PARADIGMI DI PROGRAMMAZIONE

Gli algoritmi che vogliamo eseguire su una macchina devono essere rappresentati mediante le istruzioni di un opportuno linguaggio di programmazione L,che sarà poi definito da una specifica sintassi e semantica.

Definizione 1.0 (Programma di L) : un insieme di istruzioni di L
Definizione 1.1 (Macchina astratta) la M.A è un insieme di strutture dati ed algoritmi in grado di memorizzare ed eseguire programmi.

La macchina astratta è composta da una memoria e da un interprete . La memoria serve per memorizzare dati e programmi mentre l'interprete esegue le istruzioni contenute nei programmi.

Le operazioni che l'interprete esegue sono :

- *operazioni per l'elaborazione dei dati primitivi*,dovremmo avere delle operazioni che permettono di manipolare dati primitivi cioè dati che sono rappresentabili direttamente dalla macchina;Esempio : aritmetico logiche,manipolazione stringhe...
- *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 ; Esempio : salti,condizioni,chiamate a funzioni...
- *operazioni e strutture dati per il controllo del trasferimento dati*,servono per controllare come gli operandi e in generale i dati devono essere trasferiti dalla memoria all'interprete e viceversa;Esempio : acquisizione operandi e memorizzazione risultati
- *operazioni e strutture dati per la gestione della memoria*,riguarda tutte le operazioni relative all'allocazione di memoria per i dati e per i programmi;Esempio : Tramite registri (statica) o pile (dinamica)

Definizione 1.2 (Linguaggio macchina) data una macchina astratta M_L , il linguaggio L compreso dall'interprete di M_L è detto linguaggio macchina di M_L .

Si può realizzare una macchina astratta tramite :

- *realizzazione in hardware* : è teoricamente sempre possibile ma è utilizzato solo per macchine di basso livello o macchine dedicate ovvero apparecchi che sono nati per svolgere una determinata operazione. Questo la rende ovviamente molto veloce ma ne annulla la flessibilità;
- *simulazione emulazione mediante firmware* : le strutture dati e algoritmi della MA sono realizzati tramite microprogrammi residenti in rom.. Questo permette comunque una buona velocità ed una discreta flessibilità anche se costringe a riprogrammare la rom
- *simulazione mediante software* : le strutture dati e algoritmi sono scritti nel linguaggio della macchina ospite. Questo permette di aver una grande flessibilità in quando è possibile crearlo su qualsiasi macchina ma ovviamente ne rallenta la velocità;

Un programma scritto in L può essere visto come una funzione parziale :

$$P^L : D \rightarrow D$$

tale che :

$$P^L(\text{Input}) = \text{Output}$$

se l'esecuzione di P^L sul dato di ingresso Input termina e produce come risultato Output,mentre la funzione non è definita se l'esecuzione di P^L sul dato di Input non termina.
Per eseguire un programma scritto in un linguaggio L,su un calcolatore che capisce solo il suo linguaggio Lm abbiamo a disposizione due metodi per risolvere questo problema :

- Compilazione : I programmi sono tradotti da un programma esterno che traduce L in Lm
- Interpretazione : I programmi in L sono eseguiti dalla macchina astratta Lm tramite un interprete che è scritto in Lm e simula la macchina di L;

Definizione 1.3 (Interprete) : un interprete per il linguaggio L,scritto nel linguaggio Lo è un programma che realizza una funzione parziale :

$$I^{L}o : (Prog^{L} \times D) \rightarrow D \text{ tale che } I^{L}o (P^{L}, input) = P^{L} (input)$$

Definizione 1.4 (Compilatore) : un compilatore da L a Lo è un programma che realizza una funzione

$$C_{L,Lo} : Prog^{L} \rightarrow Prog^{LO}$$

Approccio compilativo :

- abbiamo difficoltà dato la lontananza tra L e Lm
- maggiore efficienza in quanto il costo di decodifica è a carico del compilatore ed ogni istruzione è tradotta una sola volta
- poca flessibilità
- occupazione di memoria del codice prodotto

Approccio Interpretativo :

- abbia una scarsa efficienza della macchina di partenza
- abbiamo però una maggiore flessibilità e portabilità
- facilità di interazione durante l'esecuzione (debugging)

Esempio : le syscall sono tutte interpretate
Nel caso reale non è tutto interpretato oppure compilato ma si tende a mettere insieme le due tecniche,quindi si tende a compilare quei costrutti di L che sono vicini a quelli finali, e a interpretare i restanti.

Una volta lo sviluppo di un programma era operato spesso in linguaggi macchina per poter sfruttare più efficientemente l'hardware,al giorno d'oggi invece si utilizza una struttura a livelli nella quale si trovano macchine virtuali.

Definizione 1.4.1 (gerarchia di macchine astratte) : ogni livello usa le funzionalità di quelli inferiori ed aggiunge nuove funzionalità questa permette di avere una maggiore astrazione in quanto la macchina ospite può non essere necessariamente una macchina HW ma una macchina simulata già presente. Capita spesso che la nostra macchina è in realtà la macchina intermedia di un altro linguaggio.

Definizione 1.4.2 (linguaggio di programmazione) : è un formalismo artificiale nel quale poter esprimere algoritmi .

La descrizione di un linguaggio può avvenire individuando tre grandi ambiti :

- *grammatica* : è quella parte della descrizione di un linguaggio che risponde alla domanda "quali frasi sono corrette". Una volta definito l'alfabeto del linguaggio,si esegue un analisi lessicale utilizzando questo alfabeto,sono individuate le sequenze corrette di simboli che costituiscono le parole o token del linguaggio;
- *semantica* : è quella parte della descrizione del linguaggio che cerca di dare una risposta alla domanda "cosa significa una frase corretta" ,quindi attribuisce un certo significato ad una frase corretta

- *pragmatica* : è quella parte della descrizione di un linguaggio che si chiede “come usare una frase corretta e sensata”.
- *Implementazione*

Come facciamo a riconoscere cosa è corretto da cosa non lo è? Esistono diversi strumenti per arrivare ad una soluzione :

- *grammatiche libere*
- *automi*
- *BNF*

Definizione 2.2 (grammatica libera) : una grammatica libera da contesto è una quadrupla (NT,T,R,S) dove :

- NT è un insieme finito di simboli (i simboli non terminali,o le variabili,o le categorie sintattiche);
- T è un insieme finito di simboli
- R è un insieme finito di produzioni o regole ciascuna delle quali consiste in espressioni della forma V → w dove V è la testa della produzione , v è un simbolo non terminale
- S è un elemento di NT (simbolo iniziale)

Definizione 2.3 (BNF) : è una grammatica libera da contesto dove si usa :

- la notazione <w> per indicare un simbolo non terminale (w è una sequenza qualsiasi di caratteri)
- la notazione <A> : : = B per indicare A → b;
- la notazione <A> : : = b | c per indicare in un colpo solo A → b e A → c ;

Definizione 2.5 (Derivazione) : Fissata una grammatica G = (NT,T,R,S) e assegnate due stringhe v,w su NT e T , diciamo che da v si deriva immediatamente w (o anche : v si riscrive in un passo di w) e scriviamo v → w se w si ottiene sostituendo ad un simbolo non terminale V ,presente in v ,il corpo di una produzione di R la cui testa si V.

Le derivazioni sono quelle che ci danno le stringhe corrette secondo la grammatica.
Possiamo avere più derivazioni per la stessa stringa che sono equivalenti,esse ricostruiscono nello stesso modo la struttura della stringa mentre differiscono soltanto nell'ordine come le produzioni sono applicate.

Definizione 2.6 (Albero di derivazione) : è un modo con cui si rappresenta la derivazione di una stringa,e sono uno degli strumenti più importanti dell'analisi sintattica di un linguaggio di programmazione .
Più tecnicamente data una grammatica G=(NT,T,R,S) un albero di derivazione è un albero ordinato in cui :

- ogni nodo è etichettato con un simbolo NT ∪ T ∪ {∈}
- la radice è etichettata con S;
- ogni nodo interno è etichettato con un simbolo in NT

Definizione 2.7 (ambiguità) : una grammatica G si dice ambigua quando ammette più di un albero di derivazione . Si osservi che l'ambiguità nasce non dall'esistenza di più derivazioni ma dal fatto che almeno uno stringa ha più alberi di derivazione.

Una grammatica ambigua quindi è inutile per descrivere un linguaggio di programmazione perché non può essere usata per tradurre in modo univoco un programma .

La correttezza sintattica di un linguaggio di programmazione dipende talvolta dal contesto nel quale la frase si trova . Quindi stringhe corrette secondo la grammatica dunque possono essere legali solo in un determinato contesto . Alcuni esempi di vincoli sintattici contestuali :

- identificatore deve essere dichiarato prima dell'uso
- il numero di parametri formali di una funzione deve essere lo stesso di quello dei parametri attuali

Per quanto i vincoli contestuali facciano certo parte degli aspetti sintattico grammaticali di un linguaggio,nel gergo dei linguaggi di programmazione ci si riferisce ad essi come a vincoli di semantica statica che significa descrivibile con vincoli contestuali verificabili staticamente sul testo del programma , mentre semantica dinamica si riferisce quando il programma sarà eseguito .

La compilazione è composta da una serie di fasi in cascata . Le varie fasi partendo dal codice sorgente generano varie rappresentazioni intermedie fine a generare una stringa nel linguaggio oggetto . Esso non è necessariamente il linguaggio macchina o di basso livello, ma è soltanto il linguaggio verso la quale avviene la traduzione .

Le diverse fasi della compilazione sono :

- analisi lessicale : detta anche scansione è un operazione che scandisce il testo del programma da sinistra a destra in una sola passata cercando i token , però non viene fatto nessun controllo sulla sequenza dei token . *Lo strumento tecnico che si utilizza sono le grammatiche regolari* utili per riconoscere i token e anche perché per i linguaggi di programmazione il lessico può essere costituito da un insieme infinito di simboli e quindi un semplice elenco non basta .
- analisi sintattica : costruita la lista di token,l'analizzatore sintattico (o parser) cerca di costruire un albero di derivazione . Se la costruzione non và a buon fine per colpa di errori riscontrati,la compilazione viene abortita .
- analisi semantica : l'albero di derivazione viene sottoposto ai controlli relativi ai vincoli contestuali del linguaggio e viene aumentato e nuove strutture dati vengono generate . Ad esempio per un token che corrisponde ad un identificatore di variabile sarà associato il tipo,il luogo di derivazione e altre informazioni (scope etc..) .
- generazione della forma intermedia : una visita dell'albero di derivazione aumentato permette una prima generazione del codice , non è ancora opportuno generare codice nel linguaggio oggetto perché ci sono ancora ottimizzazioni da fare . Un compilatore può generare codice in più linguaggi oggetti .
- ottimizzazione del codice: il codice che si ottiene dalla fase precedente è inefficiente .Vi sono molte ottimizzazioni da fare quali :
    - rimozione del codice inutile : codice mai eseguito
    - espansione in-line delle chiamate di funzioni : sostituzione della chiamata di funzione con il corpo
    - fattorizzazione delle sottoespressioni : alcuni programmi calcolano più volte lo stesso valore
    - ottimizzazione dei cicli : eliminare all'interno del ciclo il calcolo di una sottoespressione il cui calcolo rimane costante durante le varie iterazioni .
- generazione del codice : a partire dalla forma intermedia ottimizzata viene generato il codice oggetto finale .

I NOMI E L'AMBIENTE

Definizione (nome) : un nome dunque non è altro che una sequenza di caratteri usata per rappresentare o denotare un altro oggetto .
Per quanto ovvio è importante sottolineare che il nome e l'oggetto da questo denotato non sono la stessa cosa : il nome infatti è solo una sequenza di caratteri mentre la sua denotazione può essere un oggetto complesso quale una variabile o una funzione
Quando un oggetto può avere più nomi si parla in questo caso di aliasing .L'uso dei nomi per le variabili realizza così un primo meccanismo elementare di astrazione sui dati ,questo ci permette di astrarre cosa avviene ai livelli più bassi ed evita a livello di programma di doversi preoccupare,di quale sia questa locazione . La corrispondenza tra nome e locazione di memoria dovrà essere garantita dall'implementazione e chiameremo ambiente quella parte dell'implementazione che è responsabile delle associazioni tra i nomi e gli oggetti che questi denotano.

Definizione (oggetti denotabili) : sono quelli oggetti a cui può essere dato un nome.
Il legame tra un nome e l'oggetto da esso denotato può avvenire in momenti diversi :

- progettazione del linguaggio : in questa fase sono definite le associazioni tra nomi e costanti primitive (es : + per la somma e int per indicare il tipo degli interi)
- scrittura del programma : possiamo considerare questa fase come in cui inizia la definizione di alcune associazioni che poi saranno completate successivamente (es : associazione tra un identificatore e una variabile ma viene effettivamente completata solo quando sarà allocato spazio in memoria)
- compilazione : il compilatore traducendo i costrutti del linguaggio di alto livello in codice macchina , alloca anche spazio di memoria per alcune strutture dati che possono essere gestite staticamente (es variabili globali di un sistema) .
- esecuzione : tutte le associazioni che non vengono definite devono essere realizzate in questa fase (procedure ricorsive)

in pratica si distinguono due fasi principali usando i termini “statico” e “dinamico” .

- Statico : ci si riferisce a tutto quello che succede prima dell'esecuzione
- dinamico : tutto quello che avviene al momento dell'esecuzione

AMBIENTE

definizione (ambiente) : l'insieme delle associazioni tra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma e in uno specifico momento dell'esecuzione.insomma l'ambiente è quella componente della macchina astratta che per ogni nome introdotto dal programmatore e in ogni punto del programma permette di determinare quale sia l'associazione corretta .

Definizione (dichiarazione):è un costrutto che permette di introdurre una nuova associazione nell'ambiente .

Definizione (blocco) : è una regione testuale del programma identificata da un segnale di inizio e fine che può contenere dichiarazioni locali a quella regione (cioè che compaiono nella regione)
i blocchi possono essere distinti in due casi :

- blocco associato ad una procedura : è il blocco associato alla dichiarazione di una procedura e che testualmente corrisponde al corpo della procedura stessa , esteso con le dichiarazioni relative ai parametri formali ;
- blocco in-line (anonimo) : è il blocco che non corrisponde ad una dichiarazione di procedura ma che può comparire in una qualsiasi posizione

L'ambiente cambia durante l'esecuzione di un programma ,tuttavia i cambiamenti avvengono

generalmente in due momenti distinti : all'entrata e all'uscita di un blocco . 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 il cui compare la ridefinizione la nuova dichiarazione nasconde o maschera la precedente quindi l'associazione sarà disattivata appena entra nel blocco e riattivata all'uscita .

L'ambiente associato ad un blocco è costituito dalle tre parti seguenti :

- ambiente locale : quello costituito dall'insieme delle associazioni per nomi dichiarati localmente al blocco,nel caso in cui il blocco sia relativo ad una procedura l'ambiente locale contiene anche le associazioni relative ai parametri formali dato che possono essere visti come variabili dichiarate localmente .
- Ambiente non locale : questo è l'ambiente costituito dalle associazioni relative ai nomi che sono visibili all'interno di un blocco ma che non sono stati dichiarati localmente.
- Ambiente globale : infine questo è l'ambiente costituito alle associazioni create all'inizio dell'esecuzione del programma principale . Contiene dunque le associazioni per i nomi che sono usabili in tutti i blocchi che compongono il programma .

Quando si entra in un nuovo blocco si hanno le seguenti modifiche dell'ambiente :

- sono create le associazioni fra i nomi dichiarati localmente al blocco e i relativi oggetti denotati
- sono disattivate le associazioni per quei nomi già esistenti all'esterno del blocco che siano ridefiniti al suo interno

anche all'uscita da un blocco l'ambiente viene modificato in questo :

- sono distrutte le associazioni fra i nomi dichiarati localmente al blocco e i relativi oggetti denotati
- sono riattivate le associazioni per i nomi già esistenti all'esterno del blocco che erano stati ridefiniti al suo interno .

Più in generale possiamo identificare le seguenti operazioni sui nomi e sull'ambiente :

- creazione di un associazione fra nome ed oggetto denotato (naming) : corrisponde all'elaborazione di una dichiarazione (o di un legame tra parametro formale e attuale)
- riferimento di oggetto denotato mediante il suo nome (referencing) : corrisponde all'uso di un nome,che servirà per accedere all'oggetto denotato
- disattivazione di un'associazione fra il nome e l'oggetto denotato : corrisponde all'ingresso in un blocco dove viene creata una nuova associazione per quel nome,la vecchia associazione non viene distrutta ma rimane nell'ambiente (inattiva) .
- riattivazione di un'associazione fra il nome e l'oggetto denotato : viene effettuato relativamente alle associazioni locali ,quando si esce dal blocco dove tali associazioni erano state create . L'associazione viene rimossa dall'ambiente e non più utilizzabile

per quanto riguarda gli oggetti denotabili sono possibili le seguenti operazioni :

- creazione di un oggetto denotabile : questa operazione avviene allocando la memoria necessaria a contenere l'oggetto ; a volte include anche l'inizializzazione
- accesso ad un oggetto denotabile : tramite il nome e quindi l'ambiente possiamo accedere all'oggetto denotabile e quindi ottenerne il valore .
- Modifica di un oggetto denotabile : sempre tramite il nome possiamo accedere all'oggetto denotabile e quindi modificarne il valore
- distruzione di un oggetto denotabile : un oggetto può essere distrutto deallocando la memoria che gli era stata riservata

tuttavia non è detto che il tempo di vita di un oggetto denotabile coincida con il tempo di vita

dell'associazione tra nome e oggetto denotabile .

SCOPE STATICO E DINAMICO

Definizione (regola di scope) : come abbiamo enunciato prima si è stabilito che una dichiarazione locale ad un blocco è visibile all'interno del blocco e nei blocchi annidati ma non specifica se tale nozione debba essere considerato in modo statico (dipende dalla struttura del programma) o dinamico (quando sono influenzate anche dal flusso di controllo a run-time) .

Definizione (scope statico) : la regola dello scope statico o regola dello scope annidato più vicino è definita dai seguenti punti :

- le dichiarazioni locali di un blocco definiscono l'ambiente locale di quel blocco . Le dichiarazioni locali di un blocco includono solo quelle presenti nel blocco e non quelle eventualmente presenti in blocchi annidati all'interno del blocco in questione
- 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 esiste . Se non esiste si considera quella nel blocco esterno e così fin quando non la si trova altrimenti errore .
- Un blocco può avere un nome , nel qual caso tale nome fa parte dell'ambiente locale del blocco immediatamente esterno che contiene il blocco a cui abbiamo dato un nome . Questo è il caso anche dei blocchi associati alle procedure .

Tra i vari dettagli della regola che la dichiarazione di una procedura introduce un'associazione per il nome della procedura nell'ambiente locale del blocco che contiene la dichiarazione .

Definizione (scope dinamico) : secondo la regola dello scope dinamico,associazione valida per X in un qualsiasi punto P del programma è la più recente associazione creata per X che sia ancora attiva quando il flusso d'esecuzione arriva a P .

occorre notare che queste due regole servono solo per la determinazione dell'ambiente che è contemporaneamente non locale e non globale : per l'ambiente locale e globale le due regole coincidono .

GESTIONE DELLA MEMORIA

La gestione della memoria costituisce una delle funzionalità dell'interprete associato ad una macchina astratta . Questa funzionalità gestisce l'allocazione di memoria per programmi e per dati ossia stabilisce :

- come debbano essere disposti in memoria
- quanto tempo vi debbono rimanere e
- quali strutture dati ausiliarie siano necessarie per reperire le informazioni dalla memoria

Nel caso di una macchina astratta di basso livello,quale ad esempio la macchina hardware,la gestione della memoria è molto semplice e può essere interamente statica : prima dell'inizio dell'esecuzione il programma e i dati vengono disposti in opportune zone di memoria dove vi rimangono per tutta la durata dell'esecuzione . Nel caso di linguaggi di alto livello le cose si complicano se si permette la ricorsione e quindi l'allocazione statica non è più sufficiente . Una tale gestione dinamica della memoria può essere realizzata usando una pila in quanto le attivazioni di procedure (o blocchi in-line) seguono una politica LIFO l'ultima procedura chiamata sarà la prima ad uscire .Vi sono casi in cui l'uso della pila non basta , e abbiamo bisogno di una particolare struttura di memoria detta heap per gestire allocazioni di memoria esplicite

Gestione statica della memoria : la memoria gestita staticamente è quella allocata dal compilatore prima dell'esecuzione . Gli oggetti per i quali la memoria è allocata staticamente risiedono in una zona fissa di memoria per tutta la durata dell'esecuzione del programma .
Tipici elementi allocati con questo tipo di memoria sono :

- *variabili globali* : sono visibili in tutto il programma
- *le istruzioni del codice oggetto* : non cambiano durante l'esecuzione
- *le costanti* : noti a tempo di compilazione
- *tabelle prodotte dal compilatore* : necessarie per il supporto a run-time

nel caso in cui il linguaggio non supporti la ricorsione possiamo gestire anche la memoria per le rimanenti parti del linguaggio , quindi di associare ad ogni procedura una zona di memoria nella quale memorizzare le informazioni locali della procedura stessa .
Tali informazioni sono costituite da :

- variabili locali
- eventuali parametri della procedura
- l'indirizzo di ritorno
- eventuali valori temporanei usanti in calcoli complessi

si noti che le successive chiamate in assenza di ricorsione occupano la stessa zona di memoria perché non possono esservi due chiamate di procedura attivi contemporaneamente .

Gestione dinamica mediante pila : la maggior parte dei linguaggi di programmazione permette una struttura a blocchi dei programmi . Essi vengono aperti usando una politica LIFO quindi è naturale gestire lo spazio di memoria usando una pila .

Definizione (record di attivazione) : Lo spazio di memoria allocato sulla pila dedicato ad un blocco in-line o ad un'attivazione di procedura . Si noti che ogni rda è associato ad una specifica attivazione di procedura . La pila dove sono memorizzati gli rda è detta pila a run-time o pila di sistema .

Osservazioni : Il sistema è usato in implementazioni di linguaggi che non supportano la ricorsione, se infatti il numero di procedure attive in quel momento è minore di quelle dichiarate una gestione a pila fa risparmiare un bel po' di spazio .

I vari settori del record di attivazione contengono le seguenti informazioni :

- *risultati intermedi* : nel caso in cui si debbano effettuare calcoli complessi si ha il bisogno di memorizzare i risultati intermedi
- *variabili locali* : sono quelle dichiarate all'interno di un blocco
- *puntatore di catena dinamica* : serve per memorizzare il puntatore al precedente record di attivazione sulla pila ,quest'informazione è necessaria perché gli rda possono avere dimensioni diverse
- *puntatore di catena statica* : serve per gestire le informazioni necessarie a realizzare le regole di scope statico
- *indirizzo di ritorno* : contiene l'indirizzo della prima istruzione da eseguire dopoché la chiamata di procedura attuale ha terminato l'operazione
- *indirizzo del risultato* : presente solo nel caso di funzioni contiene l'indirizzo della locazione di memoria dove memorizzare il valore restituito dalla funzione quando essa termina
- parametri : sono memorizzati i parametri attuali usati nella chiamata della procedura o funzione

gli indirizzi dei vari campi si ottengono aggiungendo un offset al valore del puntatore dinamico.
I nomi delle variabili non vengono memorizzati nell'rda e i riferimenti locali sono sostituiti dal compilatore con un indirizzo relativo (offset) rispetto ad una posizione fissa dell'rda del blocco nel quale le variabili sono dichiarate . Quindi il compilatore associa ad ogni variabile locale una posizione precisa all'interno del record di attivazione dato che le dichiarazioni di variabili all'interno di un blocco sono note staticamente .

GESTIONE DELLA PILA

Definizione (frame pointer) : è un puntatore esterno alla pila che indica l'ultimo rda inserito nella pila stessa

Definizione (stack pointer) : che indica la prima posizione di memoria libera nella pila .

Gli rda vengono inseriti o rimossi dalla pila a tempo di esecuzione : quando si entra in un blocco o si chiama una procedura , il relativo rda viene inserito nella pila per poi essere eliminato quando si esce da un blocco o termina la procedura .

Definizione (chiamante e chiamato) : la prima indica il programma o la procedura che effettua una chiamata (di procedura) e la procedura che è stata chiamata . La gestione della pila è fatta sia dal chiamante che dal chiamato

Per questo scopo e anche per gestire altre informazioni di controllo nel *chiamante* viene aggiunta una parte di codice detta :

- sequenza di chiamata : che è eseguita in parte prima della chiamata di procedura e in parte dopo la terminazione della procedura chiamata

Nel *chiamato* invece viene aggiunto un :

- prologo : da eseguirsi subito dopo la chiamata
- epilogo : da eseguirsi al termine dell'esecuzione della procedura

al momento della chiamata di procedura *la sequenza di chiamata* ed il *prologo* si devono occupare delle seguenti attività :

- modifica del contatore programma : è necessario passare il controllo alla procedura chiamata
- allocazione dello spazio sulla pila : predisporre lo spazio per il nuovo rda
- modifica del puntatore al rda : il puntatore dovrà indicare il nuovo rda che è stato inserito
- passaggio dei parametri
- salvataggio dei registri : i valori per la gestione del controllo,memorizzati nei registri devono essere salvati (es vecchio puntatore all'rda che viene salvato come puntatore di catena dinamica) .
- esecuzione del codice di inizializzazione : alcuni linguaggi prevedono costrutti espliciti per inizializzare alcuni elementi memorizzati nel nuovo record di attivazione .

Al momento del ritorno al controllo al programma chiamante,quando la procedura termina,*l'epilogo e la sequenza di chiamata* devono gestire le seguenti operazioni :

- ripristino del valore contatore del programma
- restituzione dei valori : valore calcolato dalla funzione,valori dei parametri che permettono di passare informazioni dal chiamato al chiamante .
- Ripristino dei registri
- esecuzione del codice di finalizzazione : alcuni linguaggi prevedono l'esecuzione di codice prima che alcuni oggetti locali siano distrutti .

- Deallocazione dello spazio sulla pila

Gestione dinamica mediante heap : nel caso in cui il linguaggio includa comandi per l'allocazione esplicita di memoria la gestione mediante pila è insufficiente . Dato che le operazioni di allocazione e deallocazione non possono essere gestite in ordine LIFO abbiamo bisogno di una particolare zona di memoria detta heap .

Definizione (heap) : è una zona di memoria nella quale i blocchi di memoria possono essere allocati e deallocati in modo libero.

Definizione (frammentazione interna) : si verifica quando si alloca un blocco di dimensioni maggiori di quella richiesta dal programma e quindi la porzione di memoria non verrà utilizzata
Definizione (frammentazione esterna) : si verifica quando la lista libera è composta da blocchi di dimensione piccola per cui anche se la somma della memoria totale è sufficiente non si riesce ad usare la memoria libera

I metodi di gestione dello heap si dividono in due categorie principali a seconda che i blocchi di memoria siano considerati di dimensione fissa o variabile .

- Dimensione fissa : in questo caso lo heap è diviso in blocchi di dimensione fissa limitata collegati con una struttura a lista . Quando si richiede a run-time un blocco di memoria viene preso il primo elemento della lista e viene rimosso da essa e viene restituito il puntatore a tale elemento e il puntatore alla lista viene aggiornato . Quando invece viene liberata la memoria il blocco viene collegato alla testa della lista .
- Dimensione variabile : se per esempio vogliamo allocare un array di dimensione variabile l'altro tipo di allocazione è insufficiente . Questo metodo migliora l'occupazione di memoria e la velocità di esecuzione per le operazioni di gestione dello heap .
  Riguardo all'occupazione di memoria si cerca di evitare fenomeni di frammentazione della memoria . Le tecniche di memoria tendono quindi a ricompattare la memoria libera unendo blocchi contigui in modo tale da evitare la frammentazione esterna per ottenere questo obbiettivo possono essere richieste operazioni aggiuntive che appesantiscono la gestione e ne riducono l'efficienza :
  - unica lista libera : costituita inizialmente da un unico blocco di memoria contenente l'intero heap , infatti è conveniente cercare di mantenere insieme i blocchi della massima dimensione possibile . Quando viene richiesta l'allocazione di un blocco di n parole di memoria le prime n parole vengono allocate e il puntatore dell'heap avanza di n,mentre i blocchi di memoria deallocati vengono collegati in una lista libera . Quando si raggiunge la fine dello spazio libero si dovrà passare ad utilizzare quello deallocato in un due modi diversi :
    1. **utilizzo diretto della lista libera** : viene mantenuta una lista di blocchi di dimensione variabile , quando viene richiesta un allocazione di un blocco di n parole si cerca un blocco maggiore di n in base alle politiche best fit,worst fit e first fit
    2. **compattazione della memoria** : quando si raggiunge la fine dello spazio libero dedicato inizialmente allo heap si spostano tutti i blocchi ancora attivi ad un'estremità dell'heap lasciando gli altri in modo contiguo a questo punto si aggiorna il puntatore dell'heap e si ritorna come prima .
  - Liste libere multiple : per ridurre il costo della gestione di allocazione di un blocco alcuni metodi di gestione dello heap usano più liste libere di dimensione diverse . Quando viene richiesto un blocco di dimensione n viene selezionata la lista che contiene blocchi di dimensione maggiore o uguale a n . Tutto questo è possibile implementarlo tramite buddy list oppure heap di fibonacci .

IMPLEMENTAZIONE REGOLE DI SCOPE

scope statico : catena statica : il rda direttamente collegato dal puntatore di catena dinamica non è necessariamente il primo rda nel quale cercare di risolvere un riferimento non locale ma tale primo rda è definito dalla struttura testuale del programma

definizione (puntatore di catena statica) : punta all'rda immediatamente esterno al blocco ,nel caso in cui B sia il blocco di una chiamata di procedura , quando verrà chiamata la procedura il puntatore di catena statica punterà al blocco che contiene la sua dichiarazione.

Definizione (puntatore di catena dinamica) : segue l'ordine di inserimento degli rda nella pila .

La gestione della catena statica a run-time rientra è fra le funzioni svolte dalla sequenza di chiamata dal prologo e dall'epilogo . Secondo l'approccio più comune al momento in cui si entra in un nuovo blocco il chiamante calcola il puntatore di catena statica del chiamato e quindi passa tale puntatore al chiamato, tale calcolo è abbastanza semplice e può essere compreso dividendo due casi :

- il chiamato si trova all'esterno del chiamante : secondo le regole di visibilità dello scope statico il chiamato essendo in un blocco esterno si deve già trovare sulla pila,se tra il chiamato e il chiamante vi siano k livelli di annidamento nella struttura a blocchi del programma : se il chiamante si trova a livello di annidamento n e il chiamato si trova a livello m supponiamo che si abbia k = n – m ,questo valore di k è determinabile dal compilatore dato che dipende dalla struttura statica del programma e dunque può essere associato alla chiamata in questione
- il chiamato si trova all'interno del chiamante : il chiamato è dichiarato nello stesso in cui avviene la chiamata e quindi il primo blocco esterno al chiamato è proprio quello del chiamante, quindi il chiamante può passare al chiamato direttamente il puntatore al proprio record di attivazione.

Una volta che il chiamato ha ricevuto il puntatore di catena statica deve soltanto memorizzarlo in un apposita area del record di attivazione,operazione che sarà fatta dal prologo . Questa distanza calcolata serve anche per risolvere a run-time riferimenti di variabili non locali senza effettuare alcuna ricerca sugli rda presenti nella pila. Per permettere questo tipo di gestione a run-time delle chiamate di procedure il compilatore si serve di una *tabella dei simboli* dove vengono memorizzati i nomi usati nel programma e tutte le informazioni necessarie a gestire gli oggetti denotati da tali nomi e per realizzare le regole di visibilità .

Osservazioni : è evidente osservare che il compilatore non può risolvere completamente in modo statico un riferimento di un nome non locale quindi occorre sempre seguire i link di catena statica . Questo perché non è possibile conoscere il numero di rda nella pila

Scope statico : il 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 dobbiamo effettuare k accessi in memoria invece con la tecnica del display riusciamo a ridurre il numero di accessi a due .
Il display non è altro che un vettore contenente tanti elementi quanti sono i livelli di annidamento dove l'elemento k-esimo contiene il riferimento al livello k correntemente attivo. La gestione del display è molto semplice oltre ad aggiornare il puntatore presente nel vettore si deve anche salvare il vecchio valore . Questa necessità può essere compresa esaminando i due casi possibili :

- il chiamato si trova all'esterno del chiamante : se il chiamante si trova a livello n e il chiamato a livello m con m < n , l'elemento m del display è aggiornato con il puntatore all'rda del chiamato e fino a quando il chiamato non termina la propria esecuzione il display

è costituito dai primi m elementi . Il vecchio valore dovrà essere salvato perché questo punta all'rda del blocco che tornerà ad essere attivo quando il chiamato terminerà la propria esecuzione e quindi il display tornerà ad essere quello esistente prima della chiamata .

- Il chiamato si trova all'interno del chiamante : in questo caso si incrementa il livello di profondità di annidamento raggiunto . Se il chiamante si trova a livello n allora si aggiunge un nuovo valore nella posizione n+1 contenente il puntatore al rda del chiamato . Nel caso si tratti della prima attivazione di quel blocco il vecchio valore non ci interessa . Tuttavia non possiamo sapere se è questo il caso infatti in cui si potrebbe giungere all'attuale chiamata tramite una serie di chiamate precedenti che utilizzavo il livello n+1

Scope dinamico : lista di associazioni
alternativamente alla memorizzazione diretta negli rda, le associazioni nome-oggetto possono essere memorizzate in una lista di associazioni detta A-list e gestita come 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 le associazioni vengono rimosse . Le informazioni conservate conterranno la locazione di memoria dell'oggetto,il suo tipo e altre informazioni necessarie per effettuare controlli semantici a run-time .
In questo caso però abbiamo due inconvenienti :

- dato che se usiamo la regola di scope dinamico non possiamo determinare staticamente qual'è il blocco da usare per risolvere tale riferimento non locale,l'unica possibilità è dunque memorizzare esplicitamente il nome e fare una ricerca
- il secondo inconveniente è proprio questa ricerca a run-time

Scope dinamico : CRT
per limitare gli inconvenienti precedenti possiamo usare una tabella centrale dell'ambiente. Secondo questa tecnica tutti i blocchi del programma ai fini della definizione degli ambienti fanno riferimento ad un'unica tabella centrale dove sono presenti tutti i nomi usati del programma e per ogni nome c'è un flag che indica se l'associazione è attiva o meno e un puntatore alle informazioni all'oggetto associato al nome . A run-time l'accesso avviene in un tempo costante se invece non tutti i nomi sono noti a tempo di compilazione la ricerca sulla tabella può avvenire a run-time tramite tecniche di hashing . Prendiamo il caso in cui un blocco A entra in un blocco B .
Le operazioni associate a questo metodo sono :

- disattivazione : quando si entra in un nuovo ambiente le vecchie associazioni vengono disattivate e salvate
- attivazione : all'uscita da un blocco le associazioni vengono ripristinate.

Alternativamente possiamo realizzare *un'unica pila nascosta* separata dalla tabella centrale in cui vengono messe le associazioni deattivate . Usando una crt normale o a pila nascosta l'accesso ad un'associazione avviene mediante un accesso alla tabella ed un accesso ad un'altra zona di memoria mediante il puntatore memorizzato nella tabella , non serve quindi alcuna ricerca a run-time .

STRUTTURARE IL CONTROLLO

Definizione (espressione) : è un'entità sintattica la cui valutazione produce un valore oppure non termina nel caso l'espressione è indefinita .

Espressione vs comando : la caratteristica principale di un'espressione che la differenzia da un comando è dunque che la sua valutazione produce un valore .

In generale un'espressione è composta da un'entità singola (variabile,costante...)oppure da un operatore applicato ad un certo numero di argomenti che sono a loro volta espressioni .

Possiamo distinguere tre tipi principali di notazioni :

- infissa : il simbolo di un operatore binario è posto fra le espressioni che rappresentano due operandi (es x+y)
- prefissa : il simbolo che rappresenta l'operatore precede gli operandi (es + x y)
- postfissa : è analoga alla prefissa solo che l'operatore segue gli operandi (es x y +)

a seconda di come si rappresenta un'espressione varia il modo in cui se ne determina la semantica e quindi la sua modalità di valutazione . In particolare l'assenza di parentesi potrebbe causare problemi di ambiguità .

- **Notazione infissa : precedenza e associatività** : usando questa notazione abbiamo facilità e naturalezza d'uso con una maggiore complicazione dei meccanismi di valutazione delle espressioni . Abbiamo dei problemi :
  - Se non si usano le parantesi bisogna chiarire le precedenze tra gli operatori , però per evitare un massiccio uso di parentesi i linguaggi usano delle regole di precedenza che specificano una gerarchia di valutazione . I linguaggi tuttavia rispettano la notazione matematica .
  - Un secondo problema è nella valutazione di un'espressione relativo all'associatività degli operatori che vi compaiono [es 15 – 5 – 3 = (15 - 5) - 3 = 15 - (5 – 3)]
- **Notazione prefissa** : le espressioni scritte usando la notazione prefissa possono essere valutate da sinistra a destra usando una pila .
- **Notazione postfissa** : la valutazione di un'espressione in questo caso è ancora più semplice in questo caso non serve controllare che sulla pila ci siano tutti gli operandi per l'ultimo operatore inserito , dato che gli operandi vengono letti prima degli operatori .

Le espressioni così come gli altri costrutti di un linguaggio di programmazione è possibile esprimerlo tramite un albero sintattico nel quale :

- ogni nodo è un operatore
- ogni foglia è un operando
- ogni sottoalbero è un'espressione

ogni albero può essere ottenuto dagli alberi di derivazione di una grammatica (non ambigua) per le espressioni . La rappresentazione ad albero chiarisce senza bisogno di parentesi la rappresentazione ed associatività degli operatori . Questo tipo di regole però non chiarisce come valutare i diversi operandi di uno stesso operatore , non è esplicito dunque se la valutazione degli operandi debba procedere o meno quella dell'operatore né più in generale se espressioni matematicamente equivalenti possono essere sostituite tra loro senza modificare la semantica . Quest'aspetto assumo particolare importanza nel nostro caso per cinque motivi :

- **Effetti collaterali** : la valutazione di un espressione può modificare il valore di alcune variabili mediante effetti collaterali , un effetto di questo genere è un'azione che influenza i risultati parziali o finali di una computazione . L'ordine di valutazione degli operandi è rilevante quindi ai fini del risultato . Riguardo gli effetti collaterali i linguaggi dichiarativi puri non ammettono effetti collaterali tout court, nei linguaggi che lo permettono si vieta l'uso di funzioni che possono causare gli effetti . Si ammettono questi effetti solo dichiarando l'ordine con cui vengono valutate le espressioni.
- **Aritmetica finita** : dato che l'insieme dei numeri rappresentabili in un calcolatore è finito il riordinamento di espressioni può causare problemi di overflow
- **Operandi non definiti** : quando si valuta un'applicazione di un operatore ad un operando si possono seguire due strategie di valutazione :
  - eager : consiste nel valutare prima tutti gli operandi e poi passare ad applicare l'operatore

ai valori ottenuti dalla valutazione degli operandi
    - lazy : consiste nel non valutare gli operandi prima dell'operatore ma nel passare gli operandi non valutati all'operatore il quale al momento della sua valutazione deciderà quali operandi effettivamente sono necessari .
    - Valutazione corto-circuito : se il primo operando di una disgiunzione ha valore vero allora è inutile controllare il secondo dato che comunque il risultato non cambia e quindi l'espressione è detta corto-circuitata
    - ottimizzazione : spesso l'ordine di valutazione delle sotto-espressioni influenza l'efficienza della valutazione di un'espressione a causa di considerazioni che hanno a che fare con la struttura della macchina hardware .

Se le espressioni sono presenti in tutti i linguaggi di programmazione non è così per i comandi che sono presenti solo nei linguaggi imperativi .

Definizione (comando) : un comando è un'entità sintattica la cui valutazione non necessariamente restituisce un valore ma può avere un effetto collaterale .

Comando vs espressione : una distinzione tra loro è possibile solo in sede di definizione formale della semantica di un linguaggio . Il risultato della valutazione di un'espressione è un valore, mentre per il comando è un nuovo stato che differisce da quello di partenza proprio per le modifiche apportate dagli effetti collaterali del comando stesso .

Definizione (variabile) : il paradigma imperativo classico usa la variabile modificabile,secondo questo modello la variabile è vista come una sorta di contenitore o locazione al quale si può dare un nome e che può contenere dei valori questi valori possono cambiare nel tempo per via di comandi di assegnamento .

- Modello a riferimento : una variabile non è costituita da un contenitore per un valore ma è un riferimento per un valore memorizzato sullo heap . Si tratta quindi di una nozione analoga a quello di puntatore

Definizione (assegnamento) : è il comando base che permette di modificare il valore delle variabili modificabili e quindi dello stato del linguaggio imperativo . L'assegnamento è un effetto collaterale dato che la valutazione di per se non restituisce alcun valore .

- Esempio x := 2 ; il simbolo " : = " è usato per indicare il contenitore al cui interno troviamo il valore della variabile . Quindi l'occorrenza a destra del simbolo indica il valore all'interno del contenitore . Questa importante distinzione è formalizzata nei linguaggi di programmazione usando due diversi insiemi di valori :
    - gli l-valori : sono quei valori che sostanzialmente indicano locazioni e quindi sono i valori di espressioni che possono stare alla sinistra di un comando di assegnamento .
    - Gli r-valori : sono invece i valori che possono essere contenuti nelle locazioni .

In generale dunque il significato di un tale comando è :

- calcola innanzitutto l'l-valore determinando così un contenitore
- calcolo quindi l'r-valore e modifica il contenuto del contenitore
- sostituendo il valore calcolato con quello precedente

In un linguaggio con modello a riferimento un assegnamento del tipo "x = e" fa si che x sia un riferimento per l'oggetto ottenuto dalla valutazione dell'espressione 'e' . dopo questo assegnamento x ed e puntano allo stesso oggetto , nel caso in cui l'oggetto sia modificabile la modifica si ripercuote su entrambi gli operandi .

Oltre al comando di assegnamento vi sono anche altri tipi di comandi :

- i comandi per il controllo di sequenza esplicito : quali il comando goto e il comando composto
- comandi condizionali : if_the_else
- comandi iterativi : for while do_while

COMANDI PER IL CONTROLLO DI SEQUENZA

definizione (comando sequenziale) : indicato in molti linguaggi da un " ; " permette di specificare in modo diretto l'esecuzione sequenziale di comandi .

Definizione (comando composto) : è possibile raggruppare una sequenza di comandi in un comando composto usando opportuni delimitatori quali : begin_end { } …

definizione (goto) : tale comando è ispirato direttamente alle istruzioni di salto nei linguaggi assembly e quindi al modello di controllo di sequenza della macchina hardware . L'esecuzione di questo comando trasferisce il controllo al punto del programma nel quale è presente l'etichetta .

Altri comandi sono : break,continue,case,return.

COMANDI CONDIZIONALI

i comandi condizionali o di selezione esprimono un'alternativa fra due o più possibili prosecuzioni della computazione sulla base di opportune condizioni logiche .

Comando if : è presente in tutti i linguaggi imperativi e in varie forme sintattiche che si possono ricondurre nella forma " if exp then c1 else c2 " dove exp è un espressione da valutare e c1 e c2 sono comandi . Per evitare problemi di ambiguità alcuni linguaggi usano un "terminatore" che indica dove termina il linguaggio condizionale .

Comando case : questo comando è una specializzazione del comando if con più rami nella sua forma più semplice è espresso come "case exp of ...label1 : c1...." .
Il case anche se è possibile sostituirlo con if annidati è utili in termini di leggibilità del codice . Esso è implementato a livello assembly usando un vettore di celle contigue detto "tabella di salto" ,in cui ogni elemento contiene l'indirizzo della prima istruzione dei comandi dei vari rami .

- Valutazione dell'argomento del case
- il valore ottenuto è usato come offset per calcolare la posizione, nella tabella di salto, dell'istruzione che permette il salto al comando del ramo scelto

case vs if annidati : con la tabella una volta calcolato il valore dell'espressione con due istruzioni di salto si arriva subito al codice da eseguire invece usando if annidati si dovrebbero valutare O(n) condizioni . Lo svantaggio della tabella è lo spazio,nel caso che i valori siano molto distanti . In questo caso utilizziamo tecniche di hashing e ricerca binaria.

COMANDI ITERATIVI

While : l'iterazione indeterminata è formata da due parti :

- una condizione del ciclo
- il corpo : costituito da un comando

in esecuzione il corpo verrà eseguito fin quando la guardia diviene falsa. L'aggiunta di questo costrutto in un linguaggio di programmazione che contenga solo comandi di assegnamento e condizionali rende subito il linguaggio turing completo .

For : l'iterazione determinata è realizzata tramite una variante del comando for che può essere descritto come :

for I = inizio to fine by passo do
corpo

componenti :

- I è una variabile detta indice
- inizio e fine sono espressioni
- passo è una costante (a tempo di compilazione) intera diversa da zero
- corpo è il comando che si vuole iterare

supponendo un passo positivo nel ciclo si avrà :

- valutazione e congelamento delle variabili inizio e fine .
- I viene inizializzata a inizio
- se I è maggiore di fine allora il ciclo termina
- si esegue corpo e si incrementa I del valore passo

Diversità tra linguaggi :

- non tutti prevedono il congelamento della variabile di controllo
- **numero di iterazioni** : anche se inizio è maggiore di fine in alcuni linguaggi si esegue il test solo dopo aver eseguito il corpo
- **passo** : affinché il compilatore possa generare il codice per il test la costante passo deve essere positiva,alcuni linguaggi usano speciali costrutti per indicare che il passo sia negativo in altri linguaggi viene guardato solo l'iteration count ([fine – inizio + passo]/passo)
- **valore finale dell'indice** : in molti linguaggi la variabile indice è visibile anche al di fuori del ciclo , inoltre l'indice alla fine dell'iterazione dovrebbe valere l'ultimo valore assegnatogli ciò potrebbe generare errori di ambiguità ed errori di tipo
- **salto del ciclo** : esso riguarda la possibilità di saltare all'interno del ciclo tramite un istruzione di salto goto, la maggior parte dei linguaggi la vietano mentre è possibile uscire da un ciclo usando in goto

nonostante i comandi condizionali,iterazione determinata e di assegnamento un linguaggio del genere non può considerarsi turing completo . Quello che manca è l'iterazione indeterminata . Dato che l'iterazione determinata è possibile tradurla in quella indeterminata attraverso il while,il uso è prettamente pragmatico , si ha più coscienza di cosa faccia quel ciclo e si evitano errori comuni

PROGRAMMAZIONE STRUTTURATA

definizione (programmazione strutturata) : permette un una sviluppo il più possibile strutturato del codice e del flusso di controllo .

Punti salienti di questa programmazione :

- **progettazione del programma top-down o comunque gerarchica** : il programma è sviluppato su una specifica abbastanza astratta e poi in seguito vengono aggiunti ulteriori dettagli
- **modularizzazione del codice** : raggruppamento dei comandi di ogni specifica funzione

dell'algoritmo che si vuole implementare

- **uso di nomi significativi** : l'uso di nomi significativi semplifica la comprensione del codice e quindi anche gli interventi di manutenzione
- **uso estensivo di commenti** : essi sono essenziali per la comprensione del codice,testing verifica e correzione
- **uso di tipi di dato strutturati** : la possibilità di usare opportuni tipi di dato come record per raggruppare informazioni di diverso tipo
- **uso di costrutti strutturati per il controllo** : per la realizzazione della programmazione strutturata abbiamo bisogno di questi costrutti cioè costrutti che abbiano un solo punto d'ingresso e un solo punto d'uscita . Se il comando C2 segue il comando C1 allora l'unica uscita del comando C1 sarà l'entrata del comando C2 ,questa proprietà di solo ingresso\uscita è violata però dal comando goto

RICORSIONE

definizione (ricorsione) : è un meccanismo alternativo all'iterazione per ottenere linguaggi di programmazione turing equivalenti . Una funzione ricorsiva è una procedura nel cui corpo compare una chiamata a se stessa .

Definizione (mutua ricorsione) : quando una procedura P chiama un'altra procedura Q che a sua volta chiama P .

il record di attivazione di una chiamata ricorsiva conterrà :

- valore attuale della procedura
- risultato intermedio : può essere determinato solo quando la i-esima chiamata ricorsiva termina
- indirizzo risultato

Definizione (ricorsione in coda) : sia f una funzione che nel suo corpo contiene la chiamata ad una funzione g (diversa da f oppure anche uguale ad f stessa) . La chiamata di g si dice “chiamata in coda” se la funzione restituisce il valore restituito da g senza dover fare alcuna ulteriore computazione . Diciamo che la funzione f ha la ricorsione in coda se tutte le chiamate presenti in f sono chiamate in coda

Ricorsione vs ricorsione in coda : la ricorsione in coda necessita di un solo record di attivazione quindi l'allocazione è possibile anche staticamente , perché al contrario della ricorsione normale non necessita di continuare a mantenere le informazioni presenti nel record di attivazione delle chiamate precedenti . In generale è sempre possibile trasformare una funzione ricorsiva in una ricorsiva in coda . La trasformazione di una funzione in una equivalente con la ricorsione in coda può essere fatta in modo automatico usando la tecnica

- “continuation passing style” : consiste nel rappresentare in un dato punto del programma mediante una funzione la parte rimanente del programma mediante una funzione detta continuazione

Ricorsione o iterazione : l'uso dell'uno o dell'altro metodo è dovuto oltre che alla predisposizione del programma anche alla natura del problema : cioè per l'uso di strutture dati rigide è più facile usare costrutti iterativi mentre dove si usano alberi,liste è spesso più naturale usare procedure ricorsive .

ASTRARRE SUL CONTROLLO

definizione (funzione) : è una porzione di codice identificata da un nome,dotata di ambiente locale proprio,capace di scambiare informazioni con il resto del codice mediante parametri .
Una funzione scambia informazioni con il resto del programma mediante tre meccanismi principali :

- parametri
- il valore di ritorno
- ambiente non locale

i parametri si distinguono in :

- **parametri formali** : che compaiono nella definizione di funzione
- **parametri attuali** : che compaiono invece nella chiamata

valore di ritorno : oltre ai parametri una funzione scambia informazioni anche restituendo un valore come risultato della funzione stessa . A volte si utilizza il termine funzione quando restituisce un valore altrimenti vengono chiamate procedure quei sottoprogrammi che interagiscono con il chiamante solo attraverso i parametri o l'ambiente non locale .

Ambiente non locale : si tratta del meccanismo meno sofisticato col quale una funzione scambia informazione con il resto del programma ,se il corpo della funzione modifica un ambiente non locale allora la modifica si ripercuote in tutte le porzioni del programma dove quella variabile è visibile .

Definizione (astrazione funzionale) : si ha una vera astrazione funzionale quando il cliente non dipende dal corpo di una funzione ma solo dalla sua intestazione .

PASSAGGIO DI PARAMETRI

I tipi di comunicazione permessa da un parametro sono :

- di ingresso : se permette una comunicazione solo unidirezionale dal chiamante al chiamato
- di uscita : se permette una comunicazione solo uno unidirezionale dal chiamato al chiamante
- di ingresso e di uscita : quando permette una comunicazione bidirezionale

passaggio per valore :

- modalità : parametro di ingresso
- ambiente : l'ambiente locale della procedura è esteso con un'associazione tra il parametro formale ed una nuova variabile

Al momento della chiamata l'attuale viene valutato e lo r-valore così ottenuto viene assegnato al parametro formale . Al termine della procedura il parametro formale viene distrutto come l'ambiente locale e non c'è alcun legame tra i parametro formale e l'attuale . Se il formale fa parte del record di attivazione della procedura allora si osservi che è una modalità costosa in quanto se abbiamo una struttura di grosse dimensioni bisogna copiarla nel formale però l'accesso al parametro formale è minimo dal momento che coincide all'accesso di una variabile locale .

passaggio per riferimento :

- modalità : parametro di ingresso e di uscita
- comunicazione : bidirezionale

il parametro attuale deve essere un'espressione dotata di l-valore , al momento della chiamata viene valutato lo l-valore dell'attuale (creando così una situazione di aliasing) . Al termine della

procedura il legame tra il formale e l-valore dell'attuale viene distrutto .
Nel modello di macchina astratta a pila, al formale viene associata una locazione nel record di attivazione della procedura , nella quale la sequenza di chiamata memorizza lo l-valore dell'attuale.
L'accesso al formale avviene attraverso questa locazione

- costo : molto contenuto,poiché al momento della chiamata si deve solo memorizzare un indirizzo

passaggio per costante :

- modalità : parametro di ingresso
- scopo : non dover copiare dati di grosse dimensioni

il parametro non riceve nessuna modifica all'interno della funzione,quindi è possibile mantenere la semantica del parametro per valore implementandola attraverso il passaggio per riferimento , questo fa si che la modalità si solo in lettura .

Passaggio per risultato :

- modalità : parametro di uscita

l'ambiente locale è esteso con una nuova associazione tra il parametro formale e una nuova variabile,il parametro attuale deve essere un'espressione dotata di l-valore .
Al momento della chiamata non vi è nessuna comunicazione,mentre subito prima della distruzione dell'ambiente locale,il valore corrente del formale viene assegnato alla locazione corrispondente al parametro attuale

Passaggio per valore-risultato :

- comunicazione : bidirezionale

il formale è una variabile locale alla procedura , l'attuale deve essere un espressione dotata di l-valore . Alla chiamata il parametro attuale viene valutato e lo r-valore così ottenuto viene assegnato al formale . Al termine della procedura subito prima della distruzione dell'ambiente il valore del formale viene assegnato alla locazione corrispondente al parametro attuale , durante l'esecuzione del corpo non vi è alcuna legame tra il formale e l'attuale

passaggio per nome :

- modalità : parametro di ingresso e di uscita
- regola di copia : sia una funzione di parametro formale x e sia a un'espressione compatibile con il tipo di x , allora una chiamata a f con parametro attuale a è semanticamente equivalente all'esecuzione del corpo di f nel quale tutte le occorrenze del parametro formale x sono sostituite con a

Non potendo impedire che vi siano più variabili distinte con lo stesso nome,si ottiene una sostituzione senza cattura richiedendo che il parametro formale anche dopo la sostituzione venga valutato nell'ambiente del chiamante e non del chiamato

implementazione : abbiamo la necessità per il chiamante di passare

- l'espressione testuale che costituisce il parametro attuale
- l'ambiente nel quale tale espressione deve essere valutata

Definizione (chiusura) : una coppia (espressione,ambiente) nella quale l'ambiente comprenda tutte le variabili libere (non locali) presenti nell'espressione

quando avviene il passaggio per nome quindi,il chiamante passa una chiusura costituita dal parametro attuale e dall'ambiente corrente . Quando il chiamato incontra un riferimento al parametro formale ,tale riferimento viene risolto mediante la valutazione della prima componente della chiusura nell'ambiente costituito dalla seconda componente

In una macchina astratta con allocazione a pila , un chiusura è costituita da una coppia di puntatori

- il primo punta al codice di valutazione dell'espressione che costituisce il parametro formale
- il secondo è un puntatore di catena statica che indica il blocco che costituisce l'ambiente locale nel quale valutare l'espressione

il parametro formale non è una variabile locale alla procedura e il parametro attuale può essere una generica espressione, tra questi due parametri è possibile che ci sia aliasing

- costo : molto costoso sopratutto per la necessità di passare una struttura complessa e la valutazione ripetuta del parametro attuale in un ambiente diverso da quello corrente

FUNZIONI DI ORDINE SUPERIORE

Definizione (funzione di ordine superiore) : quando una funzione ha3 come parametro o restituisce come risultato un'altra funzione .

Al momento dell'esecuzione di una funzione f invocata tramite un parametro formale h abbiamo due possibilità di ambiente non locale da utilizzare :

- utilizzare l'ambiente attivo al momento della creazione del legame tra h e f ,questa viene chiamato **deep binding**
- utilizzare l'ambiente attivo al momento della chiamata di f tramite h in questo caso adottiamo una politica **shallow binding**

implementazione del deep binding : lo shallow binding è facile da implementare,basta ricercare per ogni nome l'ultima associazione presente . Invece il deep binding richiede strutture dati ausiliarie rispetto all'ordinaria catena statica o dinamica . L'informazione relativa al puntatore di catena statica deve essere determinata al momento dell'associazione tra il parametro formale e il parametro attuale . Al parametro formale non solo deve essere associato il codice della funzione ma anche l'ambiente in cui deve essere valutato . Questo ambiente è determinato in corrispondenza di una chiamata alla funzione (es g (f)) ,possiamo associare staticamente al parametro l'informazione relativa al livello di annidamento della definizione della funzione rispetto al blocco nel quale compare la chiamata . Queste informazioni sono usate infine dalla macchina astratta sia per associare al parametro formale corrispondente alla funzione,sia il suo codice che il puntatore al record di attivazione del blocco all'interno del quale la funzione è dichiarata .
Quindi al parametro formale viene associato una coppia (testo , ambiente) rappresentata da una coppia (puntatore al codice , e puntatore al rda)

politiche di binding e scope statico : a prima vista può sembrare che shallow binding e deep binding non facciano alcuna differenza nel caso di scope statico . Ma non è così , la ragione è da ricercare sulla pila nel caso siano presenti più record di attivazione della stessa funzione . Se all'interno di una di queste attivazioni viene passata per parametro una procedura, le sole regole di scope non bastano per sapere quale ambiente non locale và utilizzato .

In conclusione per determinare un'ambiente sono necessarie :

- le regole di visibilità
- le eccezioni alle regole di visibilità
- le regole di scope
- le regole relative alla modalità di passaggio dei parametri
- la politica di binding

funzioni come risultato : quando una funzione restituisce una funzione come risultato, tale risultato è una chiusura . Quando viene invocato una funzione il cui valore è stato ottenuto dinamicamente, il puntatore di catena statica del rda viene determinato utilizzando la chiusura associata . Se è possibile restituire la funzione all'interno di un blocco annidato si può creare la possibilità che il suo ambiente di valutazione faccia riferimento a un nome che secondo la disciplina a pila andrebbe distrutto . Per ovviare a questo problema si abbandona la disciplina a pila e gli rda rimangono in vita indefinitivamente perché potrebbero costituire l'ambiente di funzione che verranno valutate solo in seguito , quindi gli ambienti hanno tempo di vita illimitato . Un altra soluzione è quella di allocare gli rda sullo heap e lasciare al garbage collector la possibilità di deallocarli

definizione (eccezioni) : è un evento particolare che si verifica durante l'esecuzione di un programma e non deve essere gestito dal normale flusso del controllo , si può trattare di errori di semantica dinamica (divisione per zero,overflow...) oppure è il programmatore che decide di terminare esplicitamente la computazione corrente e trasferire il controllo in un altro punto del programma

per gestire correttamente le eccezioni un linguaggio deve :

- specificare quali eccezioni sono gestibili
- specificare come un'eccezione può essere sollevata
- specificare come un'eccezione possa essere gestita

quando un'eccezione è un tipo qualsiasi in genere essa può contenere un valore generato dinamicamente che poi viene passato al gestore . Una volta definita un'eccezione essa può essere sollevata implicitamente se si tratta di un'eccezione della macchina astratta oppure esplicitamente dal programmatore . Per la gestione di un'eccezione ha bisogno di due costrutti :

- un meccanismo per definire una capsula attorno la porzione di codice con lo scopo di intercettare le eccezioni
- la definizioni di un gestore dell'eccezione

aspetti principali delle eccezioni :

- l'eccezione non è un evento anomalo ma ha un nome che viene esplicitamente menzionato nel costrutto throw di java . E che viene usato dai costrutti try-catch per intrappolare una specifica classe di eccezioni
- l'eccezione potrebbe inglobare un valore usato dal costrutto che solleva l'eccezione . Il valore viene passato al gestore come argomento su cui reagire all'avvenuta eccezione

se si verifica un eccezione all'interno di una procedura : se l'eccezione non è gestita all'interno della procedura occorre terminare la procedura e risollevare l'eccezione dal punto in cui è stata invocata,se l'eccezione non è neppure gestita dal chiamante allora bisogna risalire lungo la catena delle chiamate di procedura fino a quando non si incontra un gestore compatibile altrimenti c'è il gestore di default . Le eccezioni quindi si propagano lungo la catena dinamica .

implementazione :

- un modo intuitivo è quello di sfruttare la pila degli rda . Ogni volta che che si entra in un blocco protetto , nel rda viene inserito un puntatore al gestore corrispondente . Quando si esce da un blocco protetto viene tolto dalla pila il riferimento al gestore . Quando viene sollevata un'eccezione la macchina astratta cerca un gestore per l'eccezione nel rda,se non lo trova toglie il rda dalla pila e risolleva l'eccezione , questa tecnica ha un unico difetto la macchina astratta deve manipolare la pila

- ad ogni procedura viene associato un blocco protetto nascosto costituito dal corpo della procedura e il cui gestore è responsabile solo del ripristino dello stato e del risollevamento dell'eccezione immutata . Il compilatore poi prepara una tabella in cui per ogni blocco protetto (inclusi quelli nascosti) inserisce due indirizzi (ip,ig) che corrispondono all'inizio del blocco protetto e all'inizio del corrispondente gestore . Quando viene sollevata un eccezione si cerca nella tabella una coppia di indirizzi,questa ricerca permette determinare un gestore dell'eccezione . Questa tecnica è più costosa della precedente nel caso in cui si verifichi un'eccezione ma non costa nulla all'ingresso e uscita di un blocco protetto . Però siccome è raro il verificarsi dell'eccezione la tecnica è più efficiente

STRUTTURARE I DATI

i tipi di dato sono presenti nei linguaggi di programmazione per almeno tre scopi diversi :

- a livello di progetto , come supporto all'organizzazione concettuale
- a livello di programma, come supporto alla correttezza
- a livello di traduzione, come supporto all'implementazione

definizione (tipo di dato) : un tipo di dato è una collezione di valori omogenei (numeri interi o un intervallo di interi) ed effettivamente presentati , dotata di un insieme di operazioni che manipolano tali valori .

I tipi di dato hanno diversi scopi :

- supporto all'organizzazione concettuale : l'uso di tipi distinti serve al progettista per utilizzare il tipo più adatto per ogni classe di concetti
- correttezza : ogni linguaggio ha delle regole di controllo dei tipi , che disciplinano come i tipi possono essere usati in un programma . Questi vincoli sono presenti sia per evitare errori hardware a runtime e sia per evitare errori logici . Molti linguaggi verificano che i vincoli di tipo siano tutti soddisfatti prima di procedere all'esecuzione , questo è il ruolo del controllore dei tipi .
- Implementazione : i tipi costituiscono un importante informazione per la macchina astratta :
    1. dimensione della memoria da allocare
    2. accesso ad una variabile allocata avviene tramite offset e non per ricerca a tempo di esecuzione

ogni linguaggio ha un proprio sistema di tipi ossia il complesso delle informazioni e delle regole che governano i tipi di quel linguaggio . Un sistema è costituito da :

- l'insieme di tipi predefiniti dal linguaggio
- i meccanismi che permettono di definire nuovi tipi
- i meccanismi relativi al controllo dei tipi
- la specifica se i vincoli sono da controllare staticamente o dinamicamente

definizione (type safe) : quando nessuno programma può violare le distinzioni tra tipi definite in quel linguaggio

a seconda del linguaggio di programmazione possiamo classificare i tipi a seconda di come i suoi valori vengano manipolati :

- denotabili : se possono essere associati ad un nome
- esprimibili : se possono essere il risultato di un'espressione complessa
- memorizzabili : se possono essere memorizzati in una variabile

definizione (tipizzazione statica) : i controlli dei vincoli di tipizzazione possono essere condotti a tempo di compilazione sul testo del programma , in questo modo gli errori sono rilevati prima che il programma venga consegnato all'utente finale
definizione (tipizzazione dinamica) : i controlli avvengono a tempo d'esecuzione ,in questo caso se si verifica un errore il programma potrebbe già essere nelle mani dell'utente finale

il controllo dei tipi prevede che ogni oggetto possieda un descrittore a run-time che ne specifica il tipo, la macchina astratta è responsabile di controllare che ogni operazione sia applicata ad operandi del tipo corretti .

Controlli statici vs controlli dinamici : Una macchina astratta con controllo statico è più difficile da realizzare rispetto a quella con controllo dinamico specialmente se si vuole garantire la sicurezza dei tipi , però la compilazione viene fatta solo poche volte quindi anche se lenta non importa , un'ultima differenza risiede nel caso che alcuni controlli statici possono decretare come errati programmi che in realtà non causano un errore a tempo d'esecuzione

i tipi base di un linguaggio sono :

- scalari : sono tutti quei i tipi i cui valori non sono costituiti da aggregati di altri valori
- booleani : possono assumere solo i valori true false. Sono comandati da operatori logici e condizionali . Occupano un solo byte
- caratteri : possono assumere l'insieme dei caratteri ASCII e unicode . Sono comandati da operatori quali uguaglianza , la decodifica ecc occupano un solo byte per gli ASCII e due per gli unicode
- interi : possono assumere l'insieme dei valori numerici interi compresi nel range di possibilità dell'architettura . Sono comandati da operatori quali il + , - ecc . Occupano da 2 a 4 byte in base alla loro rappresentazione
- reali : possono assumere l'insieme dei razionali compresi nel range di possibilità dell'architettura . Hanno operatori identici agli interi e sono rappresentati da 4 byte
- virgola fissa : un sottoinsieme dei numeri razionali,la struttura di tale sottoinsieme dipende dalla rappresentazione adottata . Gli operatori sono quelli tradizionali
- complessi : sottoinsieme dei numeri complessi
- void : tipo primitivo la cui semantica è quella di avere un solo valore
- enumerazioni : consiste in un insieme finito di costanti ciascuna caratterizzata dal proprio nome , essi sono tipi scalari definiti dall'utente
- intervalli : costituiscono un sottoinsieme contiguo dei valori di un altro tipo scalare
- ordinali : contengono quelle tipologie che possono essere ordinate cioè quei tipi per cui ha senso chiedersi quale sia il successivo o il precedente . Sono gli interi,caratteri e anche i booleani

i tipi non scalari sono detti composti cioè che si ottengono combinando tra loro altri tipi mediante l'utilizzo di opportuni costruttori :

- record : è una collezione costituita da un numero finito di elementi detti campi distinti dal loro nome,ciascun campo può essere un tipo diverso dagli altri . Nella maggioranza dei casi ogni campo si comporta come una variabile del proprio tipo
- record varianti o unioni : una forma particolare di record è quella in cui alcuni campi sono tra loro mutualmente esclusivi , per quanto riguarda le varianti esse condividono la stessa zona di memoria visto che non possono essere attive contemporaneamente , risparmiando così spazio
- unioni : in c non esiste il concetto di record variante ma quello di unione . Essa è analoga ad un record nella definizione dei campi ma con la differenza che uno dei campi è attivo in un

qualsiasi momento

- array : è una collezione finita di elementi dello stesso tipo indicizzata su un intervallo di un tipo ordinale (da un punto di vista semantico l'array è una funzione che ha come dominio l'intervallo degli indici e come codominio il tipo degli elementi degli array
  - *operazioni* : quella più semplice è la selezione di un elemento che si ottiene mediante un valore dell'indice altri linguaggi permettono assegnamento,uguaglianza confronti ed anche operazioni aritmetiche
  - *controlli* : ogni accesso ad un elemento del vettore deve avvenire entro i limiti dell'array e che non si tenti di accedere ad elementi che non esistono , quindi un linguaggio sicuro dovrà far si che il compilatore generi opportuni controlli in corrispondenza di ogni accesso . Uno degli attacchi più comuni è il buffer overflow
  - *memorizzazione e calcolo degli indici* : un array è memorizzato 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 . L'accesso ad un particolare valore dell'array può avvenire in vari modi , solitamente si crea un offset a partire dall'indirizzo di partenza del vettore : " dimensione elemento * indice " nel caso di array monodimensionali invece nel caso di matrici "dimensione elemento * indice + dimensione riga * secondo indice " .
  - allocazione : per la memorizzazione di un array abbiamo tre forme diverse :
    - *forma statica* : il tutto è deciso a tempo di compilazione e l'array può essere memorizzato nel rda del blocco in cui compare la sua definizione
    - *forma fissata al momento dell'elaborazione della dichiarazione* : la forma è nota al momento in cui il controllo raggiunge la dichiarazione dell'array . Anche in questo caso l'array può essere allocato nell'rda del blocco in cui compare la sua definizione ma il compilatore non ha modo di riconoscere l'offset tra l'inizio dell'array e il punto fissato dell'rda cui si riferisce il frame pointer in questo caso l'rda viene diviso in due parti, una di lunghezza fissa e l'altra di lunghezza variabile , nella parte a lunghezza fissa si accede per offset invece nell'altra parte si accede per accesso indiretto attraverso un descrittore dei dati a lunghezza variabile . Il descrittore (dope vector) è contenuto nella parte a lunghezza fissa in modo che ad esso si acceda per offset
    - *forma dinamica* : in questo caso l'array può cambiare forma dopo la sua creazione per effetto dell'esecuzione , esso verrà memorizzato sullo heap mentre un puntatore verrà memorizzato nella parte di lunghezza fissa dell'rda

dope vector : esso contiene :

- un puntatore alla prima locazione nella quale è memorizzato l'array
- tutte le informazioni dinamiche necessarie al calcolo della quantità , cioè il numero di dimensioni (detto rango dell'array) , il valore del limite inferiore di ogni dimensione , l'occupazione di ogni dimensione . Per accedere ad un elemento dall'array si accede (per offset mediante il frame pointer) al dopo vector , si calcola l'espressione e la si somma all'indirizzo dell'array

- insiemi : i valori del tipo insieme sono costituiti da sottoinsiemi di un tipo base usualmente ristretto ad essere un tipo ordinale
- puntatori : alcuni linguaggi permettono di manipolare direttamente gli l-valori , il tipo corrispondente è detto tipo dei puntatori . La variabile in sé è sempre un riferimento ossia è sempre considerata come un l-valore anche se tale valore non può essere modificato esplicitamente . Invece i puntatori forniscono un modo per riferirsi ad un l-valore senza deferenziarlo (accesso all'oggetto) automaticamente . Sui puntatori è possibile effettuare alcune operazioni aritmetiche come incrementare un puntatore sommare una quantità

arbitraria ad un puntatore , quindi appare evidente che l'aritmetica dei puntatori distrugge qualsiasi ambizione di type safeness di un linguaggio : cioè non vi è più garanzia che un puntatore di tipo t punti ancora ad un'area di memoria dove è memorizzato un valore tipo t . la deallocazione della memoria può essere :

- *esplicita* : il linguaggio mette a disposizioni meccanismi di recupero della memoria . Il problema più rilevante che si verifica in questi casi sono le dangling reference cioè puntatori diversi da null che puntano ad informazioni non più significative . Se il linguaggio è di tipo permette le dangling reference allora non è più type safe
- *implicita* :il linguaggio non fornisce alcuno strumento per deallocare memoria , il programmatore chiede allocazioni fin quando c'è spazio sullo heap quando è finito non è detto che la computazione debba abortire infatti è possibile dotare la macchina astratta con meccanismi di recupero della memoria queste tecniche vanno sotto il nome di gargabe collector

- tipi ricorsivi : è un tipo composto nel quale un valore del tipo può contenere un riferimento a un valore dello stesso tipo . Questi tipi sono rappresentati di solito come strutture dati sullo heap . Un valore di un tipo ricorsivo corrisponde ad 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 . essi possono essere esprimibili e sono dinamicamente allocati ma i linguaggi che permettono questo tipi non permettono in genere la deallocazione esplicita dei valori creati
- funzioni : tutti i linguaggi di alto livello permettono di definire funzioni ma pochi permettono di denotare il tipo delle funzioni e raramente sono esprimibili o memorizzabili . L'operazione principale ammessa su un valore di tipo funzione è l'applicazione cioè l'invocazione di una funzione su alcuni argomenti

equivalenza : le regole definiscono tra i tipi una relazione di equivalenza se due tipi sono equivalenti ossia ogni espressione o valore di un tipo è anche espressione o valore nell'altro tipo . Esistono due regole di equivalenza tra tipi :

- equivalenza per nome : due tipi sono equivalenti per nome se hanno lo stesso nome cioè un tipo è equivalente solo a se stesso . In questo modo si avranno due nomi distinti per lo stesso tipo
- equivalenza strutturale : l'equivalenza strutturale fra tipi è la minima relazione d'equivalenza che soddisfa le seguenti tre proprietà
  - un tipo è equivalente a se stesso
  - se un tipo t è introdotto con una definizione type t = espressione , t è equivalente a espressione
  - se due tipi sono costruiti applicando lo stesso costruttore di tipo a tipi equivalenti, allora essi stessi sono equivalenti

  in questo caso si dà importanza alla struttura e non al nome che si dà al tipo. Il controllore dei tipi non riesce a risolvere la mutua ricorsione .

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 . In molti linguaggi è proprio la compatibilità a regolare la correttezza di un assegnamento , quella del passaggio dei parametri (il parametro attuale deve essere compatibile con il formale) .
un tipo T può essere compatibile con S quando :

- T e S sono equivalenti
- i valori di T sono sottoinsieme dei valori di S : è il caso di un tipo intervallo
- tutte le operazioni sui valori di S sono possibili anche sui valori di T

- i valori di tipo T corrispondono in modo canonico ad alcuni valori di S : int con float
- i valori di T possono essere fatti corrispondere ad alcuni valori di tipo S : conversione

abbiamo quindi due tipi di conversione di tipo :

- *conversione implicita o coercizione* : quando è la macchina astratta ad inserire tale conversione, senza che ve ne sia traccia a livello linguistico
- *conversione esplicita o cast* : quando la conversione è indicata nel testo del programma

coercizioni : in presenza di compatibilità tra il tipo T e il tipo S, il linguaggio permette la presenza di un valore di T laddove sarebbe richiesto un valore di S . in questa situazione il compilatore e/o la macchina astratta inseriscono una conversione implicita di tipo da T a S .
a seconda della nozione di compatibilità usata una coercizione può corrispondere a cose distinte :

- T è compatibile con S e condividono la rappresentazione in memoria : la coercizione rimane a livello sintattico e non genera codice
- T è compatibile con S perché esiste un modo canonico per trasformare i valori di T in S : nel caso di int e float la macchina astratta inserisce codice per trasformare a run time la rappresentazione in complemento a due in quella in virgola mobile
- la compatibilità di T e S è stabilita in virtù di una corrispondenza arbitraria tra i valori di T e S : la macchina inserisce codice . In questo caso non trasforma la rappresentazione ma la controlla dinamicamente

conversioni esplicite : sono annotazioni nel linguaggio che specificano che un valore di un tipo deve essere convertito in un altro tipo . Non tutte le conversioni sono consentite ma solo quelle per le quali il linguaggio conosce come implementare la conversione quindi è evidente che si può inserire un cast solo dove c'è compatibilità .

POLIMORFISMO

definizione (monomorfo) : in un sistema di tipi nel quale ogni oggetto (valore , funzione ecc) del linguaggio ha un unico tipo .

Definizione (polimorfo) : un sistema di tipi nel quale uno stesso oggetto può avere più di un tipo. Per analogia diremo che un oggetto è polimorfo quando il sistema di tipi gli assegna più di un tipo

ci sono due forme di polimorfismo :

- polimorfismo ad hoc detto anche overloading
- polimorfismo universale :
  - parametrico
  - sottotipo o inclusione

overloading : è in realtà polimorfismo solo in apparenza . Un nome è overloaded quando ad esso corrispondono più oggetti . Un impiego di questo meccanismo è di definire più funzioni (o costruttori) con lo stesso nome ma con tipi di parametri diversi . Questa situazione viene risolta staticamente utilizzando l'informazione presente nel contesto , cioè viene fatta una sorta di pre-analisi del programma e risolve i casi di overloading e sostituisce ogni simbolo sovraccaricato con un nome non ambiguo che denota univocamente un solo oggetto . Non si confonda le coercizioni con l'overloading , dato che corrispondono a problemi distinti

polimorfismo vs overloading : l'overloading ci permette di usare lo stesso nome per algoritmi diversi per tipi di dati diversi mentre il polimorfismo ci permette di usare lo stesso algoritmo per tipi di dati diversi

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 (perchè le informazioni di tipo non sono utilizzate nell'algoritmo che la funzione implementa) . Quindi una funzione del genere è dunque costituita da un unico codice che lavora uniformemente su tutte le istanze del suo tipo generale .
Polimorfismo universale Vs overloading : in un linguaggio senza polimorfismo dovremmo scrivere la stessa funzione per qualsiasi tipo di variabile

il polimorfismo parametrico è il più flessibile e generale ed è presente nei linguaggi di programmazione in due forme distinte che sono dette :

- polimorfismo esplicito : nel programma sono presenti esplicite annotazioni (parametri generici) che indicano quali tipi devono essere considerati come parametri
- polimorfismo implicito : le variabili di tipo non appaiono nel programma ma vengono generate secondo necessità dall'algoritmo di inferenza di tipo ; sono sempre istanziate automaticamente

polimorfismo universale 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 . Anche in questo caso uno stesso oggetto ha un infinità di tipi diversi che si ottengono per istanziazione da uno schema di tipo più generale . Però non tutte le possibili istanziazioni dello schema di tipo più generale sono ammissibili ma dobbiamo limitarci a quelle definite da una qualche nozione di compatibilità "strutturale"tra i tipi . Assumiamo che tra i linguaggi sia definita una relazione di sottotipo che indichiamo con il simbolo "< : " leggendo come c <: d , c è sottotipo di d

implementazione del polimorfismo : un primo modo di gestire il polimorfismo è quello di risolverlo a tempo di linking (quando si collegano i files oggetto rilocabili generati dal compilatore, si associa a ogni chiamata del sottoprogramma polimorfo l'istanza richiesta in base al tipo dei parametri attuali) . Si generano diverse istanze del codice del sottoprogramma polimorfo . A tempo di linking quindi vengono identificate le funzioni polimorfe , il loro codice viene modificato per tener conto dei tipo con cui sono chiamate e il codice risultante viene collegato tutto assieme , si osservi che esistono nel codice eseguibile più copie dello stesso template una per ogni istanza , l'uso o non uso dei template è relativo dato che a tempo di compilazione non esistono più .
Nel caso di linguaggi con polimorfismo viene generata un'unica versione del codice di una funzione polimorfa ed è proprio quel codice che viene eseguito quando serve un'istanza . Se le informazioni dipendono dal tipo è opportuno cambiare la rappresentazione dei dati , al posto memorizzare nell'rda il dato si usa un puntatore al dato stesso .

CONTROLLO E INFERENZA DI TIPO

nel caso di un linguaggio con controlli statici il controllore (**type checker**) è un modulo del compilatore nel caso di controlli dinamici invece è un modulo del supporto a run-time . Per ottenere il suo scopo il controllore deve determinare il tipo delle espressioni presenti nel programma sfruttando le informazioni di tipo che il programmatore ha esplicitamente inserito in alcuni punti critici quali le dichiarazioni di nomi , parametri e cast . Per determinare il tipo delle espressioni complesse il controllore esegue una semplice visita dell'albero sintattico del programma a partire dalle foglie (variabili di cui riconosce il tipo) risalendo verso la radice calcolando il tipo delle espressioni composte a partire dalle informazioni del programmatore .

Al posto del semplice controllo dei tipi alcuni linguaggi adottano un procedimento più sofisticato che si chiama **inferenza di tipo** . Essa è un processo di attribuzione del tipo ad un'espressione nella

quale non compaiono esplicite dichiarazioni di tipo per i suoi componenti . Come per il type checker anche qui si lavora sull'albero sintattico a partire dalle foglie ma può capitare che in qualche espressione atomica non sia possibile determinare un tipo specifico , a questo punto l'algoritmo di tipo assegna una variabile di tipo . Sfruttando le informazioni del contesto vengono collezionati alcuni vincoli sulle variabili di tipo .

L'algoritmo procede nel seguente modo :

- assegna un tipo per ogni nodo dell'albero sintattico : per i nomi predefiniti usa il tipo indicato nella tabella dei simboli , per gli identificatori usa una variabile di tipo
- risale l'albero sintattico generando un vincolo (di uguaglianza) tra tipi in corrispondenza ad ogni nodo interno
- risolve i vincoli così raccolti usando l'algoritmo di unificazione

SICUREZZA

possiamo distinguere i diversi linguaggi di programmazione relativamente alla sicurezza in :

- **linguaggi non sicuri :** quando il linguaggio ci permette di aggirare i controlli di tipo, permette di accedere alla rappresentazione di un tipo di dato e quando ci permette di accedere al valore di un puntatore (es. c , c++)
- **linguaggi localmente non sicuri :** quando il sistema di tipi è bel regolato e i tipi controllati ma ci sono specifici costrutti che permettono di scrivere programmi non sicuri . La locale non sicurezza dipende anche dalla presenza di unioni e dalla deallocazione esplicita della memoria (dangling reference)
- **linguaggi sicuri** : un teorema garantisce la type safeness

come evitare le dangling reference :

- **tombstone** : mediante questo la macchina astratta è in grado di segnalare ogni tentativo di deferenziare una dangling reference . Ogni volta che viene allocato un nuovo oggetto sullo heap cui si accede per puntatore , la macchina astratta alloca anche un'ulteriore parola di memoria (tombstone) . in modo analogo la tombstone viene allocata anche tutte le volte che viene creato un puntatore che si riferisce alla pila . La tombstone viene inizializzata all'indirizzo dell'oggetto allocato mentre il puntatore riceve l'indirizzo della tombstone . Alla deallocazione di un oggetto la tombstone viene invalidata con un valore particolare .
  - *Svantaggi* : allocazione e controllo delle tombstone,lo spazio . Inoltre le tombstone invalidate rimangono sempre in memoria , per ovviare a questo problema è possibile riusarle mediante un piccolo garbage collector
- **lucchetti e chiavi** : esso risolve il problema dell'accumulo delle tombstone in memoria , ogni volta che viene creato un oggetto sullo heap viene associato all'oggetto anche un lucchetto cioè una parola di memoria nella quale viene memorizzata un valore casuale . Il puntatore in questo caso è costituito dall'indirizzo vero e proprio e una chiave cioè una parola di memoria che viene inizializzata al valore del lucchetto corrispondente all'oggetto puntato . Quando un puntatore viene assegnato ad un altro viene assegnato tutta la coppia . Al momento della deallocazione , l'oggetto viene deallocato e il lucchetto viene annullato così che tutte le chiavi che prima lo aprivano ora causano errore
  - *svantaggi* : in termini di spazio costano più delle tombstone visto che è necessaria una parola di memoria per ogni puntatore , il costo della creazione , l'assegnamento di un puntatore e quello necessario al controllo se la chiave apre il lucchetto

GARBAGE COLLECTOR

nei linguaggi senza deallocazione esplicita si rende necessario dotare la macchina astratta di un

meccanismo con il quale si possa recuperare memoria sullo heap : il garbage collector .

Il funzionamento del garbage collector si divide in due fasi :

- distinguere gli oggetti vivi da quelli non più utilizzati
- recuperare gli oggetti non più utilizzati così che il programma li possa riutilizzare

si possono classificare i garbage collector classici a seconda di come determinano gli oggetti non più in uso :

- *contatori dei riferimenti*
- *mark*
- *copy*

contatore dei riferimenti : quando un oggetto non è più utilizzato vuol dire che non ci sono più puntatori verso di esso , questa tecnica è il metodo più elementare per costruire un garbage collector. Al momento della creazione dell'oggetto viene allocato insieme ad esso un intero . La macchina astratta si incarica di far si che per ogni oggetto a run-time tale contatore contenga il numero di puntatori attivi a quell'oggetto .

- Add : in corrispondenza di un assegnamento il contatore dell'oggetto viene incrementato di uno mentre l'altro viene decrementato di uno .
- Sub : Quando si esce da un ambiente locale sono decrementati di uno tutti i contatori degli oggetti puntati da puntatori locali a quell'ambiente
- Del : quando un contatore raggiunge valore 0 allora l'oggetto può essere deallocato

vantaggio : controllo e recupero sono mescolati con il normale funzionamento del programma . Con poche modifiche è possibile adottare questa tecnica in real time

svantaggio : deallocazione delle strutture circolari , i contatori dei riferimenti sono abbastanza inefficienti , in quanto hanno un costo proporzionale al lavoro complessivo compiuto dal programma

mark and sweep : la tecnica prende il nome da queste due fasi che si dividono :

- mark : per riconoscere cosa è inutilizzato si attraversano tutti gli oggetti dello heap, marcandoli come inutilizzati poi partendo dai puntatori attivi sulla pila (root set) si attraversano tutte strutture dati presenti sullo heap marcando come “in uso” ogni oggetto attraversato
- sweep : tutti i blocchi come “in uso “ sono lasciati immutati invece gli altri sono restituiti alla lista dei blocchi liberi

mark and sweep vs contatore di riferimenti : esso non è incrementale , quindi verrà invocato dalla macchina astratta quando la memoria libera disponibile sullo heap è prossima ad esaurirsi

svantaggi :

- *frammentazione esterna* : oggetti vivi e non più in uso sono intercalati nello heap così da rendere l'allocazione di un oggetto di grandi dimensioni impossibile
- *efficienza* : richiede tempo proporzionale in base alla dimensione dello heap
- *località dei riferimenti* : gli oggetti in uso rimangono al loro posto , mentre è possibile che oggetti contigui siano stati recuperati e al loro posto allocati nuovi oggetti . Quindi troveremo oggetti di età diverse che diminuisce la località di riferimenti degradando la performance in presenza di gerarchie di memoria

intermezzo (rovesciamento dei puntatori): per visitare una struttura concatenata occorre marcare

un nodo e visitare ricorsivamente le sottostrutture i cui puntatori siano parte del nodo . In questo schema ricorsivo bisogna memorizzare sulla pila l'indirizzo del blocco appena visitato così da poter tornare indietro quando si è arrivati al termine della sottostruttura . Arrivati a questo punto si ripristina il puntatore al suo valore originale cosicché al termine della visita della struttura è esattamente nella stessa situazione iniziale .

Mark and compact : per ovviare alla frammentazione possiamo modificare la fase di sweep in quella di compattamento : gli oggetti vivi sono spostati in modo da renderli contigui e lasciare tutta la memoria libera in un unico blocco contiguo .

- Compattazione : può avvenire traslando ogni blocco vivo che si incontra

svantaggi : il compattamento da solo necessita di due o tre passaggi il primo per calcolare la nuova posizione che ogni blocco vivo assumerà,il secondo consiste nell'aggiornare i puntatori interni agli oggetti , un terzo per spostare gli oggetti.

Vantaggi : è una buona soluzione per la frammentazione e permette la gestione della lista libera come un unico blocco

Stop and copy : lo heap è diviso in due parti di dimensioni uguali . Durante l'esecuzione normale solo uno dei due semispazi è in uso , la memoria è allocata ad un estremità del semispazio mentre la memoria libera consiste in un unico blocco contiguo .

Vantaggi : l'allocazione è efficiente e non c'è frammentazione, è efficiente se i due semispazi sono grandi

quando la memoria del semispazio è finita , viene invocato il garbage collector . Questi a partire dal root set inizia una visita di tutte le strutture concatenate e le copia nell'altro semispazio compattandole ad un'estremità di questo . Al termine del processo il ruolo dei due semispazi viene invertito

ASTRARRE SUI DATI

da un punto di vista generale dunque , mentre il linguaggio fornisce l'astrazione di tipi che nascondono l'implementazione lo stesso non è possibile per il programmatore . Per ovviare a questo problema alcuni linguaggi di programmazione permettono di definire delle astrazioni sui dati che si comportano come quelli predefiniti rispetto all'inaccessibilità della rappresentazione .

Questo meccanismo che chiamiamo **tipo di dato astratto** è caratterizzato da :

- un nome per il tipo
- un implementazione o rappresentazione per tale tipo
- un insieme di nomi di operazioni per la manipolazione dei valori di quel tipo con i loro tipi
- per ogni operazione un'implementazione che usi la rappresentazione fornita al punto 2
- una capsula di sicurezza che separi i nomi del tipo e delle operazioni dalle loro realizzazioni

un adt dunque è una capsula opaca :

- **sulla superficie esterna** : è visibile a chiunque , dove troviamo il nome del tipo , il nome e il tipo delle operazioni ; al suo interno invisibili da fuori ci sono l'implementazione del tipo e delle operazioni .
- **La superficie esterna** : si chiama signature o interfaccia dell'adt al suo interno c'è implementazione .

La distinzione tra interfaccia e implementazione è molto importante permette di separare l'uso di una componente dalla sua definizione . Nell 'astrazione sui dati non viene nascosto solo come una certa operazione è realizzata , ma anche le modalità di rappresentazione dei dati in modo tale che il linguaggio possa garantire che l'astrazione non venga violata questo fenomeno viene indicato come occultamento dell'informazione (information hiding) .
E' possibile sostituire l'implementazione di un adt con un'altra senza modificarne l'interfaccia .

Definizione (specifica) : la descrizione della semantica delle operazioni di un adt espressa non in termini del tipo concreto , ma per mezzo di relazioni generali astratte

definizione (indipendenza dalla rappresentazione) : due implementazioni corrette di una stessa specifica di un adt sono indistinguibili da parte dei clienti di quel tipo . Quindi se un tipo gode di questa proprietà è possibile modificare la sua implementazione con una equivalente e forse più efficiente senza che ciò provochi problemi nei clienti la comparsa di (nuovi) errori .

Moduli : gli adt fanno parte della programmazione in piccolo , e sono pensati per un tipo con le relative operazioni . È molto più comune un'astrazione composta da più tipi (o strutture dati) tra loro correlate , nelle quali si vuole dare ai clienti una versione limitata . I meccanismi che realizzano questo tipo di incapsulamento sono i moduli o package .

- Scopo : permette di partizionare staticamente un programma in parti distinte ciascuna delle quali sia dotata sia di dati che di operazioni
- differenza : non c'è grande differenza con gli adt se non la possibilità di definire più tipi contemporaneamente (l'adt è un caso particolare di un modulo)
- vantaggi : questo meccanismo fornisce molta flessibilità sia nella definizione del grado di permeabilità delle capsula (è possibile indicare in modo individuale quali operazioni siano o meno visibili all'esterno e selezionare il grado di visibilità)

il modulo è suddiviso :

- come un adt, in una **parte pubblica** visibile a tutti gli utenti : un modulo può menzionare alcune delle proprie strutture dati nella parte pubblica cosicché chiunque può usarle o modificarle
- **una parte privata** invisibile dall'esterno : essa può contenere dichiarazioni non menzionate affatto nella sua parte pubblica

PROGRAMMAZIONE AD OGGETTI

limiti degli adt : sono un meccanismo che garantisce in modo pulito ed efficace l'incapsulamento e l'occultamento dell'informazione . In particolare riunisce in un unico costrutto sia i dati che i modi legali per manipolarli

- svantaggi : tutto questo è ottenuto con una certa rigidità d'uso che si manifesta quando si vuole riusare o estendere un'astrazione

principali operazioni possibili degli adt :

- permettono l'incapsulamento e l'occultamento dell'informazione
- siano dotati di un meccanismo che sotto certe condizioni permette di ereditare operazioni da altri costrutti analoghi
- siano inquadrati in una nozione di compatibilità definita in termini delle operazioni ammissibili su un certo costrutto
- permettono la selezione dinamica delle operazioni in funzione del tipo effettivo (o dell'implementazione) dell'argomento cui vengono applicate .

Oggetti : è una capsula contenente sia dati che operazioni per manipolarli e che fornisce all'esterno un interfaccia attraverso la quale l'oggetto è accessibile .

- Differenza con gli adt : sebbene nella definizione di un adt dati e operazioni stanno assieme , quando dichiariamo una variabile di un tipo di dato astratto quella variabile rappresenta solo i dati che possiamo manipolare tramite le operazioni . Invece ciascun oggetto è un contenitori che incapsula sia dati che operazioni

le operazioni sono chiamate metodi e possono accedere ai dati contenuti nell'oggetto , che sono detti variabili di istanza . L'esecuzione avviene mandando un messaggio che consiste del nome del metodo da eseguire e i suoi parametri . L'invocazione di un metodo quindi comporta una selezione dinamica di quale metodo deve essere eseguito mentre l'accesso ai dati è statico .
Insieme agli oggetti i linguaggi mettono a disposizione meccanismi di organizzazione degli oggetti stessi che permettono di raggruppare gli oggetti con la stessa struttura .

Classi : è un modello di un insieme di oggetti , stabilisce quali sono i suoi dati (quanti , di quale tipo , con quale visibilità) e fissa nome, segnatura , visibilità e implementazione dei suoi metodi .
In un linguaggio con classi , ogni oggetto appartiene ad (almeno) una classe .

- Creazione di un oggetto : essi sono creati dinamicamente mediante istanziazione di una classe : viene allocato uno specifico oggetto con la struttura stabilita dalla definizione della classe

in c++ e java la classe corrisponde ad un tipo e tutti gli oggetti istanza di una classe A sono valori di tipo A . possiamo quindi supporre che il codice dei metodi sia memorizzato una sola volta nella classe e che quando ad un oggetto viene richiesto di eseguire un determinato metodo , questo venga ricercato nella classe di cui è istanza .

Definizione (this) : è un modo per riferirsi alle variabili di istanza , nel caso di un invocazione del metodo tramite this , il legame tra questo nome e l'oggetto cui si riferisce è determinato solo dinamicamente

allocazione degli oggetti : tutti i linguaggi orientati agli oggetti permettono di creare oggetti dinamicamente , dove tali oggetti vengono creati dipende dal programma .

- Heap : La soluzione più comune è quella di allocare un oggetto sullo heap e di accedere all'oggetto tramite un riferimento . Alcuni linguaggi permettono la deallocazione esplicita degli oggetti sullo heap altri invece optano per il garbage collector .
- Pila : quando viene elaborata la dichiarazione di una variabile di tipo classe viene creato e inizializzato un oggetto di quel tipo che viene assegnato come valore di quella variabile . La creazione e deallocazione avviene implicitamente

incapsulamento : ogni linguaggio permette di definire un oggetto nascondendone una parte . Di ogni classe abbiamo due viste :

- privata : qui tutto è possibile è la forma di accesso possibile all'interno della classe stessa (da parte dei suoi metodi)
- pubblica : qui sono visibili sono le informazioni esplicitamente esportate , chiamiamo interfaccia di una classe il complesso delle informazioni pubbliche .

Differenze : questa forma di incapsulamento con gli oggetti è assai più flessibile e sopratutto più estendibile di quello possibile con gli adt

tipo associato : ad un classe può essere fatto corrispondere in modo naturale l'insieme degli oggetti che sono istanza di quella classe : quest'insieme di oggetti è il tipo associato a quella classe .

Definizione (sottotipo) : il tipo associato alla classe T è un sottotipo di S quando ogni messaggio compreso di S è compreso anche dagli oggetti di T . Se rappresentiamo gli oggetti come record che contiene funzioni e dati , questa relazione di sottotipo corrisponde al fatto che T è un tipo di record che contiene i campi di S più eventualmente altri quindi l'interfaccia di S è un sottoinsieme dell'interfaccia di T

ridefinizione di un metodo : un fondamentale meccanismo fornito dalle sottoclassi è quello di modificare la definizione di un metodo presente nella superclasse .

Shadowing : Oltre a modificare l'implementazione di un metodo una sottoclasse può anche ridefinire una variabile d'istanza definita nella superclasse . Questo meccanismo è detto shadowing ed è molto diverso dalla ridefinizione (overriding)

classi astratte : molti linguaggi permettono la definizione di classi che non hanno istanze , perché la classe manca dell'implementazione di qualche metodo . In tali classi figura solo il nome e il tipo di uno o più metodi , queste classi vengono dette astratte .

- Scopo : servono a fornire interfacce di cui sarà data l'implementazione in qualche sottoclasse che ridefinirà il metodo di cui manca l'implementazione .

Relazione di sottotipo : i linguaggi vietano in genere che vi siano cicli nella relazione di sottotipo : non può accadere che due tipi A e B siano mutuamente sottotipi tra loro a meno che non coincidono La relazione di sottotipo è un ordine parziale . In molti linguaggi c'è un elemento massimo che indicheremo con object gli altri sono suoi sottotipi . La gerarchia di sottotipi non è pertanto un albero ma solo un grafo orientato aciclico .

Costruttori : la creazione di un oggetto ha una certo complessità che è composta di due azioni :

- allocazione delle memoria necessaria
- inizializzazione dei dati : compiuto dal costruttore della classe

il costruttore è il codice associato alla classe di cui il linguaggio garantisce l'esecuzione al momento della creazione dell'istanza .
Si pongono dunque una serie di questioni legate ai costruttori :

- **scelta del costruttore** : qualora il linguaggio permetta più di un costruttore associato alla classe dobbiamo specificare la nostra scelta . Essi possono essere distinti per il tipo o il numero degli argomenti
- **concatenamento dei costruttori** : l'inizializzazione di quella parte di un oggetto che proviene dalle superclassi avviene se il programmatore stesso intende chiamare i costruttori delle superclassi perché i linguaggi si limitano solo ad eseguire il costruttore della classe di cui si sta creando l'istanza . In altri linguaggi invece garantiscono che al momento dell'inizializzazione dell'oggetto viene invocato anche il costruttore della superclasse (constructor chaining)

ereditarietà : quando la sottoclasse non ridefinisce il metodo allora lo eredita nel senso che l'implementazione del metodo è resa disponibile alla sottoclasse . L'ereditarietà è un meccanismo che permette la definizione di nuovi oggetti a partire da altri oggetti esistenti .

- **Vantaggi** : si permette il riuso del codice in un contesto estendibile
- **differenza** : il sottotipo ha a che vedere con la possibilità di usare un oggetto in un altro contesto cioè è una relazione tra le interfacce di due classi , invece l'ereditarietà permette il

riuso del codice che manipola un oggetto è la relazione tra le implementazioni di due classi **visibilità ed ereditarietà :** una sottoclasse a volte può aver bisogno di accedere ad alcuni dati non pubblici , quindi molti linguaggi introduco una terza visita di una classe cioè quella che possono fare solo le sottoclassi (protected). Questo tipo di accesso dipende dall'implementazione della superclasse , ogni modifica di essa imporrà una modifica della sottoclasse .

Ereditarietà singola e multipla : in alcuni linguaggi una classe può ereditare solo una superclasse quindi la gerarchia è ad albero e viene chiamata ereditarietà singola . Altri linguaggi invece permetto che una classe erediti metodi da più superclassi e la gerarchia è rappresentata da un grafo aciclico e viene detta ereditarietà multipla

- problemi : conflitti di nomi , entrambe le superclassi forniscono l'implementazione di un metodo con la stessa signature .

Possiamo risolvere il problema in tre modi distinti :

- vietare sintatticamente che si possa avere produrre un conflitto di nomi
- imporre che ogni conflitto sia risolto esplicitamente dal programma, qualificando opportunamente ogni riferimento al nome in conflitto
- decidere convenzionalmente come risolvere il conflitto ad esempio a favore della prima classe

selezione dinamica dei metodi : essa permette la convivenza tra compatibilità dei sottotipi e l'astrazione . Quando un metodo m viene invocato su un oggetto o allora pertanto vi possono essere più versioni di m tutte possibili per o . La selezione di quale implementazione usare viene determinata a tempo di esecuzione in funzione del tipo dell'oggetto che riceve il messaggio .

Overloading vs selezione dinamica : in entrambi i casi il problema è lo stesso però nel primo caso l'ambiguità è risolta staticamente in base al tipo dei nomi coinvolti , nella selezione dei metodi invece la risoluzione dell'ambiguità avviene a tempo di esecuzione sfruttando il tipo dinamico dell'oggetto e non del suo nome .

ASPETTI IMPLEMENTATIVI

oggetti : un oggetto è rappresentato come se fosse un record con tanti campi quante sono le variabili della classe di cui è istanza più tutte le variabili che appaiono nella superclasse . Nel caso di shadowing ossia quando uno stesso nome per le variabili di istanza in una classe è usato con lo stesso tipo in una sottoclasse , l'oggetto ha più campi ciascuno corrispondente ad una dichiarazione (spesso non è possibile accedere al nome usato nella sopraclasse se non usando qualificatori particolari per esempio super) .
Nel caso di linguaggi con sistema di tipi statico , di ogni variabile di istanza è noto staticamente l'offset rispetto all'inizio del record , se ad un oggetto di istanza di una classe si accede mediante un riferimento statico di una superclasse,il controllo statico dei tipi assicura che si può accedere solo ad un campo della superclasse , che sarà allocato nella parte iniziale del record .

Classi ed ereditarietà : l'implementazione più intuitiva è quella che rappresenta la gerarchia delle classi come una lista concatenata . Ogni elemento rapprensenta una classe e contiene l'implementazione di tutti i metodi esplicitamente definiti o ridefiniti in una classe . Gli elementi sono collegati tra loro attraverso un puntatore che và dalla sottoclasse alla superclasse immediata .

- Svantaggi : è assai inefficiente perchè ogni invocazione richiede la scansione lineare della gerarchia delle classi .

Late binding di self : un metodo viene eseguito come una funzione , viene messo un rda sulla pila per le variabili locali del metodo , i parametri e tutte le altre informazioni . Però a differenza della funzione un metodo deve accedere anche alle variabili di istanza dell'oggetto sul quale è invocato che non è noto a tempo di compilazione . Però sappiamo la struttura di tale oggetto e dunque per ogni variabile di istanza è noto il suo offset all'interno della rappresentazione dell'oggetto . Quando viene invocato un metodo gli viene passato anche un puntatore all'oggetto che ha ricevuto il metodo , durante l'esecuzione del corpo del metodo tale puntatore è il this del metodo , se il metodo viene invocato sullo stesso oggetto che lo sta invocando viene passato this . Questo this serve per accedere alle variabili di istanza infatti l'accesso avviene tramite offset rispetto a questo puntatore .

Ereditarietà singola : al posto della lista concatenata la selezione dei metodi avviene in tempo costante . Se i tipi sono statici allora è noto a tempo di compilazione l'insieme di tutti i metodi che possono essere inviati all'oggetto . L'elenco di questi metodi è mantenuto nel descrittore della classe in esso vengono mantenuti non sono i metodi della classe o ridefiniti ma anche quelli dalle superclassi . Tale struttura dati è detta vtable .

- Vtable di una sottoclasse : si ottiene facendo una copia della vtable della sopraclasse sostituendo in questa copia tutti i metodi ridefiniti e aggiungendo in fondo i nuovi metodi definiti nella sottoclasse .
- Accesso : avviene al prezzo di due accessi indiretti visto che è staticamente noto l'offset di ogni metodo all'interno della vtable

Cast verso il basso : se la vtable di una classe contiene anche il nome della classe stessa è possibile gestire anche il cast verso il basso . Esso un meccanismo che permette di specializzare il tipo di un oggetto andando in senso contrario alla gerarchia di sottotipi . Un esempio si ha con l'uso di alcuni metodi di libreria definiti per essere più generali possibili (classe object) .

Problema della classe base fragile : tutte le implementazioni viste sono assai efficienti tranne il puntatore a this le altre informazioni sono determinate staticamente . Alcune superclassi si possono dimostrare fragili perchè modifiche innocue ad essa si possono ripercuotere sulla sottoclasse causando malfunzionamenti . In genere chi riscrive la superclasse non ha accesso alle sottoclassi . Il problema può insorgere per più motivi , ma i casi principali sono :

- il problema è architetturale : qualche sottoclasse sfrutta aspetti dell'implementazione della superclasse che sono stati modificati nella nuova versione .
  - Soluzione : si limita riducendo l'ereditarietà (di implementazione) a favore della relazione di sottotipo (cioè l'ereditarietà di interfaccia)
- il problema è implementativo : il malfunzionamento della sottoclasse dipende solo da come la macchina astratta (e il compilatore) ha rappresentato l'ereditarietà

selezione dinamica dei metodi nella JVM : l'architettura jvm è una macchina basata su pila (non ha registri accessibili agli utenti e tutti gli operandi delle operazioni sono passati su una pila che è contenuta nel rda della funzione correntemente in esecuzione) .

- compilazione : la compilazione delle classi avviene in modo separato, ogni classe da origine ad un file che la macchina astratta carica dinamicamente quando il programma è in esecuzione effettua un riferimento a quella classe
- file generato : contiene una tabella dei simboli usati dalla classe stessa , cioè variabili di istanza metodi pubblici e privati ecc..
- associazione : ad ogni nome di variabili di istanza e di metodo sono associate alcune informazioni tra le quali la classe dove i nomi sono definiti e il loro tipo
- riferimento : quando a run-time di fa riferimento ad un nome per la prima volta questo viene risolto usando le informazioni della tabella dei simboli

- rappresentazione: dei metodi in un descrittore di classe può essere pensata analogamente a quella di una vtable : la tabella relativa ad una sottoclasse inizia con una copia di quella della superclasse dove i metodi ridefiniti sono stati sostituiti con la nuova definizione . Gli offset tuttavia non sono calcolati staticamente .
- invocazione : si possono distinguere 4 casi principali :
  - il metodo è statico cioè un metodo associato ad una classe e non ad un'istanza come tale non può fare riferimento a this
  - il metodo deve essere selezionato dinamicamente
  - il metodo deve essere selezionato dinamicamente e invocato tramite this
  - il metodo proviene da un'interfaccia (cioè da una classe astratta che non fornisce implementazioni)
- differenze tra i casi : i primi tre casi si differenziano soltanto per i parametri passati al metodo , nel primo caso non viene passato alcun oggetto oltre ai parametri nominati nella chiamata , nel secondo viene passato un riferimento all'oggetto su cui è chiamato il metodo , il terzo viene passato un riferimento a this . Nell'ultimo caso l'offset potrebbe non essere lo stesso nel caso di due invocazioni dello stesso metodo su oggetti di classi diverse
- caso reale : o.m(parametri) , attraverso il riferimento ad o,la macchina astratta accede alla tabella dei simboli e d questa preleva il nome di m . A questo punto ricerca questo nome nella vtable della classe e determina il suo offset . Questo offset viene salvato per gli usi futuri

ereditarietà multipla : in questo caso si pongono problemi e richiede un overhead non trascurabile . I problemi sono di due ordini :

- chiarire come sia possibile adattare la tecnica delle vtable per gestire le chiamate di altri metodi
- occorre stabilire cosa fare dei dati presenti nelle superclassi e questo darà luogo a due interpretazioni diverse dall'eredità multipla che possiamo chiamare ereditarietà con replicazione ed ereditarietà con condivisione

struttura delle vtable :

```
class A {
        int x ;
        int f() {…}
}
class B {
        int y ;
        int g(){...}
        int h(){...}
}
class C extending A,B{
        int z ;
        int g(){...}
        int k(){...}
}
```

per rappresentare un istanza di C si può iniziare con i campi di A poi quelli di B e infine quelli di C . sappiamo che per effetto della relazione di sottotipo possiamo accedere ad un'istanza di C con riferimenti statici dei tre tipi A,B e C . vi sono due visite di un'istanza di C :

- qualora vi si acceda con un riferimento statico di tipo C o di tipo A,la tecnica descritta per l'ereditarietà semplice funziona correttamente (l'offset statico deve tener conto che nel mezzo ci sono anche le variabili statiche di B)
- quando l'istanza C viene vista come un oggetto di B occorre tener conto che le variabili di B

non sono all'inizio del record ma ad una distanza d determinata staticamente all'inizio di esso . Quando dunque un riferimento di C è convertito ad un riferimento alla stessa istanza ma con tipo statico B occorre sommare al riferimento questa costante d .

una vtable di C è divisa in due parti distinte :

- la prima comprende i metodi di A (eventualmente con la loro ridefinizione) e i metodi di C
- comprende i metodi di B eventualmente con le loro definizioni

nella rappresentazione dell'istanza di c vi sono due puntatori alle vtable :

- in corrispondenza della vista A e C avremo il puntatore ai metodi di A e C
- in corrispondenza della vista B avremo il puntatore alla vtable con i metodi di B (si osservi che questa è una vtable della classe C

invocazione :

- per i metodi propri di C o che viene ridefinito o ereditato da A segue le stesse regole dell'eredità singola
- per i metodi di B il compilatore deve tener conto che il puntatore alla vtable di questi metodi non è all'inizio dell'oggetto ma di d posizioni in avanti .

Binding di this : quale riferimento passare ai metodi di C?

- Per i metodi di A e C basta passare this
- per i metodi di B bisogna distinguere due casi :
  - il metodo è ereditato da B : in tal caso basta passare al metodo la vista dell'oggetto attraverso la quale abbiamo trovato la vtable
  - il metodo è ridefinito in C : in tal caso il metodo potrebbe far riferimento alle variabili d'istanza di A , e dunque occorre passargli la vista della superclasse

ereditarietà multipla con replicazione : la struttura della vtable viene allargata con i metodi della superclasse nella vista iniziale .

Ereditarietà multipla con condivisione : C++ per default segue ogni percorso di ereditarietà separatamente, quindi l'oggetto D contiene due separati oggetti di A . Se l'ereditarietà da A a B e l'ereditarietà da A a C sono entrambe segnate come "virtual" ("class B : virtual A"), C++ si prende particolare cura di creare solo un oggetto A, l'utilizzo dei membri di A funziona correttamente. Se l'ereditarietà virtuale e l'ereditarietà non virtuale sono mischiate, esiste solo un A virtuale e un A non virtuale per ogni percorso di ereditarietà non virtuale a A . ad ogni classe del diamante corrisponde una specifica vista sulla rappresentazione dell'oggetto . In corrispondenza di ogni vista c'è un puntatore alla porzione condivisa dell'oggetto .

POLIMORFISMO E GENERICI

quando un linguaggio una nozione d compatibilità strutturale tra tipi come appunto la relazione di sottotipo tra classi è chiaro che si ha una qualche forma di polimorfismo pur non completamente generale come il polimorfismo universale si sottotipo . Per la relazione di sottotipo ogni istanza di una classe A ha per tipo anche tutte le superclassi di A . questa proprietà è particolarmente interessante nel caso dei metodi .

Consideriamo un metodo :

B foo (A x) {… }

in virtù della compatibilità di sottotipo , foo può riceve come argomento un valore di una qualsiasi sottoclasse di A . il codice di foo non ha bisogno di essere adattato alle specifiche sottoclassi : la

struttura stessa delle classi assicura che le operazioni possibili per i valori di tipo A sono possibili anche per i valori delle sue sottoclassi . Si tratta di una forma di polimorfismo implicito, perchè l'istanziazione avviene automaticamente al momento dell'applicazione del metodo ad un valore della sottoclasse .

Consideriamo un metodo :

A ide (A x) { return x; }

tuttavia supposto che C sia sottotipo di A e c sia un istanza di C il seguente assegnamento sarebbe rifiutato dal controllore statico dei tipi :

C cc = ide (c) ;

nonostante sia perfettamente sensato da un punto di vista semantico . Il punto è che i linguaggi comuni hanno un sistema di tipi poco astuto che non è in grado di riconoscere ,per ogni T sottotipo di A avremo come output sempre T ,è corretto per ide né hanno un meccanismo linguistico con il quale il programmatore possa esprimere che il tipo del risultato di un metodo dipende dal tipo dell'argomento . Tuttavia questo ragionamento garantisce la correzione dinamica con un cast all'ingiù

C cc = (C) ide (c)

non causerà problemi a tempo di esecuzione .

Generici in java : il template è un frammento di programma dove alcuni tipi sono indicati come parametri , che possono essere poi opportunamente istanziati su tipi concreti . La versione di java 5 introduce un concetto analogo i cui costrutti vengono chiamati generici .
In java 5 possono essere generici sia le definizioni di tipo che quelle di metodo . I tipi che costituiscono i parametri attuali devono essere tipi di classe o array . Il cast dinamico che doveva essere aggiunto nella versione non generica non è ora più necessario .

Implementazione dei generici in java : a differenza dei template di c++ che vengono risolti a tempo di linking mediante duplicazione e specializzazione del codice . Dei generici di java esiste un'unica copia . I generici sono implementati a livello di compilazione mediante un meccanismo di cancellazione : un programma che contiene generici viene sottoposto al controllore di tipi . Quando è finito il controllo allora il programma viene trasformato in un programma analogo dove tutti i generici sono stati cancellati , infine se dopo tutte le manipolazioni il programma fosse scorretto rispetto ai tipi , sono inseriti alcuni cast dinamici .
Due importanti conseguenze di questa implementazione sono :

- la macchina astratta sottostante non ha bisogno di essere modificata . L'aggiunta di generici non comporta modifiche da replicare per architetture diverse ma è localizzata nel compilatore
- è possibile mescolare codice generico e codice non generico in modo semplice e sicuro .