati in una variabile.

1. Controlli statici e dinamici

Si dice che un linguaggio ha **tipizzazione statica** se i controlli dei vincoli possono essere condotti a tempo di compilazione. Ha **tipizzazione dinamica** se i controlli avvengono a tempo d'esecuzione.
Il controllo dinamico prevede che ogni oggetto abbia un descrittore a run-time che ne specifica il tipo. Questo tipo di controllo non è molto efficiente in quanto ogni volta che deve essere eseguita un'operazione viene fatto un controllo, e per di più l'errore è rilevato solo a tempo d'esecuzione quando ormai il programma potrebbe essere in esecuzione presso l'utente.
Con un controllo di tipo statico invece gli errori vengono rilevati a tempo di compilazione, permettendo al programmatore di correggerli. Inoltre a tempo d'esecuzione non sarà necessario mantenere dei descrittori che identifichino il tipo degli oggetti, permettendo un'esecuzione più efficiente. Gli svantaggi di questo tipo di controllo sono:

12. Progettazione di un linguaggio con controllo statico è più difficoltosa;
13. La compilazione è più complessa e lenta;
14. Possono essere riscontrati errori che in realtà non lo sono.

Non esiste un compilatore statico che possa riscontrare tutti e soli i programmi che possono generare errori a tempo d'esecuzione. Per questo motivo si tende ad assumere una posizione "prudenziale" nel senso che si escludo più programmi di quelli davvero necessari, in modo da garantire la correttezza. Come spesso accade però, non esiste solo controllo statico o solo controllo dinamico, ma è un'unione dei due.

3. Tipi scalari

Definizione 2: ***Tipi scalari***[13] *sono tutti quei tipi i cui valori non sono costituiti da aggregazioni di altri valori.*

[13] O semplici

Tipi scalari sono anche i tipi definiti dal programmatore, basta che non vengano definiti come aggregazioni di altri tipi. Per tanto la seguente sintassi in C crea un nuovo tipo scalare dal nome nuovotipo:

```
typedef espressione nuovotipo
```

Ad esempio

```
typedef unsigned long ulong
```

1. **Booleani**

Sono i valori logici, o booleani. Sono costituiti da:

15. Valori: i due valori di verità vero e falso;
16. Operazioni: operazioni logiche quali: and, or, not, uguaglianza, or esclusivo ecc.

Laddove presente i suoi valori sono memorizzabili, esprimibili e denotabili[14].

2. **Caratteri**

17. Valori: un insieme di codici di caratteri fissato al momento della definizione del linguaggio[15].
18. Operazioni: dipendono molto dal linguaggio. Si trova sempre l'uguaglianza e i confronti.

I valori sono memorizzabili, esprimibili e denotabili.

3. **Interi**

19. Valori: un sottoinsieme finito dei numeri interi. Solitamente l'intervallo viene fissato dal linguaggio, ma alcune volte è la macchina a deciderlo. L'intervallo è così definito: $[-2^t, 2^t-1]$;
20. Operazioni: sono sempre presenti i confronti e quasi tutte le operazioni aritmetiche.

I valori sono memorizzabili, esprimibili e denotabili.

4. **Reali**

21. Valori: un opportuno sottoinsieme dei numeri razionali, di norma fissato al momento della definizione del linguaggio;
22. Operazioni: sono sempre presenti i confronti e quasi tutte le operazioni numeriche;

I valori sono memorizzabili, definibili e denotabili. La rappresentazione in memoria avviene secondo lo standard IEEE 754.

5. **Virgola fissa**

23. Valori: un opportuno sottoinsieme dei numeri razionali, di norma fissato al momento della definizione del linguaggio;
24. Operazioni: sono sempre presenti i confronti e quasi tutte le operazioni numeriche;

I valori sono memorizzabili, esprimibili e denotabili. La rappresentazione in memoria avviene secondo la tecnica del complemento a due.

6. **Complessi**

25. Valori: un opportuno sottoinsieme dei numeri complessi, di norma fissato al momento della definizione del linguaggio;
26. Operazioni: sono sempre presenti i confronti e quasi tutte le operazioni numeriche;

I valori sono memorizzabili, esprimibili e denotabili. La rappresentazione in memoria consiste in una coppia di valori in virgola mobile.

7. **Void**

27. Valori: uno solo che possiamo indicare con ();
28. Operazioni: nessuna.

Viene usato per indicare il tipo delle operazioni che modificano lo stato ma non restituiscono alcun valore.

8. **Enumerazioni**

Può essere visto come un nuovo modo per definire tipi. Un tipo enumerazione consiste in un insieme finito di costanti, ciascuna caratterizzata dal proprio nome.

---

[14] Il C non possiede queste caratteristiche

[15] Solitamente ASCII o UNICODE

Le operazioni disponibili su un'enumerazione sono costituite dai confronti e da un meccanismo per passare da un valore al successivo e/o al precedente.
Un valore di un'enumerazione è in genere rappresentato con un intero su un byte; i diversi valori hanno rappresentazione contigua, partendo da zero.

9. Intervalli

I valori di un tipo intervallo costituiscono un sottinsieme contiguo dei valori di un altro tipo scalare. Si noti quindi che si può avere anche un intervallo di un'enumerazione.
La verifica che un valore di una certa espressione appartenga davvero all'intervallo non può che essere fatta in modo dinamico.
Il vantaggio di usare intervalli ed enumerazioni è quello di avere un controllo sui tipi più stringente, e al tempo stesso programmi più leggibili.

10. Tipi ordinali

I tipi booleani, caratteri, interi, enumerazioni e intervalli sono esempi di tipi ordinali perché sono dotati di una ben definita nozione di ordine totale (e quindi di predecessore e successore).

4. Tipi composti

I tipi non scalari sono detti composti, in quanto si ottengono combinando tra loro altri tipi mediante l'utilizzo di opportuni costruttori.

1. Record

Un record è una collezione costituita da un numero finito di elementi, detti campi, distinti dal loro nome. Ogni campo si comporta come una variabile del proprio tipo. La terminologia è varia: record, struct e classi (da Java, uso improprio).
Solitamente per indicare un campo del record si usa la notazione

```
nomeRecord.nomeCampo
```
[16]

Possono esistere anche record annidati, ovvero dei record all'interno di altri record.
Non sempre viene definita un'uguaglianza tra record, e nemmeno è sempre definito un assegnamento tra due record. Solitamente sta al programmatore creare dei metodi laddove mancano.
L'ordine dei campi viene solitamente rispettato nella rappresentazione in memoria. Si noti che tuttavia non sempre sono allocati in zone di memoria contigua: alcune volte per motivi di allineamento vengono lasciate delle zone vuote. Per questo alcuni compilatori possono non rispettare la rappresentazione in memoria per ottimizzare lo spazio usato.

2. Record varianti e unioni

Una forma particolare di record è quella in cui alcuni campi sono tra loro mutuamente esclusivi. Questa forma di record presenta molte varianti, e anche molte complicazioni. Uno dei linguaggi in cui la nozione di record variante è meglio presente è il Pascal. Ad esempio si può definire un record come:

```
type Stud = record
      nome : array [1...6] of char;
      matricola : integer;
      case fuoricorso : boolean of
            true : (ultimoanno : 2000..maxint);
            false:(
                  inpari : boolean;
                  anno : (primo, secondo, terzo)
            )
end;
```

[16] Sintassi presa soprattutto da C e di Java.

ogni record successivamente definito avrà di certo i campi nome e matricola, mentre i successivi campi dipendono dal tag fuoricorso. Il tag può essere un qualsiasi tipo ordinale, e per tanto può essere seguito da un numero di varianti pari alla cardinalità del tipo. Ciò che compare tra parentesi tonde, quindi (ultimoanno) e (inparti, anno) sono dette varianti.
Al tag e alle varianti si può accedere come per qualsiasi altro campo del record. Questo è solitamente garantito dal fatto che non possono essere contemporaneamente attive, e per lo stesso motivo generalmente condividono lo stesso spazio di memoria.

Unioni in C

In C non esiste il concetto primitivo di record variante, ma quello di tipo unione. La differenza è minima, ma risulta molto pesante. Prendiamo la seguente struttura di C, del tutto uguale a quella precedente:

```
struct Stud{
      char nome [6];
      int matricola;
      bool fuoricorso;
      union {
            int ultimoanno;
            struct {
                  int inpari;
                  int anno;
            } stud_in_corso;
      } campivarianti;
};
```

Praticamente in C quando si deve fare una scelta tra delle varianti, si crea un tipo unione con un nome (campivarianti) al cui interno vanno scritte le possibili varianti, in questo caso ultimoanno e una struttura di nome[17] stud_in_corso che contiene altri due campi. Si noti che l'utilizzo della struct stud_in_corso non è uno zuccherino sintattico, ma è necessario per dire che inpari e anno fanno parte della stessa variante. Per tanto quando vorremo riferirci a inpari si dovrà scrivere:

```
nomeStud.campovarianti.stud_in_corso.inpari
```

che come si può notare è ben più lungo della variante di Pascal

```
nomeStud.inpari
```

si può notare subito che in C il tag è del tutto svincolato dall'unione. Ciò può portare a degli errori che non vengono segnalati dal compilatore, ad esempio quando nonostante il tag sia falso e si accede alla variante sbagliata.

Varianti e sicurezza

Sembrerebbe quindi che la versione di Pascal sia da preferire a quella di C, in quanto più chiaro e più compatto. A una prima occhiata, per quanto riguarda il discorso del tag verrebbe da dire che sia anche più sicuro, tuttavia così non è, perché anche Pascal non riesce a garantire che vi sia un legame formale tra il valore del tag e la significatività di una delle varianti. La definizione del linguaggio infatti, non richiede nessun controllo sul tag quando si accede alle varianti, e tuttavia questa soluzione riuscirebbe a catturare solo alcuni errori semantici. Inoltre tale controllo avverrebbe dinamicamente, generando quindi errori a run-time.
Ci si potrebbe pertanto domandare se valga la pena avere un costrutto così "costoso" in termini di sicurezza. La risposta, nella prospettiva odierna, è negativa: il risparmio di memoria che i record varianti assicurano non è giustificato dai problemi che essi causano al sistema di tipi.

3. Array

Un array (o vettore) è una collezione finita di elementi dello stesso tipo, indicizzata su un intervallo di un tipo ordinale. Poiché tutti gli elementi sono dello stesso tipo, allora l'array è una struttura dati

[17] È necessario darle un nome alla fine così da poterla successivamente chiamare.

omogenea. Sebbene la sintassi vari molto da linguaggio a linguaggio, tuttavia ci sono degli elementi fondamentali quali il nome, il tipo (che è quello dei suoi elementi), e l'indice. Ad esempio in C:

```
int V [10]
```

indica un array di nome V di 10 elementi di tipo int. Molti linguaggi permettono di definire il numero degli elementi come un intervallo (anche di un'enumerazione), ad esempio:

```
int W [21...30];
type Nano = {Brontolo, Cucciolo, Dotto, Eolo, Gongolo, Mammolo,
     Pisolo);
float Z [Dotto...Mammolo];
```

vengono creati 2 array, il primo W di 10 elementi di tipo int, e il secondo di 4 elementi di tipo float. Per accedere quindi agli elementi potremmo scrivere:

```
V[0] //Per indicare il primo elemento dell'array V
V[9] //Per indicare l'ultimo elemento dell'array V
W[21] //Per indicare il primo elemento dell'array W
W[30] //Per indicare l'ultimo elemento dell'array W
Z[Dotto] //Per indicare il primo elemento dell'array Z
Z[Mammolo] //Per indicare l'ultimo elemento dell'array Z
```

Vi possono poi essere degli array multidimensionali, anche detti matrici.

```
int M [1...10, 1...10]
```

In questo modo si crea una tabella 10x10. Alcuni linguaggi permettono di definire gli array multidimensionali come array di array[18]. Sono ovviamente possibili anche gli altri metodi di dichiarazione per mezzo di intervalli.

```
int M [10][10]
```

Per l'accesso agli array multidimensionali sono validi:

```
M [i][j]
M [i, j]
```

A seconda che si enfatizzi l'aspetto "array di array" o multidimensionale.

Operazioni sugli array

Alcuni linguaggi mettono a disposizione operazioni definite sulla globalità degli array, quali assegnamento, uguaglianza, confronti e operazioni matematiche (eseguire elemento per elemento).
Alcuni linguaggi poi mettono a disposizione degli strumenti per ottenere degli slice di array, ovvero fette di elementi contigui.

Controlli

Quasi tutti i linguaggi verificano che l'accesso di un array avvenga entro i limiti dell'array, in base all'indice. Ad eccezione di alcuni casi particolari, tale controllo può avvenire solo a tempo d'esecuzione, e per tanto deve essere il compilatore a generare opportuni controlli in prossimità di ogni accesso che verranno poi eseguiti. Poiché questo può incidere sulla velocità di esecuzione, alcuni linguaggi permettono di disattivarli.
Questo tipo di rischio è molto elevato in quanto alcuni degli attacchi più comuni a un linguaggio avvengono per mezzo di un buffer overflow e conseguente possibilità di inserire il codice che si vuole.

Memorizzazione e calcolo degli indici

Un array è memorizzato di solito in una porzione contigua di memoria. Per un array monodimensionale, l'allocazione segue l'ordine degli indici. Nel caso di array multidimensionali si seguono due tecniche alternative, dette memorizzazione in ordine di riga e in ordine di colonna.
Le due modalità di memorizzazione non sono equivalenti quando si è in presenza di cache. Infatti nel caso che si lavori con array di grandi dimensioni, che non entrano a pieno in cache, allora conviene posizionare gli array secondo la cache in modo che il primo miss porti in cache gli elementi che saranno acceduti dopo. Se tale ciclo opera per righe, allora anche la cache dovrebbe

[18] Soluzione più comune

essere caricata per righe, e quindi una memorizzazione per righe conviene, se invece opera per colonne, allora conviene una memorizzazione per colonne.
Per ottenere l'indirizzo corrispondente a un generico elemento di un array, conoscendo la disposizione in memoria, e l'indirizzo base, il calcolo è molto facile. Prendiamo ad esempio il vettore;

```
int V[10][10]
```

memorizzato per righe, che ha come indirizzo base 0x2000 e vogliamo conoscere l'indirizzo dell'elemento V[1][5]. Il calcolo che si esegue è:

$$indirizzoFin = indirizzoBase + \Big(\big(i * dim_j *\big)j * dim\Big)_{16}$$

Dove la dim_j è il numero degli elementi per ogni riga (10) e dim è la dimensione degli elementi in byte.
Quindi si calcola

$$(i * dim_j) * f * dim = (1 * 10) * 5 * 1 = 50$$

Lo si trasforma in esadecimale:

$$50_{10} = 32_{16}$$

e lo si somma all'indirizzo base:

$$indirizzoFin = 0x2000 + 32 = 0x2032$$

Se invece l'array fosse stato memorizzato per colonne il calcolo da eseguire è lo stesso soltanto invertendo gli indici (sia di V[10][10] che di V[1][5]).

Forma di un array: dove viene allocato un array

La forma di un array è costituita dal numero delle sue dimensioni[19] e dall'intervallo su cui ciascuna di esse può variare.
Il momento in cui debba essere fissata la forma di un array non è fisso. Ci sono 3 tipologie in base al momento:

- Forma statica: in tal caso tutto è deciso a tempo di compilazione e l'array può essere memorizzato nel RdA del blocco in cui compare la sua definizione. In questo caso la dimensione dell'array deve essere costante e nota a tempo di compilazione. Per questo motivo l'accesso a un elemento dell'array è molto semplice in quanto anche l'offset sarà costante.
- Forma fissata al momento dell'elaborazione della dichiarazione: è il caso in cui non si è in grado di definire la forma a tempo di compilazione, ma questa è nota al momento in cui il controllo raggiunge la dichiarazione dell'array[20]. Anche in questo caso l'array può essere memorizzato nell'RdA del blocco in cui compare la sua definizione, ma possono sorgere dei problemi con l'offset, che possono poi ripercuotersi sulle strutture statiche circostanti. Per ovviare a questi problemi si divide l'array in due parti, una parte di lunghezza costante e una parte di lunghezza variabile. A quella di lunghezza costante si accede per offset diretto, mentre a quella di lunghezza variabile si accede per offset indiretto per mezzo di un descrittore.
- Forma dinamica: in questo caso un array può cambiare forma dopo la sua creazione e di conseguenza l'allocazione sulla pila non è più sufficiente. Vengono pertanto allocati sullo heap, mentre un puntatore all'inizio dell'array rimane memorizzato nell'RdA.

Un altro fattore da tenere in considerazione è la lunghezza di vita di un array. Quelli messi sull'RdA di un blocco hanno un tempo di vita che è appunto quello del blocco. In Java però possono esserci anche array statici con tempo di vita illimitato e per tanto vengono comunque posti sullo heap.

Dope vector

È il descrittore di un array di forma non nota staticamente. Questo contiene:

[19] Quindi se è multidimensionale e a quante dimensioni.
[20] Ad esempio l'intervallo dell'indice dipende dal valore di una variabile.

- Un puntatore alla prima locazione in cui è memorizzato l'array
- Tutte le informazioni dinamiche come il numero di dimensioni, l'occupazione di ogni dimensione.

Qualora una di queste quantità fosse staticamente nota, essa non viene memorizzata nel dope vector.
Per accedere a un elemento dell'array, si calcola l'offset necessario a raggiungere il dope vector, quindi si calcola l'indirizzo in base ai dati contenuti in questo.

4. Insiemi

Il tipo insieme è costituito da sottoinsiemi di tipi base (o universo). Prendendo la sintassi di Pascal possiamo scrivere:

```
set of char S;
set of Nano N
```

per avere un sottoinsieme di caratteri in S e un sottoinsieme del tipo Nano in N. Ovviamente ogni linguaggio che permette la dichiarazione di sottoinsiemi permette anche l'assegnazione, in Pascal:

```
N=(Eolo, Brontolo);
```

Le operazioni possibili sono quelle per l'appartenenza, l'unione, l'intersezione e la differenza.
Un insieme è in genere rappresentato come un vettore di bit di lunghezza pari alla cardinalità del tipo base. Per questo motivo molti linguaggi limitano i tipi di base che possono dare insiemi, oppure usano delle rappresentazioni diverse, soprattutto per mezzo di tabelle hash.

5. Puntatori

Alcuni linguaggi permettono di manipolare direttamente l-valori: il tipo corrispondente è detto tipo dei puntatori. Solitamente esiste un tipo puntatore per ogni tipo del linguaggio.
In ogni linguaggio una variabile ha implicitamente un l-valore che viene dereferenziato automaticamente per ottenere l'r-valore. Alcuni linguaggi sfruttano appunto i puntatori per permettere di riferirsi a un l-valore senza dereferenziarlo.
L'uso principale dei puntatori è quello di costruire valori di tipi ricorsivi.
Alcuni linguaggi quali il Pascal richiedono che i puntatori accedano solo a oggetti allocati sullo heap. Altri invece ammettono che un puntatore possa puntare anche a locazioni sulla pila di sistema o nell'area globale.
Tra tutti i valori che un puntatore può assumere ne esiste uno canonico che indica quando tale puntatore non punta a nulla, e si chiama appunto null.
Le operazioni permesse sui puntatori sono quelle di uguaglianza, creazione e dereferenziazione.
Il modo più comune per creare un valore di tipo puntatore è quello di usare un costrutto predefinito che alloca sullo heap un oggetto e restituisce un puntatore a tale oggetto. Questo è ad esempio il caso di malloc in C:

```
int* p;
p (int*) malloc (sizeof (int));
```

o di new in Pascal. In C è poi presente anche un altro operatore: &, che permette di ottenere l'indirizzo di un certo oggetto. Ad esempio

```
int r=131;
int* q;
q=&r;
```

q sarà un puntatore a r, il quale è stato allocato sulla pila, quindi q sarà un puntatore alla pila e non allo heap.
* oltre a indicare un puntatore, viene anche usato in C come un operatore di dereferenziazione. Riprendendo l'esempio di prima, se si volesse ottenere il valore 131 da q e sommare 1 per poi salvarlo nuovamente in q bisognerebbe scrivere:

```
*q=*q+1;
```

Poiché q punta a r, una volta modificato il valore di q, viene modificato anche il valore di r, quindi al termine dell'esecuzione dell'operazione precedente, sia q che r avranno valore 132.

Aritmetica dei puntatori
oltre alle usuali operazioni sui puntatori viste in precedenza, in C e nei suoi successori sono possibili anche operazioni aritmetiche. Questo permette ad esempio di incrementare o decrementare un puntatore, finendo in una nuova area di memoria. Ciò torna molto utile nel caso in cui si abbia un puntatore a un array dinamico, infatti incrementando o decrementando nel modo corretto si può arrivare a qualsiasi elemento dell'array.
Si osservi che questo tipo di aritmetica distrugge qualsiasi possibile sicurezza di tipi.
Deallocazione
La deallocazione della memoria può essere sia implicita che esplicita.
Nel primo caso il linguaggio non mette a disposizione nessun tipo di operazioni per deallocare la memoria. Quando la memoria sullo heap termina, allora si controlla se alcuni valori precedentemente allocati sono ancora utilizzati o meno. Se non sono più utilizzati allora vengono deallocati. Il meccanismo che prevede il recupero di porzioni di memoria viene detto garbage collection e sono un aspetto molto importante delle macchine astratte.
Nel caso di deallocazione esplicita, il linguaggio mette a disposizione un meccanismo che permette di liberare la memoria riferita a un puntatore. Ad esempio in C se si alloca un puntatore p con malloc, si può successivamente liberare lo spazio usato da p con la funzione free(p). Però qualsiasi chiamata a p risulterà in un errore (a meno che non venga nuovamente allocato).
Uno dei problemi della deallocazione esplicita è il cosiddetto fenomeno dei dangling reference (riferimenti pendenti) cioè puntatori con un valore diverso da null, che però puntano a informazioni non più significative. Per esempio definendo due puntatori p e q, dove q punta a p, e successivamente si dealloca p, se si chiama p la macchina astratta genera errore, ma se si chiama q la macchina astratta potrebbe non rilevare alcun errore.
I dangling references sono una minaccia per la type safness. Ci sono però delle tecniche che permettono di identificarli e neutralizzare i pericoli.

6. Tipi ricorsivi

Un tipo ricorsivo è un tipo composto nel quale un valore del tipo può contenere un riferimento a un valore dello stesso tipo.
In alcuni linguaggi sono definiti come record al cui interno hanno una chiamata a sé stessi.
L'esempio più semplice è probabilmente quello di una lista di interi[21]:

```
type int_list={
      int val;
      int_list next
};
```

Per terminare la ricorsione, il linguaggio può fornire un valore speciale, solitamente null.
Un valore di tipo int_list quindi è coppia di cui il primo elemento è un intero, mentre il secondo è di nuovo un int_list:

```
{2, {33, {1, {4, null}}}};
```

Le operazioni permesse sui valori di tipo ricorsivo sono la selezione di una componente e il test di uguaglianza sul valore comune null.
In un linguaggio imperativo che ammette variabili modificabili, i tipi ricorsivi possono assumere anche una forma ricorsiva, mentre in un linguaggio funzionale hanno sempre una forma ad albero.
I tipi ricorsivi sono rappresentati di solito come strutture dati sullo heap, come una struttura concatenata: ogni elemento della struttura è costituito da un record, nel quale il riferimento ricorsivo è implementato come un indirizzo al prossimo record.
In molti linguaggi di tipo funzionale i valori di tipo ricorsivo sono esprimibili. Nel linguaggi imperativi invece il tipo ricorsivo vengono costruiti mediante allocazione esplicita dei loro componenti sullo heap.

[21] Adattato da Java.

7. Funzioni

Tutti i linguaggi di alto livello permettono di definire funzioni.
I valori di una funzione sono sempre denotabili, ma raramente sono esprimibili o memorizzabili.
Le operazioni principali sono la definizione, l'applicazione cioè l'invocazione di una funzione su alcuni argomenti (chiamati parametri attuali, mentre quelli della definizione sono i parametri formali).
In alcuni linguaggi è possibile passe funzioni come argomento di altre funzioni, oppure si possono avere funzioni che ritornano funzioni. Tali linguaggi sono solitamente quelli che appartengono al paradigma funzionale.

5. Equivalenza

Due tipi, formalmente diversi, possono considerarsi uguali all'interno di un linguaggio, quando vengono rispettate alcune regole che definiscono una relazione di equivalenza tra tipi. Se due tipi sono equivalenti allora ogni espressione o valore di un tipo è anche espressione o valore dell'altro, e viceversa.
Presa la definizione di un nuovo tipo come:

```
type nuovoTipo = espressione;
```

i vari linguaggi interpretano tale definizione in due regole diverse:

- Equivalenza per nome: quando la definizione di tipo è opaca
- Equivalenza strutturale: quando la definizione di tipo è trasparente.

1. Equivalenza per nome

Se un linguaggio adotta definizioni opache, allora ogni nuova definizione introduce un nuovo tipo diverso da quelli precedenti.

Definizione 3: ***Equivalenza per nome*** *Due tipi sono equivalenti per nome solo se hanno lo steso nome*

Questa definizione implica quindi che un tipo è equivalente solo a sé stesso.
Alcune volte questo tipo di equivalenza è troppo vincolante, ad esempio Pascal adotta un'equivalenza per nome debole, che prevede che una semplice ridenominazione di tipo sia equivalente al tipo.
Il vantaggio di questo tipo di regola è principalmente una maggior facilità di manutenzione del codice, dovuta al fatto che ogni tipo è diverso.

2. Equivalenza strutturale

Una definizione è di tipo trasparente quando il nome del tipo non è che un'abbreviazione del tipo che viene definito.
In un linguaggio con questo tipo di definizione si ha che due tipi sono equivalenti se hanno la stessa struttura, ossia se sostituendo tutti i nomi con le relative definizioni, si ottengono tipi identici.

Definizione 4: ***Equivalenza strutturale*** *L'equivalenza strutturale fra tipi è la (minima) relazione d'equivalenza che soddisfa le seguenti proprietà:*

- Un nome di tipo è equivalente a sé stesso;
- Se un tipo T è introdotto con una definizione type T=espressione, allora T è equivalente all'espressione;
- Se due tipi sono costruiti applicando lo stesso costruttore di tipo a tipi equivalenti, allora essi sono equivalenti.

Con questa regola due tipi equivalenti possono sempre essere sostituiti uno al posto dell'altro.
Come accade sempre, la maggior parte non adotta in modo puro nessuna delle due regole, ma usano una qualche combinazione delle due.

6. Compatibilità e conversione

La relazione di compatibilità tra tipi permette di usare un tipo in un contesto in cui sarebbe richiesto un altro tipo. La condizione affinché ciò si avveri è ovviamente più debole dell'equivalenza.

Definizione 5: ***Compatibilità*** *Diciamo che il tipo T è compatibile con il tipo S se un valore di tipo T è ammesso in un qualsiasi contesto in cui sarebbe richiesto un valore di tipo S.*

Solitamente nei linguaggi, è la compatibilità e non l'equivalenza a valutare la correttezza di un assegnamento o del passaggio di parametri a una funzione.
Ovviamente due tipi equivalenti sono anche compatibili.
Attenzione che non sempre se T è compatibile ad S, allora anche S è compatibile a T.
La relazione di compatibilità varia molto da linguaggio a linguaggio. Alcune linee comuni possono essere le seguenti, T è compatibile ad S:

- T e S sono equivalenti;
- I valori di T sono un sottoinsieme dei valori di S;
- Tutte le operazioni sui valori di S sono possibili anche sui valori di T;
- I valori di T corrispondono in modo canonico ad alcuni valori di S;
- I valori di T possono essere fatti corrispondere ad alcuni valori di S. Ovvero due tipi possono essere resi compatibili qualora si crei un metodo per trasformare un valore di T in uno di S.

Esistono due tipi di conversioni principalmente:

- **Conversione implicita**: detta anche coercizione, avviene quando è la macchina astratta a fare tale conversione automaticamente.
- **Conversione esplicita**: detta anche cast, è quando la conversione avviene perché indicata dal cast.

Coercizioni
Quando due tipi T ed S sono compatibili, e quindi un valore può essere rappresentato sia come T che come S, se necessario la macchina può inserire una conversione implicita da T ad S. Se da un punto di vista sintattico il suo scopo è solo quello di annotare una compatibilità, dal punto di vista pratico può corrispondere a cose distinte:

- T è compatibile con S e condividono la rappresentazione in memoria.
- T è compatibile con S perché esiste un modo canonico per trasformare i valori di T in S. In questo caso viene inserito automaticamente del codice per tale trasformazione.
- T è compatibile con S perché esiste un metodo che rende i due tipi compatibili. Anche in questo caso è la macchina che inserisce automaticamente il codice.

Linguaggi con forti controlli di tipo tendono ad avere poche coercizioni, mentre linguaggi come C ne presentano molte.
Conversioni esplicite
Le conversioni esplicite sono annotazioni nel linguaggio che specificano che un valore di un tipo deve essere convertito in uno di un altro tipo.
Non tutte le conversioni esplicite sono consentite, ma solo quelle per le quali il linguaggio conosce come avvengono. Laddove esiste una compatibilità è sempre possibile inserire un cast.
In linea generale si tende a preferire i cast alle coercizioni:

- Sono più espressivi
- Non dipendono dal contesto sintattico in cui compaiono
- Si comportano meglio in presenza di overloading e polimorfismo.

7. Polimorfismo

Un sistema di tipi nel quale ogni oggetto ha un unico tipo, si dice monomorfo.

Definizione 6: ***Polimorfismo*** *Un sistema di tipi in cui uno stesso oggetto può avere più di un tipo è detto polimorfo.*

Solitamente nei linguaggi tradizionali non è consentito definire al programmatore oggetti polimorfi. Da questo derivano più tipo di polimorfismo:

- Overloading
- Polimorfismo universale, che si divide a sua volta in:
  - Parametrico;
  - Di sottotipo, o di inclusione.

1. Overloading

Si tratta di polimorfismo solo in apparenza. Un nome si dice overloaded (sovraccaricato) quando ad esso corrispondono più oggetti, e per decidere quale usare viene analizzato a livello statico il contesto in cui si trovano i nomi. Quindi è come se a livello di compilazione ogni oggetto ottenesse un nome diverso.

2. Polimorfismo universale parametrico

Definizione 7: ***Polimorfismo universale parametrico*** *Un valore esibisce polimorfismo universale parametrico quando ha un'infinità di tipi diversi, che si ottengono per istanziazione da un unico schema di tipo generale.*

Una funzione polimorfa universale è dunque costituita da un unico codice, che lavora uniformemente su tutte le istanze del suo tipo generale. Un esempio è il valore null, che è lo stesso per ogni tipo di puntatore.

La notazione prevede di usare le parentesi angolate per indicare un tipo generico, ad esempio

<T> -> void

indica una funzione che preso un qualsiasi tipo ritorna un tipo void.

Questo polimorfismo è molto generale e diffuso, per questo motivo è presente nei linguaggi con due varianti:

- Polimorfismo esplicito: nel programma sono presenti esplicite annotazioni che indicano quali tipi devono essere considerati parametri.
- Polimorfismo implicito: il programma può non riportare alcuna annotazione sui tipi da usare, ed è il controllore dei tipi a cercare di ottenere, per ogni oggetto, il suo tipo generale da cui si possono ottenere tutti gli altri tipi per istanziazione dei parametri di tipo.

3. Polimorfismo universale di sottotipo

È presente principalmente nei linguaggi orientati ad oggetti. È una forma più limitata di quello visto prima. Anche in questo caso uno stesso oggetto ha un'infinità di tipi diversi, e anche in questo caso si ha un solo algoritmo che può essere istanziato, tuttavia, non tutte le possibili istanziazioni dello schema di tipo più generale sono ammissibili, ma solo quelle definite da una qualche nozione di compatibilità strutturale tra i tipi, quindi dalla nozione di sottotipo.

Definizione 8: ***Polimorfismo di sottotipo*** *Un valore esibisce polimorfismo di sottotipo quando ha un'infinità di tipi diversi, che si ottengono per istanziazione da uno schema di tipo generale, sostituendo ad un opportuno parametro i sottotipi di un tipo assegnato.*

4. Cenni sull'implementazione

Un primo modo per gestire il polimorfismo è quello di risolverlo in modo statico a tempo di linking. Quando una funzione polimorfa è chiamata con due istanze diverse, si genera del codice diverso per le due chiamate, in modo da poter allocare la giusta quantità di memoria per i valori.

Altri linguaggi, come ML usano un'unica rappresentazione della funzione polimorfa. Ma come si può fare quindi quando si deve basarsi sul tipo? La soluzione è quella di cambiare alla radice la rappresentazione del tipo di dato: al posto di allocarlo direttamente nell'RdA, si mette un puntatore al dato stesso, che comprende anche un suo descrittore. La memoria necessaria sarà poi allocata

sullo heap. Questa scelta pesa un po' sull'efficienza in quanto occorre sempre un accesso indiretto per riferirsi al tipo.

8. Controllo e inferenza di tipo

Per determinare il tipo delle espressioni complesse, il controllore esegue una semplice visita dell'albero sintattico del programma: a partire delle foglie risale l'albero verso la radice, calcolando i tipo in base dalle informazioni fornite dal programmatore e da quelle che derivano dal sistema di tipi.
Alcuni costrutti possono risultare ridondanti, ma sono necessari al fine di evitare errori logici. Per esempio nella funzione

```
int succ (int n)
      return (n+1);
```

è superfluo specificare il tipo della funzione in quanto essendo n un int, di sicuro ciò che si ritorna sarà un intero.
Alcuni linguaggi adottano un controllo più sofisticato ce viene chiamato **inferenza di tipo**. Il capostipite è ML, il cui sistema di tipi è molto sofisticato. Prendiamo la funzione[22]:

```
fun succ(n)
      return (n+1);
```

per ottenere il tipo di questa funzione si lavora sull'albero sintattico: può capitare che non sia possibile assegnare immediatamente un tipo, in questo caso si usa un tipo generico 'a secondo l'uso di ML. Risalendo poi l'albero e sfruttando le informazioni di tipo presenti nel contesto, vengono raccolti dei vincoli sulle variabili di tipo.
Questo tipo di inferenza è molto più generale e potente del semplice controllo dei tipi usato in altri linguaggi quali C e Java. Questa procede secondo il seguente algoritmo:

1. Assegna un tipo ad ogni nodo dell'albero sintattico;
2. Risale l'albero sintattico, generando un vincolo di uguaglianza tra tipi in corrispondenza ad ogni nodo interno.
3. Risolve i vincoli così raccolti usando l'algoritmo di unificazione.

9. Sicurezza: un bilancio

Si possono classificare i linguaggi in base alla loro sicurezza dei tipi:

- Linguaggi non sicuri: sono quei linguaggi in cui il sistema dei tipi è un suggerimento metodologico, nel senso che il linguaggio stesso permette di aggirare questo sistema. Sono dunque non sicuri linguaggi come C e C++, e discendenti.
- Linguaggi localmente sicuri: sono quei linguaggi nei quali il sistema di tipi è ben regolato e i tipi controllati, ma che contengono anche limitati costrutti che permettono di scrivere programmi non sicuri. A questi appartengono linguaggi quali il Pascal, ALGOL, e discendenti. La locale non sicurezza discende dalla presenza di unioni e dalla deallocazione esplicita di memoria.
- Linguaggi sicuri: sono quei linguaggi nei quali è un vero e proprio teorema a garantire che l'esecuzione di un programma tipizzato non possa generare errori relativi alla violazione di tipo. In questa categoria sono presenti numerosi linguaggi funzionali quali ML, Scheme, LISP, Java.

[22] Nota che non è propria di ML. Sarebbe `fun succ n = n+1;`.

10. Evitare i dangling reference

Esistono vari meccanismi che possono essere inclusi in una macchina astratta per impedire dinamicamente la dereferenziazione di dangling reference

1. Tombstone

Grazie a questo metodo una macchina astratta è in grado di segnalare ogni dangling reference: ogni volta che viene allocato un oggetto sullo heap a cui si accede per puntatore, oppure che viene creato un puntatore alla pila, la macchina astratta alloca anche un'ulteriore parola di memoria (tombstone).
Il percorso che poi la macchina fa quando si dereferenzia un oggetto è questo: puntatore -> tombstone -> oggetto. Quindi è la tombstone che ha il vero indirizzo dell'oggetto, mentre il puntatore rimanda all'indirizzo della tombstone per il controllo sull'integrità.
Questo controllo è possibile perché la tombstone, oltre al percorso dell'oggetto ha un numero di riferimento. Questo numero di riferimento permette di verificare se l'area di memoria in cui è allocato l'oggetto è ancora valida, oppure se è stato deallocato. In questo secondo caso il numero di riferimento avrà un valore predefinito dal linguaggio che segnalerà l'impossibilità di accedere a tale area.
Quando si assegna un puntatore ad un altro, non è il valore della tombstone che cambia, ma è il valore del puntatore, che al posto di puntare alla sua tombstone punterà direttamente a quella dell'altro puntatore.
Per quanto semplice, questo meccanismo presenta un conto salato da pagare in termini di spazio ed efficienza. Infatti le tombstones portano via spazio, e anche tempo, per la loro creazione, per il loro controllo e per il fatto che quello che avviene è un doppio accesso indiretto.
Inoltre le tombostones invalidate non vengono eliminate, ma vengono mantenute, con la possibilità di esaurire velocemente lo spazio. Per questo motivo alcune volte si implementa anche un piccolo garbage collector che permetta il riutilizzo delle tombstone non più utilizzate.

2. Lucchetti e chiavi

È un metodo alternativo a quello delle tombstones che permette di ovviare ai dangling reference verso lo heap è quello che viene chiamato lucchetto e chiavi (locks and keys). Inoltre questo metodo evita l'accumulo delle tombstones.
Ogni volta che viene creato un oggetto sullo heap, viene associato all'oggetto anche un lucchetto, ovvero una parola di memoria in cui viene memorizzato un valore casuale. Il puntatore è quindi costituito da una coppia: l'indirizzo vero e proprio, e una chiave. Tutte le volte che si dereferenzia un puntatore, la macchina controlla che la chiave apra il lucchetto, cioè che l'informazione contenuta nella chiave coincida con quella del lucchetto. Nel momento in cui un oggetto viene deallocato, il suo lucchetto viene annullato, memorizzandovi qualche valore canonico, così che tutte le chiavi che prima lo aprivano ora causino un errore.
Quando un puntatore viene assegnato a un altro viene assegnata tutta la coppia.
Anche questo metodo ha un costo; in termini di spazio costa anche di più delle tombstones, dall'altra parte però sia i lucchetti che le chiavi sono deallocati insieme all'oggetto o al puntatore.
In termini di efficienza va anche peggio in quanto creazione, controllo e assegnazione sono molto pesanti.

11. Garbage collection

Nei linguaggi senza deallocazione esplicita della memoria sullo heap, si rende necessario dotare la macchina astratta di un meccanismo automatico per recuperare la memoria. Questo meccanismo è chiamato garbage collector. Implementato per la prima volta in LISP, e dapprima in linguaggi funzionali, è stato implementato anche in quelli imperativi. Questo processo si compone di due fasi:

1. Distinguere gli oggetti vivi da quelli non più utilizzati

2. Recuperare gli oggetti non più utilizzati, così da poterli riutilizzare

Solitamente queste due fasi sono temporalmente separate. Inoltre le tecniche per l'individuazione degli elementi non più in uso sono solitamente conservative per una migliore efficienza.
I garbage collector possono essere distinti in base alla tecnica che usano[23].

1. Contatori dei riferimenti

La risposta più semplice che possiamo dare alla domanda relativa a quando un oggetto non è più utilizzato è: quando non vi sono puntatori verso di esso. La tecnica dei contatori dei riferimenti si basa su questa definizione e costituisce probabilmente il modo più elementare per realizzare un garbage collector. Ogni volta che viene allocato un oggetto, la macchina astratta alloca insieme anche un intero, inaccessibile al programmatore, che indica il numero di puntatori a tale oggetto.
Quando si esce da un ambiente locale, sono decrementati di uno tutti i contatori degli oggetti puntati da puntatori locali.
Una volta che un contatore raggiunge il valore 0, l'oggetto relativo viene deallocato e restituito alla lista libera. Se tale oggetto contiene dei puntatori, allora a macchina astratta segue a ritroso tutta la lista dei puntatori, decrementando tutti i contatori di uno, e recuperando ricorsivamente eventuali oggetti i cui contatori arrivano a 0.
Un chiaro vantaggio di questa tecnica è il fatto è che intrinseca nell'esecuzione del programma.
Il difetto maggiore invece è che non riesce a deallocare tutte le strutture circolari.
Inoltre sono anche molto inefficienti, in quanto hanno un costo proporzionale al lavoro complessivo del programma.

2. Mark and sweep

La tecnica di mark and sweep prende il proprio nome dalle modalità con le quali vengono realizzate le due fasi astratte che abbiamo menzionato all'inizio:

- Mark: per riconoscere cosa è inutilizzato, si attraversano una prima volta tutti gli oggetti dello heap, marcando ciascuno di essi come "inutilizzato"; partendo poi dai puntatori attivi presenti sulla pila (l'insieme radice, o root set) , si attraversano ricorsivamente tutte le strutture dati presenti sullo heap (in genere attraverso una visita in ampiezza o in profondità), marcando come "in uso" ogni oggetto che viene attraversato.
- Sweep: lo heap viene spazzato (swept): tutti i blocchi marcati come "in uso" sono lasciati immutati, mentre quelli "inutilizzati" sono restituiti alla lista libera.

È necessario riconoscere i blocchi allocati nello heap: saranno necessari descrittori che diano le dimensioni di ogni blocco allocato.
La differenza principale con il primo metodo è che non è incrementale, ovvero verrà avviato solamente quando lo spazio sullo heap sta per esaurirsi. Tuttavia in questo modo il programma dovrà sospendere momentaneamente la sua esecuzione.
Altri difetti di questa tecnica sono:

- È causa di frammentazione esterna[24]
- Inefficienza: richiede un tempo proporzionale alla dimensione totale dello heap

3. Intermezzo: rovesciare i puntatori

Il collector necessita la visita ricorsiva di un grafo, durante la fase di marcatura, che necessita però di una pila per memorizzare i punti di ritorno. Per marcare un grado in queste condizioni, pertanto, occorre usare astutamente lo spazio già presente per i puntatori, secondo una tecnica che va sotto il nome di rovesciamento dei puntatori (pointer reversal).
Per tanto data una struttura ad albero, per visitarla correttamente, occorre marcare un nodo e visitare ricorsivamente le sottostrutture, memorizzando in una pila l'indirizzo del blocco appena visitato.

[23] Come al solito molte volte vengono usati più metodi o dei misti.
[24] Come anche la tecnica dei contatori di riferimenti.

4. Mark and compact

Per ovviare alla frammentazione causata dal mark and sweep, si può modificare la fase di sweep e di convertirla in una fase di compattamento, spostando gli oggetti vivi e rendendoli contigui e lasciare memoria libera. Lo spostamento avviene scandendo tutto lo heap e spostando man mano i blocchi che si incontrano.
Si tratta tuttavia di una tecnica che necessita di diversi passaggi: calcolo della nuova posizione, aggiornare i puntatori, spostare davvero gli oggetti. Tuttavia i vantaggi sono notevoli.

5. Copia

In questa modalità non c'è una vera e propria fase di marcatura, ma avviene una copia e compattazione dei blocchi vivi.
Nei garbage collector basati su copia, lo heap è diviso in due parti di uguali dimensioni. Durante l'esecuzione normale solo uno dei due semispazi è in uso, e la memoria usata in quello in uso è tutta allocata a un'estremità andando assottigliandosi sempre di più. Una volta che la memoria del semispazio è esaurita, viene invocato il garbage collector, questo a partire dai puntatori presenti sulla pila, inizia una visita delle strutture concatenate presenti nel semispazio corrente, e le copia una dopo l'altra nell'altro semispazio compattandole a un'estremità.
La visita e la copia viene eseguita in modo efficiente con l'algoritmo di Cheney. Inizialmente si copiano nel semispazio non usato tutti gli oggetti immediatamente raggiungibili a partire dal root set (pila). Questo primo insieme viene gestito come una coda: si prende il primo di questi oggetti e si aggiungi al suo fondo il primo degli oggetti puntati da questo oggetto, e contemporaneamente si modificano questi puntatori. In questo modo si copiano nel nuovo spazio tutti gli oggetti figli del primo elemento. Si procede ricorsivamente su tutta la coda, fino ad esaurire la coda.
Questo tipo di collector lavora meglio maggiore è la memoria di ciascun semispazio. Infatti, maggiore è la memoria, minore saranno le volte in cui verrà chiamato.

Capitolo 11

Come si è visto nel capitolo precedente, i linguaggi di programmazione mettono a disposizione dei metodi per creare dei nuovi tipi di dato. Tuttavia questi metodi sembrano molto limitati in quanto fanno tutti riferimento a tipi già presenti. In questo capitolo vedremo dei metodi più sofisticati per creare nuovi tipi.

1. Tipi di dato astratto

L'introduzione di nuovi tipi (con i meccanismi precedenti= non permette al programmatore la definizione di tipi con lo stesso grado di astrazione goduto dai tipi predefiniti. Ovvero il programmatore non può nascondere l'informazione come si riesce a fare per i tipi predefiniti. Supponiamo di creare un tipo Pila con i metodi per manipolarla: non c'è nessuna sicurezza che quelli siano i soli metodi che permettano di gestirla, ma ci possono essere altri metodi creati da altri utenti.
Per ovviare a questo problema alcuni linguaggi permettono di definire delle astrazioni sui dati che si comportano come i predefiniti. Questo meccanismo, che viene chiamato **tipo di dato astratto**, è caratterizzato da:

1. Un nome per il tipo;
2. Un'implementazione (o rappresentazione) per tale tipo;
3. Un insieme di nomi di operazioni per la manipolazione dei valori di quel tipo, con i loro tipi;
4. Per ogni operazione, un'implementazione che usi la rappresentazione fornita al punto 2;
5. Una capsula di sicurezza che separi i nomi del tipo e delle operazioni dalle loro realizzazioni.

Per ciò il tipo di dato astratto pila potrebbe essere rappresentato come segue:

```
abstype Pila{
    type Pila=struct {
        //dichiarazione struttura di Pila
    }
    signature
        //Lista con le dichiarazioni dei metodi
    operations
        //definizioni dei metodi
}
```

Linguaggi che permettono i tipi di dato astratto sono ML e CLU.

2. Nascondere l'informazione

La distinzione tra interfaccia ed implementazione è fondamentale per le tecniche di sviluppo del software. Ad esempio una funzione astrae (cioè nasconde) il codice che costituisce il suo corpo (l'implementazione) mentre mostra la sua interfaccia, costituita dal suo nome e dal numero e tipi dei parametri (signature).
L'astrazione sui dati generalizza quella forma un po' primitiva di astrazione: non viene nascosto solo "come" una certa operazione è realizzata, ma anche le modalità di rappresentazione dei dati in modo che il linguaggio possa garantire che l'astrazione non venga violata.
Questo fenomeno viene chiamato **information hiding**.
L'information hiding porta a una conseguenza importante: sotto certe condizioni due tipi di dato astratti possono essere alternativi. Segue quindi la definizione di **specifica**, ovvero la descrizione della semantica delle operazioni di un certo tipo di dato, espressa non in termini del tipo concreto, ma per mezzo di relazioni generali astratte. Se le specifiche sono uguali, allora non importa (al cliente) quale tipo di dato è stato usato. Da ciò deriva anche una proprietà che si chiama indipendenza dalla rappresentazione.

1. Indipendenza dalla rappresentazione

Definizione 1: *Due implementazioni corrette*[25] *di una stessa specifica di un ADT sono osservabilmente indistinguibili da parte dei clienti di quel tipo.*

Si presti attenzione che vi è una versione debole di questa proprietà che accetta anche semplicemente che due segnature siano corrette, senza verificare cosa veramente faccia la funzione.

3. Moduli

Esiste poi una parte della programmazione, definita programmazione in grande che si occupa della realizzazione di un sistema complesso mediante composizione e assemblaggio di componenti più semplici. Per fare questo i soli tipi di dato non sono sempre sufficienti. Esistono ad esempio i **moduli**, o **packages**, che permettono di incapsulare insieme più tipi di dato astratto. Il metodo dei moduli permette di partizionare staticamente un programma in parti distinte, ciascuna delle quali dotata sia di tipi che di operazioni.

Da un punto di vista di principi non c'è grande differenza dai tipi di dato astratti, se non nella possibilità di definire più tipi in una volta. Da un punto di vista pragmatico invece fornisce molta più flessibilità, sia nella definizione del grado di "permeabilità" della capsula, sia nella possibilità di definire moduli generici, cioè polimorfi.

Tutti i moduli, indipendentemente dal linguaggio, sono divisi in parte pubblica e privata (che può contenere dichiarazioni non menzionate in quella pubblica.

[25] Con "implementazioni corrette" ci si riferisce al fatto che soddisfano entrambe una stessa specifica.

Capitolo 13

Viene ora presentata la programmazione funzionale in cui la computazione non avviene per modifica dello stato, ma per riscrittura di funzioni. La caratteristica di questi linguaggi, almeno nella loro versione pura, è quella di non avere memoria. Fissato un ambiente, un'espressione denota sempre lo stesso valore.

1. Computazioni senza stato

La macchina di Von Nuemann, che è quella da cui prendono spunto la maggior parte, se non tutti, i linguaggi di tipo imperativo ha come base la nozione di variabile modificabile e di associazione.
Non si tratta però dell'unico modello che permetta di ottenere un linguaggio Turing completo. Un altro modello è quello che non prevede la nozione di variabile modificabile, e quindi di stato della macchina. Tutta la computazione sarà espressa mediante una modifica sofisticata dell'ambiente, che avviene tramite le funzioni (spesso di ordine superiore).
Senza assegnamento anche l'iterazione perde il proprio senso. Per questo motivo il metodo usato in questo modello per ripetere computazioni sarà la ricorsione.
Abbiamo quindi delineato i punti fondamentali di questo paradigma: **funzioni**, **ordine superiore** e **ricorsione**.
Questo paradigma si chiama paradigma di programmazione funzionale, e anche se meno diffuso è tanto "antico" quanto quello imperativo, infatti la base di questo paradigma è una particolare teoria matematica detta λ[26] −calcolo che era già diffusa ai tempi di Turing.
Pochi sono i linguaggi funzionali totalmente puri (Miranda e Haskell), altri quali LISP, ML e Scheme contengono delle parti di paradigma imperativo per facilitare il programmatore.

1. Espressioni e funzioni

Nell'usuale pratica matematica c'è qualche ambiguità tra la definizione di una funzione e la sua applicazione. Nella pratica matematica tale ambiguità è del tutto superabile, mentre per il programmatore funzionale è fondamentale capire la differenza tra la definizione di una funzione e la funzione applicata, in quanto non sempre è possibile distinguere dal contesto (a differenza della matematica) a quale delle due ci si riferisce. Per tanto in matematica quando si scrive $f(x) = x^2$ ovvero si dà la definizione della funzione, in ML si scriverebbe

```
val f = fn x => x*x;
```

La parola riservata val introduce nell'ambiente una nuova associazione tra un nome e un valore, in questo caso il nome f è legato a quella funzione che trasforma x in x*x.
Per l'applicazione di una funzione ad un argomento si mantiene la notazione tradizionale, scrivendo $f(2)$, $(f\ 2)$ o $f\ 2$ si intende la valutazione dell'espressione risultante dall'applicazione della funzione f a 2. Nei linguaggi funzionali è possibile scrivere una funzione e applicarle un valore, ottenendo un risultato, senza assegnarle un nome. Ad esempio

```
(fn y => y+1)(6);
```

darà come risultato 7.
In ML si associa a sinistra, e non è necessario sempre necessario l'uso di parentesi[27].
ML permette anche un'altra notazione per definire funzioni:

```
fun f x=x*x;
```

questo tipo di notazione assomiglia molto più a quello matematico. Ad esempio la definizione:

```
fun F x1 x2 x3 … xn=corpo;
```

corrisponde a:

```
val F = fn x1 => fn x2 => fn x3 => … => fn xn => corpo;
```

o

[26] Si legge lambda calcolo
[27] L'uso delle parentesi è molto particolare, e molte volte conviene metterne in più, stando però attenti a casi particolari.

```
val F = fn (x1, x2, x3, …, xn) => corpo;
```

Vedremo successivamente cos'è l'elemento (x1, x2, x3, …, xn).
Prestiamo attenzione a un'ulteriore differenza tra fun e fn per quanto riguarda la ricorsione. Mentre fun non ha bisogno di altri specificatori, fn necessita dello specificatori rec nel caso in cui sia ricorsiva:

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

corrisponde a:

```
val rec fatt =fn n =>
      if n=0
            then 1
            else n*fatt(n-1);
```

e non a:

```
val rec fatt =fn n =>
      if n=0
            then 1
            else n*fatt(n-1);
```

2. Composizione come riduzione

Se si eccettuano le funzioni aritmetiche e l'espressione condizionale, il procedimento con il quale si passa da un'espressione complessa ad un valore può essere descritto come un processo di riscrittura definito **riduzione.**

```
fatt 3 → (fn n => if n=0 then 1 else n*fatt(n-1)) 3
       → if 3=0 then 1 else 3*fatt(3-1)
       → 3*fatt(3-1)
       → 3*fatt(2)
       → 3*((fn n => if n=0 then 1 else n*fatt(n-1)) 2)
       → 3*(if 2=0 then 1 else 2*fatt(2-1))
       → 3*(2*fatt(2-1))
       → 3*(2*fatt(1))
       → 3*(2*((fn n => if n=0 then 1 else n*fatt(n-1)) 1))
       → 3*(2*(if 1=0 then 1 else 1*fatt(1-1))
       → 3*(2*(1*fatt(0))
       → 3*(2*(1*((fn n => if n=0 then 1 else n*fatt(n-1)) 1)))
       → 3*(2*(1*(if 0=0 then 1 else n*fatt(n-1))))
       → 3*(2*(1*1))
       → 6
```

3. Gli ingredienti fondamentali

Un linguaggio funzionale non ha comandi ma solo espressioni.
I due costrutti principali che permettono di definire espressioni sono:

- Astrazione
- L'applicazione di un'espressione (f) ad un'altra espressione.

No vincoli su passaggio o ritorno di funzioni da parte di funzioni.
La valutazione può essere definita mediante un semplice procedimento di riscrittura simbolico di stringhe (riduzione), che semplifica ripetutamente un'espressione fino al raggiungimento di una forma semplice. Questo procedimento procede tramite due operazioni:

- Ricerca nell'ambiente
- Sostituzione di un identificatore con la sua definizione.

Definizione 1: *Un redex (che sta per reducible expression) è un'applicazione della forma ((fn x => corpo) arg).*

Il ridotto di un redex ((fn x => corpo) arg) è l'espressione che si ottiene sostituendo in corpo ogni occorrenza (libera) del parametro formale x con una copia di arg (evitando cattura di variabili).

Definizione 2: ***B-regola****: Un'espressione exp nella quale compaia come sottoespressione un redex, si riduce in exp1 (notazione: exp -> exp1) dove exp1 si ottiene da exp rimpiazzando il redex con il suo ridotto.*

Tutte le macchine astratte per linguaggi funzionali adottano estesi meccanismi di garbage collection.

2. Valutazione

Con il paragrafo di prima non si ha dato spiegazione di:

- Quale sia la condizione di terminazione della riduzione;
- Quale semantica precisa si debba alla *B*-regola nel caso in cui ci siano più redex.

1. I valori

Un valore è un'espressione che non dev'essere ulteriormente riscritta. In un linguaggio funzionale vi sono valori di due tipi:

- Primitivi: ad ogni tipo primitivo è associato un insieme di valori primitivi, che non danno luogo a valutazione.
- Funzioni: prendiamo ad esempio la funzione:

```
val G = fn x => ((fn y=> y+1)2);
```

La valutazione della funzione G non è 3 come si potrebbe pensare, ma bensì `fn x => ((fn y=>y+1)2)`. Infatti la maggior parte dei linguaggi funzionali definisce un valore fn x=>exp, e quindi anche se sono presenti dei redex all'interno, rimarranno tali fintanto che tale espressione non verrò applicata a qualche argomento.

2. Sostituzione senza cattura

Il metodo più usato dalle macchine astratte funzionali per realizzare una sostituzione senza cattura è quello di usare delle chiusure.

Partendo dal presupposto che lo stesso nome non è mai usato per indicare due variabili distinte in un'espressione, allora non si verificherà mai una cattura di variabile.

3. Strategie di valutazione

La maggior parte dei linguaggi funzionali quando si trova di fronte a funzioni di ordine superiore con più redex usa la strategia **leftmost** ovvero parte da sinistra. In alcuni caso però questo non è ancora sufficiente. È necessario distinguere altre tre strategie considerando il seguente codice:

```
fun K x y=x;
fun r z=r(r(z));
fun D u = if u=0 then 1 else u;
fun succ v=v+1;
val v=K(D (succ 0) (r 2);
```

1. **Valutazione per valore**: detta anche di ordine applicativo, o eager, un redex viene valutato solo se l'espressione che costituisce la sua parte argomento è già un valore, altrimenti continua in modo ricorsivo fino a trovare un argomento-valore e valuta tale espressione.
   Si noti però che se ci sono due argomenti devono essere valutati entrambi (sempre in ordine da sinistra). Con questa tipo di valutazione sarebbe stato valutato per prima cosa `succ 0` ottenendo così 1, quindi `D 1` ottenendo 1, che passato a K avrebbe dato: `K 1 (r 2)`. A questo punto però per la strategia per valore bisogna prima valutare gli argomenti, quindi si passa a valutare `r 2` che però diverge. v quindi non otterrà risultato.
2. **Valutazione per nome**: detta anche in ordine normale, prevede che un redex venga valutato prima della parte argomento. Ciò permette in alcuni casi di ottenere un valore anche laddove la strategia per valore diverge. Prendendo l'esempio si ottiene che `K(D (succ 0)` diventa: `fn y => D(succ 0)` e l'espressione per intero ha la forma: `fn y => D(succ 0)(r 2)`. Si procede quindi con il prossimo redex più esterno per ottenere

```
if (succ 0)=0
      then 1
```

```
else (succ 0)
```

e quindi si ottiene `succ 0` da cui 1. In questo caso quindi non c'è nessuna divergenze e il valore che otterrà v è 1.

3. **Valutazione lazy**: nella valutazione per nome, uno stesso redex può dover essere valutato più volte per effetto di qualche duplicazione che è intervenuta durante la riscrittura. Questo è dovuto al fatto che gli argomenti sono valutati dopo l'applicazione di una funzione. Tutto ciò è molto costoso in termini di efficienza.
   Per ovviare a questo problema, pur mantenendo i vantaggi della valutazione per nome, la strategia lazy procede come quella per nome, ma la prima vola che si incontra la copia di un redex viene salvato il suo valore, che verrà riutilizzato.

Queste ultime due strategie sono dette call by need ovvero un redex viene ridotto solo se necessario.

4. Confronto tra strategie

È sensato chiedersi se le precedenti strategie possano dare risultati diversi per una stessa espressione. Il seguente teorema fondamentale per il paradigma funzionale puro chiarisce:

Definizione 3: *Sia exp un'espressione chiusa. Se exp si riduce ad un valore primitivo val, usando una qualsiasi strategia, allora exp si riduce a val seguendo la strategia per nome. se exp diverge usando la strategia per nome, allora diverse anche per le altre due strategie.*

Questo teorema pone due limiti:

- Vale solo per i tipi primitivi;
- Le espressioni devono essere **chiuse**, ovvero tutte le sue variabili sono legate da qualche fn.

Poste queste due ipotesi, e strategie si differenziano solo per il fatto che una può divergere mentre le altre no.

Importante è anche la seguente proprietà che permette di dire che la strategia lazy e per nome hanno lo stesso valore:

Definizione 4: *Fissata una strategia qualsivoglia, nello scope dello stesso ambiente la valutazione di tutte le occorrenze di una stessa espressione produce sempre lo stesso valore.*

Ci si può chiedere che senso abbia utilizzare strategie diverse da quella per nome visto che questa è quella più generale. Inoltre non avrebbe neanche senso utilizzare strategie che non siano call by need, in modo da migliorare l'efficienza. Tuttavia il calcolo dell'efficienza non è così facile. Infatti se bene le strategie call by need risparmino sui redex, allo stesso tempo sono molto più complicate da costruire. Inoltre molte volte la strategia call by value funziona molto meglio sulle architetture tradizionali implementando semplicemente una funzione che gestisca gli argomenti inutili.

LISP, Scheme e ML usano una strategia per valore, mentre Miranda e Haskell per valutazione lazy.

3. Programmare in un linguaggio funzionale

Quello descritto finora può essere considerato il nucleo dei linguaggi funzionali, attorno a cui vengono costruiti una serie di meccanismi per rendere il linguaggio più espressivo e semplice.

1. Ambiente locale

Come per i linguaggi più "convenzionali" esiste più di un ambiente. Uno dei meccanismi per creare un ambiente locale è il seguente:

```
let x=exp in exp1 end
```

che introduce il legame tra x ed il valore di exp in uno scope che include solo exp1. Si noti che questo tipo di ambiente è ottenibile anche per mezzo di espressioni funzionali. Infatti corrisponde perfettamente a:

```
(fn x=> exp1)exp
```

2. Interattività

Sono forniti meccanismi per l'importazione di definizioni da file testuali (ad esempio in ML si usa il comando `use "nomefile.sml";`) e per l'esportazione dell'ambiente corrente.

3. Tipi

Come ogni buon linguaggio di programmazione, anche quelli funzionali permettono di definire tipi. Ad eccezione di Scheme che usa dei meccanismi dinamici per la gestione dei tipi, tutti gli altri hanno un meccanismo statico. Tali sistemi permettono la definizione di nuovi tipi quali:

- **Coppie**: ad esempio `int*int`
- **Liste:** `[1, 2, 3]` è una lista di interi
- **Record**: come le coppie solo con n valori.

Nei linguaggi funzionali è importante ricordare che le funzioni sono tipizzate ovvero ritornano sempre lo stesso tipo. Ad esempio la funzione:

```
fun F f n = if (n=0)
      then f(1)
      else f("pippo");
```

illegale perché il parametro f dovrebbe avere sia valore intero nel caso in cui venga chiamata la funzione `f(1)`, sia come stringa nel caso in cui venga chiamata la funzione `f("pippo")`.

Un altro tipo di espressione illegale è quello nel quale non c'è modo di assegnare un unico tipo consistente al parametro formale, ad esempio:

```
fun Delta x= x x;
```

infatti a sinistra dell'uguale la x ha tipo 'a->'b, mentre a destra ha tipo 'a, e non c'è modo per unificare i due tipi.

Questo tipo di sistema di tipi è detto forte. Un linguaggio come Scheme non ha problemi su una funzione come Delta.

Ci soffermiamo un attimo sul tipo lista in ML e vediamo alcuni costrutti. Per prima cosa una lista vuota si indica con `[]` o con `nil`. Per unire due liste si usa il comando `@`, quindi `[1, 2, 3]@[4, 5, 6]` crea la lista `[1, 2, 3, 4, 5, 6]`. Per unire un elemento solo a una lista si usa il comando `::`, ovvero `1::[2, 3]` darà la lista `[1, 2, 3]`. Se invece si vuole unire un valore alla fine della lista la soluzione più facile è quello di inserirlo tra parentesi per formare una nuova lista e unire le due liste: ad esempio se vogliamo inserire 3 alla fine di `[1, 2]`: `[1, 2]@[3]`. Importantissimo per le funzioni è che le liste possono essere scritte come `x::t`, dove `x` è il primo elemento della lista (**head**) e `t` (**tail**) è il resto.

4. Pattern matching

È un meccanismo alternativo all'if, molto efficace con le funzioni ricorsive. È presente soprattutto in ML e Haskell. Ricorda un po' la forma dello switch. Prendiamo ad esempio la funzione di Fibonacci così definita:

```
val rec Fibo =fn n=> if (n=0)
            then 1
            else if (n=1)
                  then 1
                  else Fibo (n-1)+Fibo(n-2);
```

può essere riscritta con il pattern matching come:

```
val rec Fibo = fn
0  =>1
|1 =>1
|n =>Fibo(n-1)+Fibo(n-2);
```

La parte più interessante probabilmente è che la definizione è costituita da parametri formali che non sono più identificatori, ma possono essere pattern, cioè espressioni formate a partire da variabili, costanti e altri costrutti. Un pattern svolge il ruolo di un modello sui cui confrontare il parametro attuale.

Un vincolo fondamentale è che una variabile non può comparire due volte nello stesso pattern.
Riprendendo la definizione di lista in ML come x::t, risulta utilissima una qualsiasi funzione ricorsiva su lista con la forma:

```
val rec fz = fn
[]     => exp
|h::t => if (exp(h))
           then fz exp1
           else fz exp2
```

ovvero guarda se la lista passata è vuota, allora ritorna exp, altrimenti fa un controllo su un primo elemento e si richiama con exp1, che può essere ad esempio solo t, quindi richiamando la funzione sul resto della coda, oppure richiamando la funzione su un'altra lista.

5. Oggetti infiniti

In linguaggi quali Haskell, che adottano strategie di valutazione per nome o lazy, è possibile creare strutture dati (potenzialmente) infinite.

6. Aspetti imperativi

ML mette a disposizione dei meccanismi imperativi che introducono una nozione di stato modificato per effetto collaterale. Per ogni tipo T del linguaggio ML mette a disposizione un tipo T ref i cui valori sono variabili modificabili che possono contenere valori. Ad esempio se volessimo associare il valore 4 a una variabile I:

```
val I=ref 4;
```

bisogna immagine I come una cella in cui è contenuto un certo valore, per tanto se volessimo tale valore dobbiamo dereferenziare I. Per fare ciò si usa l'operatore !:

```
val n=!I+1;
```

ritornerà n=5.
Si può quindi formare anche una situazione di aliasing. Se associamo infatti I, senza dereferenziazione a un altro valore:

```
val J=I;
```

a questo punto J è un alias di I.
Per lavorare sul valore di I, ad esempio incrementandolo di 1, la forma è la seguente:

```
I:=!I+1;
```

4. Il paradigma funzionale a confronto

Ci chiediamo quindi perché studiare il paradigma funzionale.

1. Correttezza dei programmi

L'informatico principiante ritiene spesso che il progetto e la redazione di un programma efficiente costituiscano le operazioni più importanti nelle quali si sostanzia il proprio mestiere. L'ingegneria del software, tuttavia, ha largamente dimostrato, sia in teoria che con amplissimi studi sperimentali, che i fattori più critici dì un progetto software sono la correttezza, la leggibilità, la manutenibilità, l'affidabilità. In termini economici, questi fattori influiscono per ben più del cinquanta per cento del costo complessivo; in termini sociali, la manutenzione del software (che dipende in modo cruciale dalla sua leggibilità) può coinvolgere centinaia di persone diverse, in un arco temporale di decine di anni; in termini etico-deontologici, dall'affidabilità e dalla correttezza di un sistema software possono dipendere la vita, o la salute, di centinaia di persone.
In particolare, è oggi ancora lontano il momento in cui l 'informatico sarà in grado di produrre software con delle garanzie di correttezza paragonabili a quelle con le qual i un progettista edile rilascia i propri manufatti (ponti , colonne, strutture). Per il progetto edile, infatti, l'ingegnere ha a disposizione tutto un corpus di matematica applicata col quale "calcola" le strutture, ma l'informatico non ha base di matematica così solide quando si tratta di effetti collaterali. Infatti il loro studio è molto impegnativo e costoso, mentre lo studio di funzioni che non hanno effetti

collaterali, ma che si basano semplicemente sul principio di induzione hanno strumenti molto più potenti. Per tanto l'uso di un linguaggio di programmazione funzionale che non presenta effetti collaterali risulta molto più corretto.

2. Schemi di programmi

L'ordine superiore ha importanti vantaggi dal punto di vista pragmatico. Infatti questo viene solitamente sfruttato per definire degli schemi di programmi generali, da cui si ottengono programmi specifici. Consideriamo ad esempio le due funzioni per la somma e il prodotto degli elementi in una lista di interi:

```
fun somma nil =0
| somma n:: resto=n+somma (resto);
fun prod nil =1
| prod n:: resto=n+prod (resto);
```

Si possono notare varie somiglianze tra le due funzioni. In particolare si può ricostruire uno schema come segue:

```
fun fold f i nil =0
| fold f i (n::resto)=f (n, fold f i resto);
```

fold è in genere il nome che viene dato uno schema. A questo punto per ottenere la funzione somma e prodotto a partire da fold basta scrivere:

```
val somma= fold + 0;
val prod= fold * 0;
```

dove + e * non verranno visti come `(fold+0)` o `(fold*0)`, ma come l'operatore necessario per tale funzione.

5. Fondamenti di λ −calcolo