Linguaggi di Programmazione

Livio Zoccarato Simone Tomada

Anno Accademico 2020–2021

Questo documento è stato scritto da due studenti del corso durante le lezioni dell'anno accademico 2020/2021 (prof. Di Gianantonio), dunque viene fornito as is e si declina ogni responsabilità in caso di errori, imprecisioni e cambiamenti di contenuti negli anni successivi. Inoltre consigliamo di accompagnarlo con le lezioni e con il libro per approfondirne i contenuti.

Indice

Capitolo 1

Introduzione

1.1 Aspetti di un linguaggio di programmazione

Sintassi quali sono le sequenze di caratteri che costituiscono un programma, qual è la struttura del programma

Es.: il programma è costituito da una dichiarazione e un corpo e cosa contengono. É facile da formalizzare

Semantica il significato di un programma, l'effetto della sua computazione. Difficile da formalizzare, viene data in linguaggio naturale.

Pragmatica uso tipico del linguaggio e le convenzioni che si usano per scrivere del codice, quando è un buon codice.

Implementazione come il codice scritto venga convertito in codice macchina.

Librerie codice fornito per implementare funzionalità.

Tools editing, debugging, gestione del codice

1.2 Tipologie di linguaggi

I linguaggi si dividono in due grandi famiglie: **imperativi** e **dichiarativi**.

1.2.1 Imperativi

Von Neumann si tratta di quella classe di linguaggi di alto livello derivanti da una copia isomorfica dello stile di programmazione dell'architettura di Von Neumann, ne sono esempio linguaggi come Fortran, Pascal e C.

Orientati agli oggetti si tratta di quell'insieme di linguaggi sviluppati per programmare secondo il paradigma di programmazione ad oggetti, un esempio è Java.

Linguaggi di scripting sono linguaggi sviluppati per alcuni ambienti real-time col fine di automatizzare alcuni task, ne fanno parte python, javascript, php, etc...

1.2.2 Dichiarativi

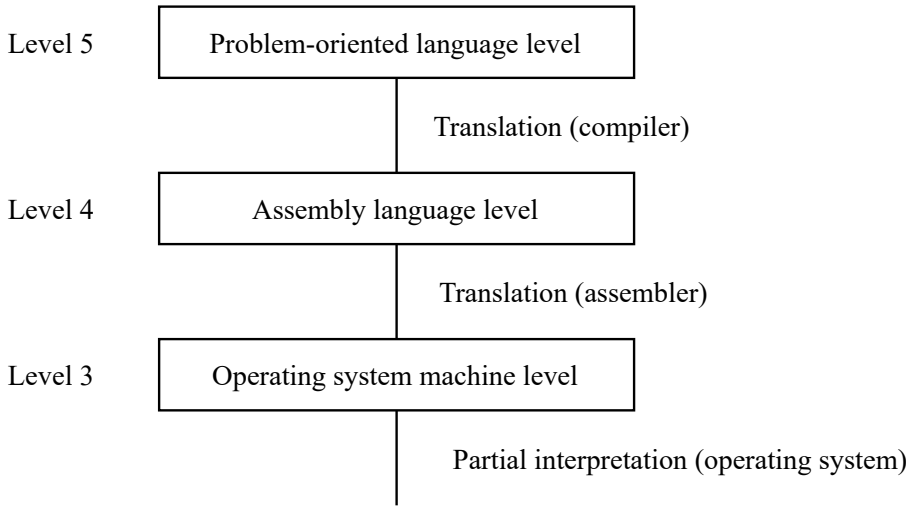
Funzionale un insieme di linguaggi che si basano sulle funzioni, per esempio haskell, scheme, pure lisp, etc...

Logico basato su vincoli, alcuni esempi sono: prolog, visiCalc e RPG

1.3 Macchina astratta

È un meccanismo per gestire la complessità del software. L'idea è di risolvere i problemi per gradi: si parte dalla macchina fisica che esegue istruzioni in linguaggio macchina, ma fare tutto con quelle sarebbe complicato. Si scrive quindi il kernel del sistema operativo, un insieme di funzioni base che permettono di gestire l'hardware e che possono essere utilizzate da altri linguaggi. Si ottiene così il primo livello della macchina astratta e man mano si aggiungono livelli.

Alla fine abbiamo una gerarchia di macchine astratte $\mathcal{M}_i$, ciascuna costruita sulla precedente, a partire dal livello hardware e ciascuna caratterizzata da un linguaggio $\mathcal{L}_i$, con cui scrivere codice.



Gran parte del lavoro di programmazione che viene svolto può essere visto come implementare una macchina astratta più complicata a partire da una più semplice, usando quindi del software già esistente a cui aggiungere funzionalità.

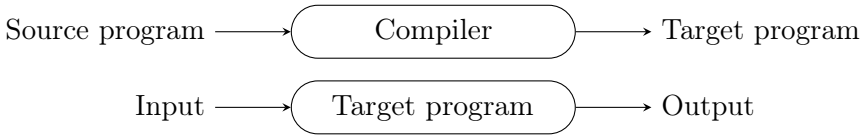
Quando si scrivono dei programmi per esempio in $\mathcal{L}_{Java}$ si sta scrivendo la macchina astratta $\mathcal{M}_{Java}$ utilizzando uno dei livelli macchina sottostanti, in questo caso la $\mathcal{M}_{JVM}$ (Java Virtual Machine) che qualcuno ha già implementato. Viene dunque eseguita una traduzione nel linguaggio relativo $\mathcal{L}_{JVM}$ (Java Byte Code, il vantaggio di questo codice è che non è specifico per una particolare macchina).

1.4 Compilazione vs. Interpretazione

Esistono due approcci per rendere eseguibile il codice:

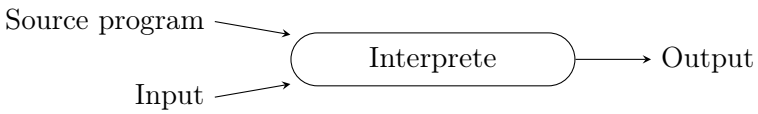
1.4.1 Compilazione pura

Il compilatore traduce il programma in un altro linguaggio equivalente ma di più basso livello. Il programma sorgente e il compilatore non sono necessari durante l'esecuzione del codice.



1.4.2 Interpretazione pura

L'interprete riceve programma sorgente e dati e traduce passo passo le singole istruzioni che vengono eseguite immediatamente. Interprete e programma sorgente sono dunque presenti durante l'esecuzione del programma.



1.4.3 Vantaggi e svantaggi

Compilazione

- Migliori prestazioni perché la traduzione si fa una volta sola
- Si possono trovare gli errori fin da subito

Interpretazione

- Vantaggi
  - Maggiore flessibilità nell'implementazione
  - Esecuzione immediata

  - Più semplice costruire il debugger e fare debugging del codice, perché è più facile mantenere dei collegamenti tra il linguaggio tradotto e il linguaggio del codice e individuare la fonte degli errori

- Svantaggi
  - Scarsa efficienza, dato che l'interprete deve effettuare la decodifica del linguaggio $\mathcal{L}$ durante l'esecuzione

1.5 Traduzione in codice macchina

Nei casi reali, la traduzione in codice macchina avviene tramite molteplici passi di traduzione. Nella maggior parte delle implementazioni si ha una combinazione tra compilazione e interpretazione.

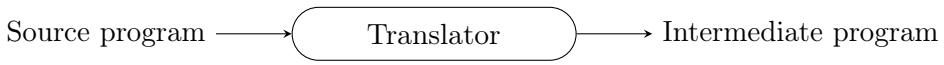
- Linguaggi interpretati

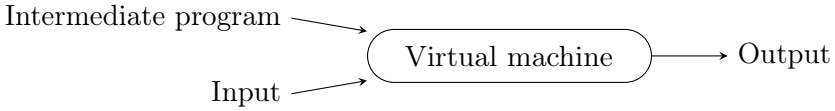
  pre-processing fase in cui il codice viene trasformato in modo che l'esecuzione da parte dell'interprete diventi più veloce. È seguita dall'interpretazione.

- Linguaggi compilati

  generazione del codice intermedio ovvero un codice non specifico per un particolare calcolatore, dunque con il vantaggio della portabilità. Esempi: Pascal P-Code, Java bytecode, Microsoft COM+.

Il caso tipico prevede una prima fase in cui il programma viene tradotto in un linguaggio non macchina, intermedio, e poi viene eseguito mediante un interprete, come viene descritto nella figura seguente:





La differenza tra i linguaggi interpretati e compilati sta nella distinzione tra compilazione e pre-processing.

Compilazione è la traduzione da un linguaggio ad un altro

- prevede un'analisi complessiva dell'input

- viene riconosciuta la struttura sintattica del programma, dunque permette un controllo degli errori preliminare

Pre-processing gli aspetti sopracitati non sono presenti, si esegue una trasformazione sintattica e locale del programma, più leggera e semplice. Es.: sostituzione dei vari simboli con una rappresentazione più sintetica così l'interprete è più veloce.

1.5.1 Supporto a run-time

Raramente un compilatore produce solo codice macchina, bensì produce anche *istruzioni virtuali* che sono più sofisticate rispetto a quelle del linguaggio macchina. Sono:

- chiamate al sistema operativo
- chiamate a funzioni di libreria es.: funzioni matematiche (sin, cos, log, ecc.), input-output

La traduzione non avviene a livello di codice macchina, ma a livello di macchina virtuale intermedia, costituita dalla macchina fisica e da un insieme di librerie, dette *supporto a real-time*. Un programma chiamato *linker* collega poi il codice generato alle librerie e si produce dunque il codice macchina.

In figura 1.1 una rappresentazione dei precedenti passaggi.

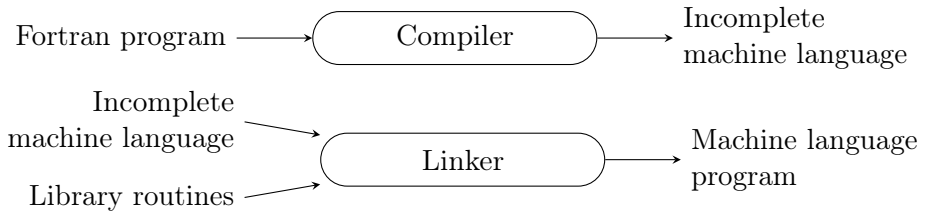


Figura 1.1: Supporto a run-time

1.5.2 Assemblaggio post-compilazione

Il compilatore, invece di produrre codice macchina, produce assembly, ovvero una rappresentazione simbolica del codice macchina.

- facilita il debugging
- isola il compilatore da modifiche nel formato delle istruzioni, in quanto il codice macchina potrebbe usare un formato delle istruzioni diverso (come un'istruzione macchina viene rappresentata come sequenza di byte). Es.: ARM.

Con questa tecnica si usa la macchina virtuale assembly.

1.5.3 Traduzioni da linguaggio a linguaggio (C++)

Esempio delle tecniche presentate precedentemente. Le prime implementazioni di C++ generavano un programma intermedio in C per poter essere compilato con lo stesso compilatore.

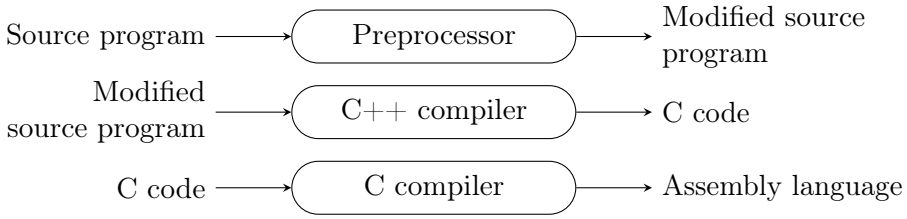


Figura 1.2: Fasi della traduzione da C++ a codice macchina

Con questa tecnica si usa la macchina virtuale C.

1.5.4 Compilazione dinamica just-in-time

La compilazione viene eseguita durante l'esecuzione del programma e solo per alcuni blocchi di codice, per esempio quelli che vengono usati più volte, in modo che vengano tradotti una sola volta al fine di migliorare le prestazioni dei programmi. Questa tecnica è spesso usata nei programmi Java.

1.6 Bootstrapping

Il termine deriva da *«sollevarsi dal suolo tirando i lacci dei propri stivali»* (Barone Munchausen) e si tratta di una tecnica che permette di scrivere un compilatore che si auto-compila, ovvero scrivere il compilatore con lo stesso linguaggio che intende compilare.

Il caso più semplice consiste nell'utilizzare un compilatore di un linguaggio per compilare un nuovo compilatore, che magari traduce il codice iniziale in codice più efficiente. Normalmente questa operazione viene eseguita due volte, per capirne il motivo supponiamo di aver scritto in C un nuovo compilatore che permette di ottenere codice compilato più efficiente, dopo la prima compilazione otterremo il nostro compilatore ma, sebbene questo compilerà programmi più efficienti, lo stesso non godrà di questi benefici. Compilando una seconda volta utilizzando il nuovo compilatore si otterrà lo stesso compilatore ma più efficiente.

1.6.1 Il caso del pascal

Pascal ebbe successo per la semplicità nella creazione di un compilatore in un momento in cui la varietà di processori, ciascuna con un proprio set

di istruzioni (codice macchina), portava alla necessità di scrivere un numero abbastanza elevato di compilatori.

Punto di partenza

Il meccanismo utilizzato dal pascal per semplificare la creazione di compilatori si basa sulla tecnica del bootstrap. I tre elementi successivi sono il punto di partenza per costruire un compilatore per un particolare linguaggio macchina:

- un compilatore Pascal, scritto in Pascal:[1]

$$\mathcal{C}_{Pascal}^{Pascal \to PCode}$$

- riscrittura del compilatore precedente in PCode:

$$\mathcal{C}_{PCode}^{Pascal \to PCode}$$

- un interprete PCode scritto in Pascal:[2]

$$\mathcal{I}_{Pascal}^{PCode} \tag{1.1}$$

Il PCode è un linguaggio intermedio che astrae dal processore specifico.

Ottenere l'interprete

Per ottenere l'interprete per il linguaggio macchina richiesto costruisco:

- Un interprete PCode scritto nel linguaggio macchina richiesto, facile da ottenere partendo dall'interprete (1.1):

$$\mathcal{I}_{LM}^{PCode}$$

Preso un programma Pascal PrPa posso:

- ottenere la sua traduzione in PCode:

$$PrPC = \mathcal{I}_{LM}^{PCode}(\mathcal{C}_{PCode}^{Pascal \to PCode}, PrPa)$$

  L'interprete essendo scritto in linguaggio macchina può essere eseguito nel calcolatore, permette quindi di eseguire (interpretare) un programma scritto PCode. Il programma che decidiamo di interpretare è proprio il compilatore da Pascal a PCode e come dati il programma PrPA ottenendo come output l'output del compilatore avente input PrPa e dunque il programma PrPA tradotto in PCode.

[1] L'apice indica il linguaggio accettato dal compilatore, la freccia indica che l'output del compilatore è del linguaggio descritto subito a destra ed infine il pedice indica il linguaggio con cui è scritto il compilatore, denotato dalla c maiuscola.

[2] L'interprete prende in input il PCode e i dati per produrre l'output, la notazione è la stessa di quella descritta precedentemente, in questo caso con la i maiuscola indicante interprete. Notare l'assenza della freccia indicante la traduzione.

- eseguire la traduzione:

$$\mathcal{I}_{LM}^{PCode}(PrPC, Dati)$$

Ottenere il compilatore

Per ottenere un compilatore Pascal scritto nel linguaggio macchina richiesto:

- manualmente, si trasforma il compilatore $\mathcal{C}_{Pascal}^{Pascal \to PCode}$ in $\mathcal{C}_{Pascal}^{Pascal \to LM}$. Il lavoro del compilatore è indipendente dal linguaggio macchina che si vuole produrre, dunque solo una piccola parte del codice va modificata.

- Ottenere il compilatore da Pascal a Linguaggio Macchina scritto in PCode:

$$\mathcal{C}_{PCode}^{Pascal \to LM} = \mathcal{I}_{LM}^{PCode}(\mathcal{C}_{PCode}^{Pascal \to PCode}, \mathcal{C}_{Pascal}^{Pascal \to LM}) \qquad (1.2)$$

  Utilizzando l'interprete (1.1) siamo in grado di interpretare il compilatore da Pascal a PCode scritto in PCode con input il compilatore da Pascal a Linguaggio macchina scritto in Pascal ottenendo la traduzione di quest'ultimo in PCode.

- Ottenere il compilatore da Pascal a Linguaggio Macchina scritto in Linguaggio Macchina:

$$\mathcal{C}_{LM}^{Pascal \to LM} = \mathcal{I}_{LM}^{PCode}(\mathcal{C}_{PCode}^{Pascal \to LM}, \mathcal{C}_{Pascal}^{Pascal \to LM})$$

  Rieseguendo la procedura precedente ma usando il compilatore ottenuto al punto precedente come programma da interpretare si ottiene il compilatore scritto in Linguaggio Macchina che traduce da pascal a linguaggio macchina.

  Abbiamo quindi ottenuto il compilatore desiderato, siamo ora in grado di tradurre un qualsiasi programma da Pascal in Linguaggio Macchina.

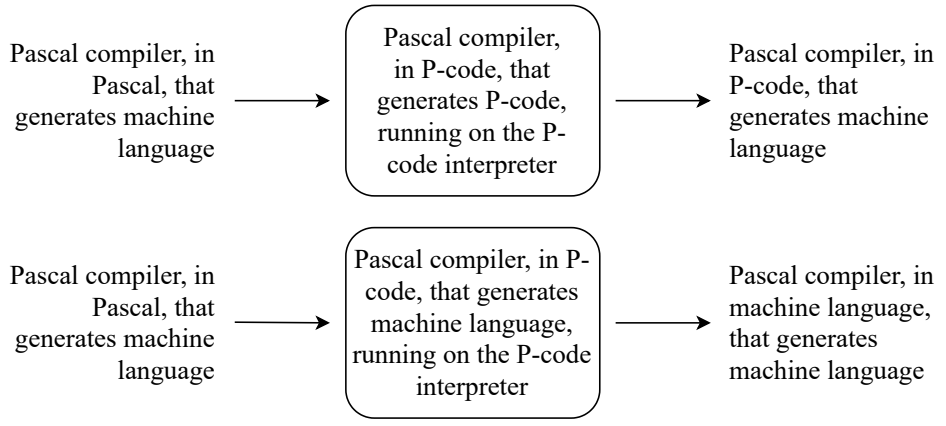


Figura 1.3: Schematizzazione dei passaggi eseguiti

1.7 Panoramica sulla compilazione

La compilazione è divisa in più fasi, rappresentate nella figura seguente:

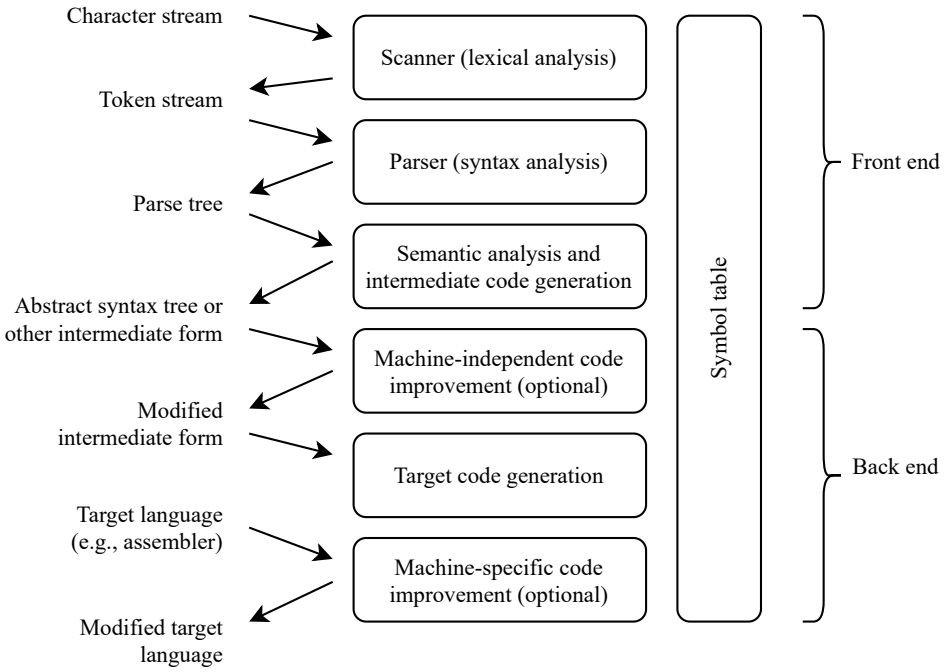


Figura 1.4: Fasi della compilazione

Front end trasformazione del codice sorgente in una struttura ad albero

Back end produce il codice intermedio, lo ottimizza e lo trasforma in codice macchina

1.7.1 Analisi lessicale - scansione

L'elemento principale è lo *scanner*, ciò che fa è dividere il programma in lessemi, le unità più piccole e significative, per esempio gli operatori o gli identificatori di funzione. Ogni classe di lessema viene espressa tramite linguaggio regolare.

Lo scanner inoltre esegue un controllo di alcuni errori, per esempio stringhe che non appartengono a nessun linguaggio regolare.

Per ogni lessema produce un token, che è una rappresentazione che si passa alla fase successiva, per esempio il tipo di dato.

Questo argomento verrà approfondito nel sottoparagrafo 2.2.1 a pagina 33.

1.7.2 Analisi sintattica - Parsing

Durante questa fase lo stream di token viene analizzato e si costruisce l'albero della sintassi, ovvero il programma viene visto come un albero di derivazione di una grammatica libera dal contesto. Durante questa fase è anche possibile individuare dei possibili errori sintattici, ovvero se il programma scritto non può essere ottenuto da nessuna derivazione della nostra grammatica.

1.7.3 Analisi semantica

L'analisi semantica si occupa di eseguire due tipi di controlli sul codice per trovare errori che non sono rilevabili nelle fasi precedenti:

- controlli statici, non implementabili dal parser (es.: type checking)
- controlli dinamici, eseguibili solo a tempo di esecuzione (es.: indice di matrice fuori limite)

Questa fase produce l'albero della sintassi astratta a partire dall'albero della sintassi creato nella fase precedente, ove gli elementi non necessari per le fasi successive sono stati rimossi (es.: parentesi).

1.7.4 Modulo intermedio

Viene prodotto il codice intermedio, indipendente dal processore, facile da ottimizzare o compatto (richieste contrastanti). Somiglia spesso al codice macchina per qualche macchina astratta (es.: una macchina stack o una macchina con molti registri).

1.7.5 Ottimizzazione

Le operazioni di ottimizzazione possono essere eseguite sia sul codice intermedio, sia sul codice macchina/assembly.

- trasforma il codice intermedio in uno equivalente ma più efficiente, che esegue più velocemente e con meno memoria.
- esegue alcune ottimizzazioni specifiche della macchina come l'uso di istruzioni speciali, modalità di indirizzamento, ecc.

Questa fase non è obbligatoria, è molto complessa e si basa su delle euristiche, non la vedremo nel dettaglio.

1.7.6 Tabella dei simboli

Si tratta di una struttura dati utilizzata durante tutte le fasi, molto semplicemente associa ad ogni identificatore tutte le informazioni necessarie, può essere mantenuta anche dopo il completamento dell'operazione di compilazione per fini di debugging.

Capitolo 2

Sintassi

2.1 Grammatiche

Una grammatica libera (dal contesto) è caratterizzata da:

- T: un insieme di simboli terminali
- NT: un insieme di simboli *non* terminali (variabili)
- R: un inseme di regole di produzione
- S: il simbolo iniziale $\in NT$

Le regole R (libere dal contesto) sono nella forma:

$$V \to w$$

dove $V \in NT$ e $w \in (T \cup NT)^*$ ovvero w è una stringa che può essere composta da simboli sia terminali che non terminali.

Una grammatica libera dal contesto può essere rappresentata come una quadrupla $\langle T, NT, R, S \rangle$.

2.1.1 Derivazione

Per derivazione si intende come arrivare da un simbolo iniziale ad una parola (stringa di terminali) seguendo le regole di produzione, la quale può essere rappresentata univocamente da un albero di derivazione che ne mette in luce la struttura.

2.1.2 Albero di derivazione

Gli alberi di derivazione sono alberi ordinati, ovvero:

- grafo orientato

- aciclico, connesso
- ogni nodo ha al più un arco entrante
- gli archi uscenti sono ordinati

Definizione formale

L'albero di derivazione (non parziale) su una grammatica $\langle T, NT, R, S\rangle$ è così formato:[1]

- la radice è etichettata con il simbolo iniziale S
- le foglie sono etichettate da dei simboli terminali in T
- ogni nodo interno n:
  - etichettato con un simbolo E non terminale
  - la sequenza w delle etichette dei suoi figli, deve apparire in una regola $E \to w$, in R

Gli alberi di derivazione sono fondamentali perché descrivono la struttura logica della stringa analizzata, ovvero permettono di visualizzare graficamente come una stringa si ottiene attraverso il processo di derivazione a partire dal simbolo iniziale. Vengono costruiti dai compilatori.

Diremo che un albero descrive una stringa $\alpha \in (V \cup T)^*$ se α è proprio la stringa che possiamo leggere dalle etichette delle foglie da sinistra a destra. È bene notare come se non si riesce ad ottenere un albero che descrive una particolare stringa allora quest'ultima non è ottenibile con la data grammatica.

Bisogna fare attenzione al fatto che sebbene un albero di derivazione rappresenti univocamente una derivazione non è detto che data una stringa esista un solo albero di derivazione che la descriva, infatti può capitare che per una stringa possano esserci diversi alberi, se una grammatica permette questa eventualità si dice che è una *grammatica ambigua*, concetto approfondito nel paragrafo 2.1.3 a pagina 26.

Un esempio di derivazione

Sia G la grammatica seguente:

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

[1] La definizione non è perfetta, ma sufficiente per comprendere i concetti descritti, per una definizione più generale è meglio consultare il materiale di *"Fondamenti di Informatica"*

ove P è costituito da:

$$E \to 0$$
$$E \to 1$$
$$E \to (E\ and\ E)$$
$$E \to (E\ and\ E)$$
$$E \to (not\ E)$$

Sia $w = ((0\ or\ 1)\ and\ (not\ 0))$ una stringa derivabile con la grammatica data.

Prendiamo la derivazione per produrre la stringa data:
$E \to (E\ and\ E) \to ((E\ or\ E)\ and\ E) \to ((E\ or\ E)\ and\ (not\ E)) \to$
$\to ((E\ or\ E)\ and\ (not\ E)) \to^2 ((0\ or\ 1)\ and\ (not\ 0))$

Costruiamone ora l'albero di derivazione, seguendo l'ordine descritto dalla derivazione precedente.

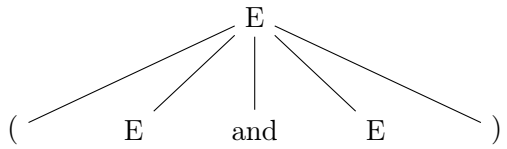


Figura 2.1: Passo 1

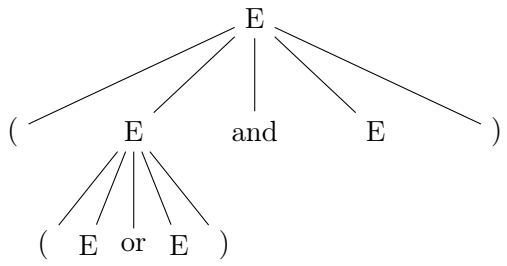


Figura 2.2: Passo 2

[2]Per semplicità sono state omesse le derivazioni intermedie che si occupano di trasformare le variabili in terminali 0 e 1, questa derivazione comprende 3 produzioni

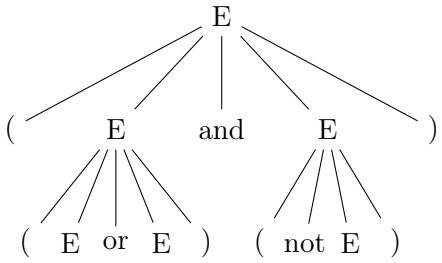


Figura 2.3: Passo 3

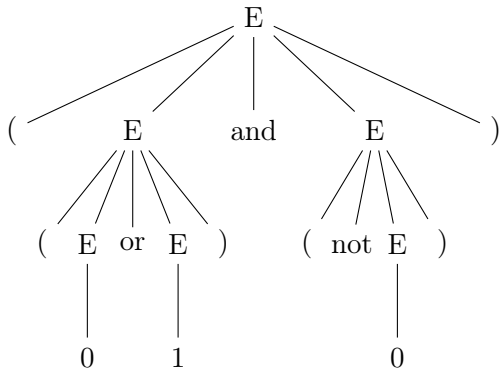


Figura 2.4: Passo 4

Se leggiamo da sinistra verso destra le foglie notare come si ottiene la stringa:

$$w = ((0\ or\ 1)\ and\ (not\ 0))$$

2.1.3 Grammatiche ambigue

Precedentemente è stato definito il concetto di grammatica ambigua, ovvero quelle grammatiche per le quali esiste almeno una parola con più di un albero di derivazione, andiamo ora ad analizzare le problematiche dell'ambiguità e come gestirla.

2.1.4 Problematiche dell'ambiguità

L'ambiguità porta con sé l'incertezza durante la valutazione della stringa, per capirne il motivo prendiamo come esempio il caso seguente.

Sia G un grammatica definita come:

$$G = \langle \{E\}, \{+, -, \times, (,), a, b, c\}, P, E \rangle \tag{2.1}$$

ove P è costituito da:

$$E \to E + E$$
$$E \to E - E$$
$$E \to E \times E$$
$$E \to (E)$$
$$E \to a$$
$$E \to b$$
$$E \to c$$

Sia $w = a + b \times c$
Per la stringa w sono presenti due alberi di derivazione.

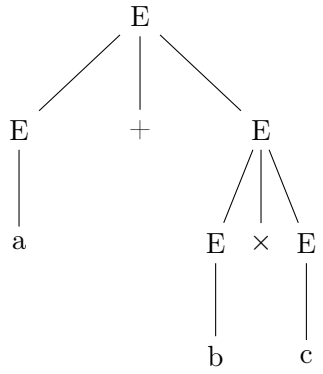


Figura 2.5: Derivazione con priorità sulla moltiplicazione

In questo caso la valutazione della stringa viene eseguita seguendo le convenzioni dell'aritmetica, ovvero la priorità viene data alla moltiplicazione.

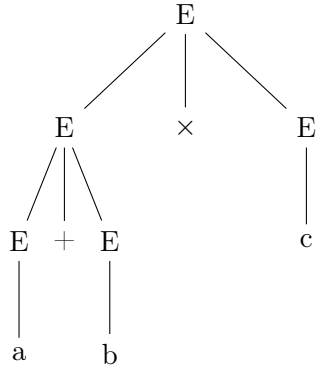


Figura 2.6: Derivazione con priorità sulla addizione

In questo caso la valutazione della stringa viene eseguita dando la priorità all'addizione.

Supponiamo ora che i simboli terminali rappresentino delle variabili in un programma, per esempio prendiamo $a = 2,\ b = 3,\ c = 4$, nel caso della valutazione che verrebbe eseguita seguendo il primo albero di derivazione si ottiene che:

$$\text{risultato} = 2 + 3 \times 4 = 2 + 12 = 14$$

Se invece seguiamo l'ordine del secondo caso si ottiene:

$$\text{risultato} = 2 + 3 \times 4 = 5 \times 4 = 20$$

Ovviamente una situazione del genere non è ottimale, basti pensare ad un linguaggio di programmazione che per la stessa espressione dia risultati profondamente diversi. Dobbiamo quindi trovare un modo per gestire l'ambiguità delle grammatiche al fine di ottenere sempre un unico albero di derivazione.

2.1.5 Gestione dell'ambiguità

Esistono due possibili approcci per gestire l'ambiguità:

- rendere la grammatica non ambigua
- convivere con la grammatica ambigua

Rendere la grammatica non ambigua

Consiste nel trasformare una grammatica ambigua in una non ambigua, in genere aggiungendo nuovi terminali e nuovi non terminali, spesso complicandola. Esistono delle tecniche standard che non vedremo nel dettaglio.

Mostriamo ora il risultato della trasformazione della grammatica ambigua 2.1 a pagina 26 in grammatica non ambigua:

$$G = \langle \{E, T, F, \mathit{Var}\}, \{+, -, \times, (,), a, b, c\}, P, E \rangle \tag{2.2}$$

ove P è costituito da:

$$
\begin{aligned}
&E \to T \\
&E \to E + T \\
&E \to E - T \\
&T \to F \\
&T \to T \times F \\
&F \to (E) \\
&F \to Var \\
&Var \to a \\
&Var \to b \\
&Var \to c
\end{aligned}
$$

Convivere con la grammatica ambigua

Con questo approccio si preferisce lasciare la grammatica così come sta e fornire delle regole per la corretta interpretazione della stringa, è l'approccio anche usato in matematica nel caso dell'aritmetica.

Supponiamo di voler valutare l'espressione $w = a + b \times c$ il cui risultato, però, come abbiamo visto varia a seconda di quale operatore riceve la priorità. Si è deciso perciò per *convenzione* che la priorità la riceve l'operatore moltiplicazione, il che permette di ottenere un risultato univoco in casi ambigui. La convenzione appena descritta è banalmente una regola imposta per convivere con l'ambiguità della grammatica.
In genere vengono specificati:

- l'ordine di precedenza degli operatori
- se un operatore (o classe di operatori), viene associato a destra o a sinistra

2.1.6 Alberi di derivazione astratti

Per alberi di derivazione astratti si intendono degli alberi di derivazione privi di informazioni ridondanti ed inutili. Una volta che l'albero di derivazione è stato generato i simboli usati per disambiguare diventano inutili siccome le informazioni portate da questi vengono già codificate dalla struttura dell'albero. Per capire meglio questo concetto guardiamo ancora l'esempio della grammatica 2.2 a pagina 28.

Sia w la stringa da derivare che definiamo come $w = (a + b) \times c$, l'albero di derivazione sarà:

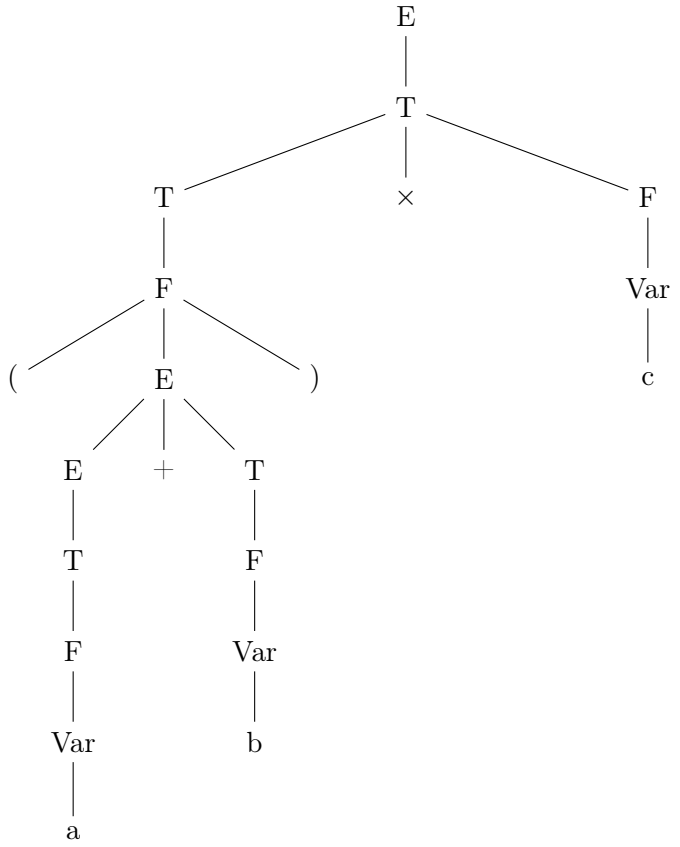


Figura 2.7: Derivazione di w

L'albero precedente descrive esattamente w, le informazioni riguardo l'ordine degli operatori sono già codificate nella struttura dell'albero, possiamo quindi costruire un albero di derivazione per la stessa stringa ma utilizzando una grammatica ambigua.

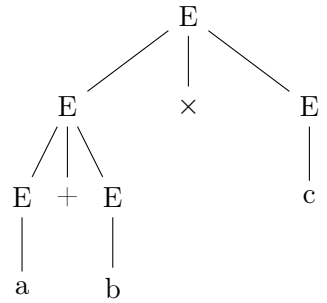


Figura 2.8: Albero astratto equivalente

Da notare la notevole differenza di dimensione tra il primo, l'albero

derivante dall'interpretazione con la grammatica non ambigua, ed il secondo ovvero quello astratto.

Sintassi astratta

Si tratta di una sintassi minimale che permette di generare gli alberi astratti, mette in luce la parte importante della sintassi concreta.

Linguaggio intrinsecamente ambiguo

Si dice che un linguaggio è intrinsecamente ambiguo se ogni grammatica che lo genera è ambigua, è necessario ricorrere a meccanismi esterni alla grammatica per risolvere le ambiguità.

2.1.7 Classi di grammatiche

Le grammatiche possono essere classificate in diverse classi:

- a struttura di frase
- dipendenti dal contesto
- libere dal contesto
- lineari destre o lineari sinistre
- regolari

Le differenze consistono in:

- diverso grado di libertà nel definire le regole, si va da regole strette nelle regolari a regole più libere delle grammatiche a struttura di frase
- ampiezza della classe dei linguaggi definibili
- complessità nel decidere se una parola appartiene al linguaggio

Di nostro interesse sono le regolari, usate dallo scanner nell'analisi lessicale, e le context free, usate dal parser nell'analisi sintattica.

2.1.8 Uso delle grammatiche libere dal contesto nei parser

Nei parser le grammatiche libere dal contesto sono il compromesso ottimale tra espressività e complessità:

- ampio insieme di linguaggi definibili
- nei casi pratici, riconoscimento in tempo lineare sulla lunghezza della stringa

Tuttavia, con le grammatiche libere dal contesto non si possono eliminare dall'insieme di programmi riconosciuti (accettati) programmi che non rispettano alcuni vincoli sintattici (contestuali) come:

- identificatori dichiarati prima dell'uso
- diverso numero di parametri nella definizione e nelle chiamate delle funzioni
- modifiche alla variabile di controllo di un ciclo for
- tipi nelle assegnazioni non rispettati

Soluzione dei compilatori

- usare grammatiche libere (efficienti)
- una volta costruito l'albero sintattico, effettuare una fase di *analisi semantica*

La fase di analisi semantica è chiamata anche semantica statica: i controlli sul codice sono eseguibili a tempo di compilazione invece che durante l'esecuzione (semantica dinamica).

Semantica

La semantica di un programma definisce il suo significato, il suo comportamento a run-time, più che un semplice controllo degli errori.

La semantica viene quasi sempre definita informalmente usando il linguaggio naturale, ma un approccio formale è possibile e preferibile:

Semantica operazionale strutturata un sistema di regole (di riscrittura) che descrivono il risultato della valutazione di un qualsiasi programma

Semantica denotazionale il significato di un programma è dato da una funzione, che esprime il comportamento input/output del programma stesso. Dominio e codominio di questa funzione sono opportune strutture matematiche che corrispondono non solo ai dati di ingresso/uscita, ma anche ad alcune strutture interne del linguaggio, quali l'ambiente e la memoria.

2.2 Front end del compilatore

Similmente ai linguaggi naturali nella cui sintassi si definiscono:

- l'insieme di parole valide, divise in categorie (articoli, nomi, verbi,...)
- le regole per costruire frasi costituite da parole

nei compilatori, formalmente si da:

- una descrizione delle parti elementari, i lessemi (fase dell'analisi lessicale)
- una descrizione della struttura generale, a partire dai lessemi (fase dell'analisi sintattica)

La separazione rende più efficiente l'analisi del testo.

2.2.1 Analisi lessicale (scanner, lexer)

Nella stringa di caratteri vengono riconosciuti i *lessemi* e per ogni lessema viene costituito un *token*

token: (categoria sintattica, valore-attributo)

Esempio: data la stringa

$$x = a + b * 2;$$

viene generata la sequenza di token:

```
[(identifier, x),(operator, =),(identifier, a),(operator, +),
(identifier, b),(operator, *),(literal, 2),(separator, ;)]
```

I lessemi sono dunque divisi in classi, per esempio:

- identificatori
- letterali
- parole chiave
- separatori
- ...

e ogni classe ha una sua sintassi per poter definire quali stringhe le appartengono. La sintassi di ogni classe è definita usando un'*espressione regolare*.

2.3 Linguaggi regolari

Sia $\mathcal{A}$ un alfabeto (insieme di simboli base, per esempio {0,1}).

Definiamo linguaggio un insieme di stringhe (chiamate anche parole) composte di simboli appartenenti ad $\mathcal{A}$.

2.3.1 Operazioni sui linguaggi regolari

Sui linguaggi regolari sono definite alcune operazioni di base:

- l'**unione** tra linguaggi: $L \cup M$
- la **concatenazione** tra linguaggi: $LM = \{st \mid s \in L, t \in M\}$
- la **chiusura di Kleene**[3]: $L^* = \{s_1 s_2 \dots s_n \mid \forall i \in \{1, \dots, n\}\,,\ s_i \in L\}$

I linguaggi regolari sono chiusi rispetto alle operazioni appena descritte.

Attraverso queste operazioni è possibile definire l'insieme delle espressioni regolari, chiamato anche algebra di Kleene, le quali permettono di rappresentare sinteticamente i linguaggi regolari.

2.3.2 Sintassi delle espressioni regolari

Oltre alle operazioni base, le espressioni regolari possono avere anche le parentesi tonde () per determinare l'ordine di applicazione. L'uso delle parentesi a volte però può risultare eccessivo e ridondante, per evitare ciò si utilizzano delle convenzioni:

- la concatenazione e l'unione sono associative, per cui:

$$L(MN) = (LM)N = LMN$$

- esiste un ordine di precedenza tra gli operatori:
  - dalla maggiore alla minore:

$$*\ ,\ \cdot\ ,\ |$$

$$a \,|\, bc^* = (b\,(c^*))$$

2.3.3 Esempi di espressioni regolari e stringhe generate

- $a \,|\, b^* = a \,|\, (b^*) \rightarrow \{"a", "b", "bb", "bbb", \dots\}$
- $(a \,|\, b)^* \rightarrow \{\varepsilon, "a", "b", "aa", "ab", "ba", "bb", "aaa", \dots\}$
- $a \,|\, bc^* = a \,|\, (b(c^*)) \rightarrow \{"a", "b", "bc", "bcc", "bccc", \dots\}$
- $ab^*(c \,|\, \varepsilon) \rightarrow \{"a", "ac", "ab", "abc", "abb", "abbc", \dots\}$

[3]L'idea della chiusura di Kleene è piuttosto semplice anche se può sembrare complessa inizialmente, molto semplicemente afferma che prendendo zero o più parole appartenenti ad un linguaggio e concatenandole insieme ottengo un nuovo linguaggio.

- $(0|(1(01^*0)^*1))^* \rightarrow \{\varepsilon, "0", "00", "11", "000", "011", "110", "0000", "0011", "0110", "1001", "1100", "1111", "00000", \dots\}$
  tutti i numeri, scritti in base 2, che sono multipli di 3.

Ogni espressione regolare rappresenta un linguaggio:

- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
- $\mathcal{L}(a) = \{"a"\}$
- $\mathcal{L}(L \,|\, M) = \mathcal{L}(L) \cup \mathcal{L}(M)$
- $\mathcal{L}(L\ M) = \mathcal{L}(L)\,\mathcal{L}(M)$
- $\mathcal{L}(L^*) = (\mathcal{L}(L))^*$

2.3.4 Estensioni

Le estensioni sono delle operazioni addizionali che vengono aggiunte al pool definito nel sottoparagrafo 2.3.1 a pagina 34 solamente per scrivere certe cose in modo più compatto, di fatto non permettono di denotare più linguaggi.

- chiusura positiva: $L^+ = L\,L^*$
- zero o un'istanza: $L? = \varepsilon \,|\, L$
- n concatenazioni di stringhe in L: $L\{n\} = \underbrace{L\,L\,\dots\,L}_{\text{n}}$
- uno tra: $[acdz] = a \,|\, c \,|\, d \,|\, z$
- range: $[a - z] = a \,|\, b \,|\, c \,|\, \dots \,|\, z$
- opposto: $[\hat{}\, a - z] \rightarrow$ tutti i caratteri meno le lettere minuscole

2.3.5 Risultati teorici

Teorema 2.3.5.1 (Teorema di equivalenza) *I linguaggi regolari possono essere descritti equivalentemente tramite:*

- *espressioni regolari*
- *grammatiche regolari*
- *automi finiti non deterministici, NFA*
- *automi finiti deterministici, DFA*

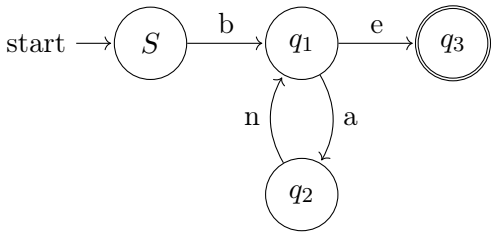
Teorema 2.3.5.2 (Teorema di minimalità) *Esiste l'automa deterministico minimo (minor numero di stati)*

2.3.6 Automi deterministici

Un automa a stati finiti deterministico (DFA) è una quintupla $\langle Q, \Sigma, \delta, q_0, F \rangle$ dove:

- Q è un insieme finito di stati
- Σ è un alfabeto
- $\delta : Q \times \Sigma \to Q$ è la funzione di transizione
- q_0 è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali

La figura seguente è un esempio di automa a stati finiti deterministico deterministico (DFA) che accetta il linguaggio $b(an) + e$.

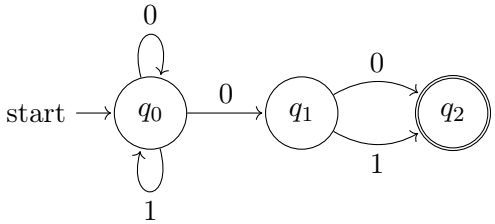


Una stringa x è detta essere accettata da un DFA M se si conclude in uno stato finale. Un linguaggio L è detto regolare se è accettato da qualche DFA.

2.3.7 Automi non deterministici

Un automa a stati finiti (NFA) è una quintupla $\langle Q, \Sigma, \delta, q_0, F \rangle$ dove Q, Σ, q_0 e $F \subseteq Q$ mantengono il significato visto per gli automi deterministici, ma la funzione di transizione porta ad un insieme di stati invece che ad uno solo.

La figura seguente è un esempio di automa a stati finiti non deterministico (NFA) che accetta il linguaggio $(0 \mid 1)^*0(0 \mid 1)$.



Una stringa x è detta essere accettata da un NFA M se esiste almeno un percorso che si conclude in uno stato finale.

2.3.8 Applicazione dei risultati teorici

Per costruire un riconoscitore per un'espressione regolare è sufficiente costruire, a partire da essa:

- un NFA equivalente, da questo
- un DFA equivalente, da questo
- il DFA minimo

Dall'automa minimo risulta poi facile costruire un programma che lo simula e che decide dunque se una parola appartiene ad un'espressione regolare.

2.4 Scanner (o lexer) per l'analisi lessicale

Lo scanner deve risolvere un problema più complesso del semplice riconoscimento di una singola espressione regolare, bensì, dati:

- un insieme di espressioni regolari (classi di lessemi, es.: identificatori, numeri, operazioni, ...)
- una stringa

lo scanner deve dividere la stringa in lessemi, distinguendo tra più espressioni regolari. Questo porta ad alcune complicazioni:

- determinare quando termina un lessema
- cosa fare se un lessema appartiene a più classi

generalmente il primo problema viene risolto cercando di riconoscere sempre la stringa più lunga possibile che fa parte di una delle espressioni regolari. Per fare questo a volte c'è bisogno di simboli di *lookahead*, ovvero bisogna andare un po' più avanti nella stringa per verificarne l'appartenenza. Di seguito un esempio:

- la stringa `'3.14e+5a'` viene interpretata come `'3.14e+5' 'a'`. La notazione esponenziale viene quindi riconosciuta quando c'è un numero seguito da `'e+'`, seguito da un altro numero
- nel caso della stringa `'3.14e+sa'`, però, dopo i caratteri `'e+'` non c'è una costante numerica, dunque per decidere che la stringa più lunga che si può riconoscere è `'3.14'`, bisogna arrivare tramite due simboli di lookahead fino a `'3.14e+'`

Il secondo problema ha delle soluzioni specifiche, che dipendono dall'implementazione, vedremo quella di (F)lex nel sottoparagrafo 2.5.1 a pagina 38.

2.4.1 Come costruire uno scanner

Per costruire uno scanner si parte dalle espressioni regolari definite per ogni token del linguaggio:

- si costruisce un automa per ognuna
- sulla stringa in input:
  - si simula l'esecuzione in parallelo di tutti gli automi
  - un lessema verrà riconosciuto quando nessun automa può continuare e dunque dall'automa che ha riconosciuto la stringa più lunga

La costruzione degli scanner può essere automatizzata.

2.5 Generatori di scanner (analizzatori lessicali)

Si tratta di programmi che prendono come dati un insieme di espressioni regolari e restituiscono un programma che riconosce i lessemi nella sua stringa di ingresso. È possibile specificare che al riconoscimento di un lessema appartenente ad una determinata espressione regolare sia eseguita una certa azione (per esempio la creazione del token per quel lessema), così da realizzare un analizzatore lessicale completo.

2.5.1 (F)LEX

Lex è un diffuso generatore di scanner disponibile su molte distribuzioni Unix (Linux). Prende in input un file di testo con la seguente struttura:

```
1  definizioni   (opzionale)
2  %%
3  regole
4  %%
5  funzioni ausiliarie (opzionale)
```

La parte concettualmente più importante del sorgente sono le regole, cioè una serie di coppie

```
espressione-regolare  azione
```

dove le azioni sono un frammento di codice C[4], istruzioni multiple appaiono tra { }. Un'azione viene eseguita quando viene riconosciuta la corrispondente espressione.

[4]Esistono strumenti equivalenti per gli altri linguaggi di programmazione.

Esempio

```
1  %%
2  aa      printf("2a")
3  bb+     printf("manyb)
4  c       printf("cc")
```

dove `aa`, `bb+` e `c` sono espressioni regolari e a destra ci sono le funzioni in C da eseguire quando una di quelle espressioni viene riconosciuta.

Viene dunque generato un programma che:

- modifica coppie di a
- modifica sequenze di b, lunghe almeno due caratteri
- raddoppia le c

Funzionamento

- si considerano tutte le espressioni regolari e si seleziona quella con il match più lungo, la parte lookahead conta nella misura
- a parità di lunghezza, si segue l'ordine in cui sono state definite le regole
- i caratteri non riconosciuti da nessuna espressione regolare vengono stampati in uscita inalterati

Nel codice delle regole si può far riferimento ad alcune variabili standard predefinite:

yytext stringa (array) contenente il lessema riconosciuto, puntatore al primo carattere

yyleng quanti caratteri contiene il lessema riconosciuto

yyval usata per passare parametri con il parser, come per esempio il token appena creato

2.5.2 Definizioni

La clausola *definizioni* permette di definire alcune espressioni regolari associandogli un nome, vengono espresse nella forma:

```
nome    espressione-regolare
```

Per esempio:

```
letter   [a-zA-Z]
digit    [0-9]
number   {digit}+
```

Notare come nella definizione di number si sia scritto `{digit}+` invece di `digit+`, nel secondo caso le stringhe accettate sarebbero state tutte quelle composte dalla stringa `digi` seguita da una o più `t`.

Le definizioni possono essere usate per la scrittura delle regole.

2.5.3 Sintassi delle espressioni regolari in C

- Metacaratteri: `* | ( ) + ? [ ] - ^ . $ { } / \ " % < >`
- `{ide}` con `ide` identificatore di espressione regolare
- `e{n,m}` con `n` e `m` numeri naturali, indica da n a m ripetizioni di `e` espressione regolare, anche `e{n,}`, `e{,n}` e `e{n}` sono accettate e significano rispettivamente almeno n, al più n ed esattamente n ripetizioni di `e`
- `[^abd]` : tutti i caratteri esclusi `a b d`
- `\n` : newline, `\s` : spazio generico, `\t` : tab
- `\*` : il carattere `*`, la barra \ serve a trasformare un metacarattere in carattere
- `"a+b"` : la sequenza di caratteri `a+b` (i caratteri all'interno di `" "` non vengono interpretati come metacaratteri)
- `.` : tutti i caratteri meno newline
- `^` : inizio riga
- `$` : fine riga

2.5.4 Funzioni ausiliarie

Nel caso in cui si voglia che il programma creato faccia qualcosa di più o si voglia ridefinire delle funzioni usate da Lex, è possibile definire delle *funzioni ausiliarie*, che vanno messe in coda al programma.

In più, nella parte delle *definizioni* è possibile mettere del codice C che si vuole sia in testa al programma generato. In questo caso va inserito tra `%{ }%`. Inserire del codice in testa può essere utile, per esempio, per definire delle variabili a cui ci si può riferire dal codice delle *regole*, come vedremo nel prossimo esempio.

Esempio

```
%{
    int val = 0;
%}
separatore [ \t\n]

%%
0   {val = 2 * val;}
1   {val = 2 * val + 1}
{separatore}+  {printf("%d", val); val = 0;}
```

Sostituisce sequenze rappresentanti numeri binari con il loro valore scritto in decimale. Esempio particolare: se la stringa in input è `01a1`, la stringa in output sarà `a3` perché `a` fa parte di un'espressione regolare non specificata, per cui viene stampata direttamente e inalterata.

Uso standard: creazione di un token per un identificatore

```
cifra          [0-9]
lettera        [a-zA-Z]
identificatore {lettera}({cifra} | {lettera})*
%%
{identificatore}  printf("(IDE,%s)", yytext);
```

Sostituisce il lessema con il token.

2.5.5 Altri esempi

Cifrario di Cesare

```
%%
[a-z]  {char ch = yytext[0];
        ch += 3;
        if (ch > 'z') ch -= ('z' + 1 - 'a');
        printf("%c", ch);
       }
[A-Z]  {char ch = yytext[0];
        ch += 3;
        if (ch > 'Z') ch -= ('z' + 1 - 'A');
        printf("%c", ch);
       }
%%
```

I due codici sono indipendenti, dunque si possono ridefinire le stesse variabili e non serve farlo nella parte di *definizione*.

Conta caratteri

```
%{
int charcount = 0, linecount = 0;
%}
%%
.   charcount++;
\n  {linecount++; charcount++;}
%%
void yyerror(const char *str)
    {fprintf(stderr, "errore: %s\n", str);}
int yywrap() {return 1;} /* funzioni usiliarie */
void main(){
    yylex();
    printf("There were %d characters in %d lines\n",
        charcount, linecount);
}
```

Di seguito alcuni commenti.

- Ricordiamo che l'espressione `'.'` rappresenta un qualsiasi carattere che non sia un `'\n'`.
- È possibile ridefinire il main di default del programma, che contiene solamente l'operazione `'yylex()'`.
- L'operazione `'yylex()'` chiama l'analizzatore lessicale e lo mette in funzione.
- `yyerror()` è la funzione che viene chiamata in caso di errore, tipicamente stampa il messaggio di errore usando la stringa argomento. Può essere ridefinita.
- `yywrap()` è la funzione che viene chiamata a fine file e restituisce 0 o 1 per dire che è la fine del programma principale.

2.5.6 Uso di Flex

```
> flex sorgente.l
```

`.l` indica che è un file lex, con la struttura di cui abbiamo parlato. Il comando genera un programma C `lex.yy.c`, compilabile con il comando:

```
> gcc lex.yy.c -ll
```

in `lex.yy.c` viene creata una funzione `yylex()`

- chiamata dal programma "parser"
- legge un lessema ed esegue l'azione corrispondente

l'opzione `-ll` è necessaria per creare un programma stand-alone

- collega alcune librerie
- con le definizioni di `main, yywrap, yyerror`
- non necessaria se inserisco nel file lex le relative definizioni

Utilizzabile per automatizzare il text editing.

2.6 Analizzatore sintattico (o parser) per l'analisi sintattica

L'analizzatore sintattico, a partire da:

- una grammatica libera dal contesto
- la stringa di token generata nella fase precedente

costruisce l'albero di derivazione della stringa, che parte dal simbolo iniziale

2.6.1 Automi a pila

Le grammatiche libere dal contesto possono essere riconosciute da automi a pila *non* deterministici. Sono automi con le seguenti caratteristiche:

- un insieme di stati finito
- una pila (stack), in cui inserire elementi finiti
- un passo di computazione è deciso in base a:
  - lo stato corrente
  - il simbolo in testa alla pila
  - il simbolo in input
- e decide:
  - il nuovo stato
  - la sequenza di simboli da rimuovere e/o inserire nella pila

Una stringa viene detta accettata da un automa a pila se, alla fine della scansione, ci si trova in uno stato finale e la pila è vuota.

Per gli automi a pila non vale l'equivalenza tra deterministici e non deterministici, in quanto con i secondi si possono riconoscere più linguaggi rispetto ai primi. Per le grammatiche libere sono necessari, in generale, automi non deterministici. Dal punto di vista pratico però ci sono alcune criticità:

- un automa a pila non deterministico, simulato tramite backtracking (ovvero simulando ogni possibile scelta), porta a complessità esponenziali. Esistono comunque due algoritmi [Earley, Cocke–Younger–Kasami] capaci di riconoscere qualsiasi linguaggio libero in tempo $O(n^3)$
- mentre un automa a pila deterministico risolve la sua classe di problemi in tempo lineare

In pratica:

- la complessità $O(n^3)$ non è accettabile per un compilatore visto che deve processare centinaia di migliaia di token
- ci si limita quindi ai linguaggi riconoscibili da automi a pila deterministici, in modo che possano essere riconosciuti in tempo lineare
- è comunque una classe di linguaggi sufficientemente ricca da contenere quasi tutti i linguaggi di programmazione (C e C++ sono un'eccezione, in quanto è possibile scrivere delle espressioni più complesse, come `'x * y;'`, che può essere interpretata in due modi: come dichiarazione della variabile `'y'` come tipo puntatore a `'x'` oppure il prodotto `'x * y'`. L'automa a pila non sa distinguere tra le due opzioni. Per risolvere bisogna correggere manualmente la costruzione automatica del parser.

Esistono due tipi di automi a pila: LL e LR, che lavorano in maniera diversa.

2.6.2 Automi a pila LL(n)

Sono la classe di automi a pila meno usati. Costruiscono l'albero di derivazione in modo top-down:

- a partire dal simbolo iniziale della grammatica
- esaminando al più n simboli della stringa non consumata (lookahead)
- si determina la prossima regola (espansione) da applicare

Spiegazione del procedimento: il parser guarda l'input e vede che strada fare per raggiungere quel terminale. Si basa su una tabella che indica cosa fare in base al simbolo in testa alla pila e ai primi n simboli di input non ancora consumati, normalmente n=1 (aumentando il lookahead si possono riconoscere più classi linguaggi). Determina dunque la prossima azione da svolgere tra le seguenti possibilità:

- applicare una regola di riscrittura, espandendo o contraendo la pila
- consumare un simbolo in input e in testa alla pila (se coincidono)
- generare un segnale di errore (stringa rifiutata)
- accettare la stringa (quando input e pila sono vuoti)

Esempio di parsing

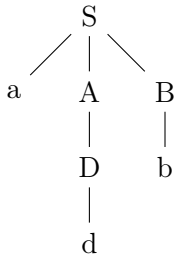
Data la grammatica:

$$
\begin{aligned}
&S \to aAB \\
&A \to C \mid D \\
&B \to b \\
&C \to c \mid \varepsilon \\
&D \to d
\end{aligned}
$$

la stringa adb viene riconosciuta con i seguenti passi:

OUTPUT	PILA	INPUT
Start	S$	adb$
S → aAB	aAB$	adb$
	AB$	db$
A → D	DB$	db$
D → d	dB$	db$
	B$	b$
B → b	b$	b$
	$	$
OK!		

L'albero generato è il seguente:

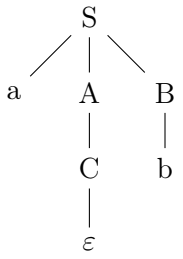


Esempio di parsing con stringa rifiutata

Sia la grammatica la stessa dell'esempio precedente, la stringa abb viene rifiutata con i seguenti passi:

OUTPUT	PILA	INPUT
Start	S$	abb$
S → aAB	aAB$	abb$
	AB$	bb$
A → C	CB$	bb$
C → ε	B$	bb$
B → b	b$	bb$
	$	b$
Errore!		

L'albero generato prima dell'errore era:



Automi LL

Data una grammatica, è facile capire se è una grammatica LL(1) e costruire l'automa corrispondente, però spesso le grammatiche non sono LL(1) nel modo in cui sono formulate, quindi bisogna prima trasformarle in modo da poter costruire l'automa corrispondente.

Significato del nome LL(n)

Le L indicano due aspetti della costruzione dell'albero:

- si esamina la stringa da sinistra verso destra (from **L**eft to right)
- si costruisce la derivazione **L**eftmost

La n indica i simboli di lookahead.

Una derivazione è sinistra (leftmost) se ad ogni passo si espande sempre il non terminale più a sinistra:

$$S \to aAB \to aDB \to adB \to adb$$

Una derivazione è invece destra (rightmost) se ad ogni passo si espande sempre il non terminale più a destra:

$$S \to aAB \to aAb \to aDb \to adb$$

2.6.3 Automi a pila LR

Sono la classe di automi più usata per i parser. Si distinguono dagli automi a pila LL(n) perché costruiscono l'albero di derivazione seguendo l'approccio bottom-up:

- a partire dalla stringa in input
- applicano una serie di contrazioni (regole al contrario)
- fino a contrarre tutto l'input nel simbolo iniziale della grammatica

Gli automi LR sono un po' più complicati da costruire rispetto agli LL, ma riconoscono più linguaggi.

Spiegazione del procedimento: ad ogni passo si sceglie tra un'azione di

- shift: si inserisce un token in input nella pila
- reduce: si riduce la testa della pila applicando una riduzione al contrario

Nella pila si introduce una coppia <simbolo della grammatica, stato> e l'azione da compiere viene decisa guardando:

- la componente stato in testa alla pila (non serve esaminarla oltre)
- n simboli in input, per l'automa LR(n)

Esempio di parsing

Data la grammatica:

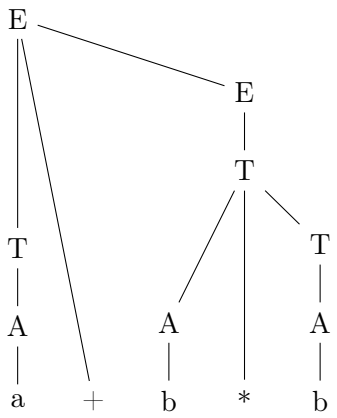
$$E \to T \mid T + E \mid T - E$$
$$T \to A \mid A * T$$
$$A \to a \mid b \mid (E)$$

la stringa a + b * b viene riconosciuta con i seguenti passi:

PILA	INPUT	AZIONE	OUTPUT
$	a + b * b$	shift	
$a	+ b * b$	reduce	A → a
$A	+ b * b$	reduce	T → A
$T	+ b * b$	shift	
$T +	* b$	shift	
$T + b	* b$	reduce	A → b
$T + A	* b$	shift	
$T + A *	b$	shift	
$T + A * b	$	reduce	A → b
$T + A * A	$	reduce	T → A
$T + A * T	$	reduce	T → A * T
$T + T	$	reduce	E → T
$T + E	$	reduce	E → T + E
$E	$	stop	OK!

Si inserisce nella pila un simbolo alla volta finché non si riesce ad applicare una regola al contrario.

L'albero generato è il seguente:



Automi LR

Esiste un algoritmo che, a partire da una grammatica libera L, permette di:

- stabilire se L è LR (in genere non serve riscriverla come per gli automi LL)
- costruire l'automa a pila relativo

Questi algoritmi sono abbastanza sofisticati e per niente ovvi da implementare. Ci sono tre versioni di questo algoritmo, che differiscono per:

- complessità della costruzione
- numero degli stati dell'automa a pila generato (direttamente proporzionale alla complessità dell'algoritmo)
- ampiezza dei linguaggi riconoscibili

Classificazione delle tre versioni:

SLR(n) il più semplice da costruire

LALR(n) stesso numero di stati di SLR, ma più complicato dunque più linguaggi riconoscibili

LR(n) complicato come LALR ma più stati, dunque più linguaggi riconoscibili

Il parametro n indica il numero di caratteri di lookahead.

LALR è il più equilibrato, il compromesso ideale tra numeri di stati e varietà di linguaggi riconosciuti.

2.7 Generatori di parser (analizzatori sintattici)

I generatori di parser sono dei programmi che, a partire da una grammatica libera e da un insieme di regole, generano un parser.

2.7.1 Yacc (Yet Another Compiler Compiler)

È un generatore di parser LR tra i più diffusi che si interfaccia con Lex per la parte lessicale (Paragrafo 2.5.1 a pagina 38).

Yacc riceve in input:

- la descrizione di una grammatica libera
- un insieme di regole da applicare ad ogni riduzione, ovvero del codice C

e restituisce in uscita:

- il programma C[5] che implementa il parser simulando l'automa a pila LALR
- l'albero di derivazione corrispondente oppure altre cose, in base a cosa viene specificato nel codice C associato alle regole

La struttura del codice Yacc è la seguente:

```
1  %{prologo%}
2  definizioni
3  %%
4  regole
5  %%funzioni ausiliarie
```

molto simile a quella di Lex.

Alle produzioni della forma

$$nonterm \rightarrow corpo\_1 \mid \ldots \mid corpo\_k$$

in Yacc vengono associate delle regole:

```
1  nonterm : corpo_1 {azione semantica_1}
2          ...
3          | corpo_k {azione semantica_k}
4         ;
```

dove `{azione semantica}` è il codice C che viene eseguito ogni volta che viene eseguita la riduzione corrispondente.

Esempio

```
exp : num '*' fact {Ccode}
```

dove `Ccode` è un codice C che tipicamente, a partire dai valori calcolati in precedenza di `num` e `fact`, calcola il valore da associare a `exp`. Per esempio a `num` viene associato l'albero sintattico che mostra come da `num` si può ottenere un certo insieme di stringhe, a `fact` viene associato un altro albero sintattico e il `Ccode` dice qual è l'albero sintattico che deve essere associato a `exp`.

Esempio: valutazione espressioni aritmetiche

Costruiamo un programma che valuta una serie di espressioni aritmetiche divise su più righe di input e restituisce il valore finale. Le espressioni sono composte da:

[5] Esistono programmi equivalenti a Yacc per costruire parser in altri linguaggi

- costanti numeriche: numeri positivi e negativi
- le quattro operazioni
- parentesi

valgono le usuali regole di precedenza tra operazioni.

```
1   /* PROLOGO */
2   %{
3   #include "lex.yy.c"
4
5   void yyerror(const char *str){
6       fprintf(stderr, "errore: %s\n", str);}
7   int yywrap() {return 1;}
8   int main() {yyparse();}
9   %}
10
11  /*DEFINIZIONI*/
12  %token NUM
13
14  %left '-' '+'
15  %left '*' '/'
16  %left NEG  /* meno unario */
17
18  /* REGOLE E AZIONI SEMANTICHE */
19  %%
20              /* si inizia con il simbolo iniziale */
21  input :     /* empty, input vuoto */
22        | input line
23  ;
24  line : '\n'
25       | exp '\n'   {printf("The value is %d \n", $1);}
26  ;
27  exp : NUM  {$$=$1;}
28      | exp '+' exp {$$=$1+$3;}
29      | exp '-' exp {$$=$1-$3;}
30      | exp '*' exp {$$=$1*$3;}
31      | exp '/' exp {$$=$1/$3;}
32      | '-' exp %prec NEG  {$$=-$2;}
33      | '(' exp ')'        {$$=$2;}
34  ;
```

A riga 21 si indica che l'input può essere vuoto oppure input seguito da una linea, quindi in sostanza l'input è una sequenza di linee.

A riga 24 si definisce cos'è una linea, ovvero un a capo oppure un'espressione seguita da un a capo. Quando il programma riconoscerà una linea stamperà il valore associato a `exp`. Ricordandoci che l'approccio è bottom-up, sappiamo che quando ci troveremo ad applicare questa regola al contrario avremo già esaminato `exp` e gli avremo associato un valore.

Dalla riga 27 in poi viene descritto quale deve essere il valore associato a `exp` (ovvero il valore finale dell'espressione) a partire dalle sue componenti. `$$` indica il valore che verrà restituito. I numeri dopo il `$` corrispondono al n-esimo sotto-termine e sono delle variabili interne.

La grammatica usata è ambigua, per disambiguarla vengono specificate dalla riga 14 delle priorità. Sono in ordine crescente di precedenza, dunque per esempio prodotto e divisione hanno precedenza su somma e sottrazione e prodotto e divisione hanno la stessa precedenza tra loro. `%left` indica che si associa a sinistra. `NEG` è un operatore fittizio che viene sfruttato a riga 32 per impostare una priorità più alta (quella relativa alla posizione di `NEG` nella definizione) usando `%prec NEG`.

Nella parte delle definizioni, inoltre, bisogna definire quali sono i token (gli elementi base dell'espressione). Qui è stato definito solo `NUM`, ma anche `- + * / ( )` sono token, non vengono esplicitati perché singoli caratteri tra apici vengono interpretati come token, codificati con il loro codice ASCII, mentre gli altri token sono codificati con un intero >257. È stato fatto in questo modo perché rende la scrittura più sintetica, altrimenti erano da esplicitare. I token diventano i simboli terminali della grammatica libera in Yacc.

La parte del prologo, da riga 1 a riga 9, come in Lex contiene eventuali ridefinizioni di procedure standard. `yyparse();` è la funzione che implementa il parser che si comporta come specificato.

Codice Lex associato

Il programma precedente, per funzionare, deve ovviamente prendere una sequenza di token che è generata dall'analizzatore lessicale associato. Nel caso di Yacc la sequenza viene generata dal codice Lex. Il seguente programma fornisce delle funzioni che vengono chiamate dal programma Yacc quando vede un token.

```
1  %{
2  #include <stdio.h>
3  #include "y.tab.h"
4  %}
5
6  %%
7  [ \t];          // ignore all whitespace
8  [0-9]           {yylval = atoi(yytext); return NUM;}
9  \n              {return *yytext;}
```

```
10  "+"           {return *yytext;}
11  [\-\*\/\(\)]  {return *yytext;}
```

A riga 8 si dice che il token associato a quell'espressione è `NUM` (lo stesso del programma Yacc, i due programmi sono in grado di interfacciarsi tramite una tabella dove sono contenute le definizioni). L'operazione da eseguire ritorna `NUM` che è l'intero che identifica il tipo di token e il valore numerico corrispondente (`atoi()` converte la stringa nella variabile `yytext` in un intero, assegnato poi alla variabile `yyval`). Nelle righe successive viene riconosciuto un carattere e si vuole che venga restituito il carattere stesso come valore finale.

Esempio: sintassi dei comandi ad un termostato

In questo esempio il testo è composto da una serie di comandi che si possono dare ad un impianto di riscaldamento. Yacc si occupa di interpretarli ed eseguirli (per semplicità di fatto stampa semplicemente il comando come stringa).

File LEX

```
1   %{
2   #include <stdio.h>
3   #include "y.tab.h"
4   %}
5   %%
6   [0-9]+        {yyval=atoi(yytext); return NUMERO;}
7   riscaldamento return TOKRISCALDAMENTO;
8   acceso|spento {yyval=strcmp(yytext,"spento"); return STATO;}
9   obiettivo     return TOKOBIETTIVO;
10  temperatura   return TOKTEMP;
11  [ \t\n]+      /* ignora spazi bianchi e fine linea */;
12  %%
```

A riga 7 l'operazione restituisce il token nella sua rappresentazione come intero. A riga 8 `yyval` sarà un booleano in base a se è acceso o spento.

File Yacc

```
1   %{
2   #include <stdio.h>
3   #include <string.h>
4
5   void yyerror(const char *str)
6   {fprintf(stderr, "errore: %s\n", str);}
7   int yywrap() {return 1;}
```

```
8   int main() {yyparse();}
9   %}
10
11  %token NUMERO TOKRISCALDAMENTO STATO TOKOBIETTIVO TOKTEMP
12
13  %%
14
15  comandi: /* vuoto */
16         | comandi comando
17         ;
18  comando: interruttore_riscaldamento
19         | imposta_obiettivo
20         ;
21  interuttore_riscaldamento: TOKRISCALDAMENTO STATO
22         {if($2)    printf("\t Riscaldamento acceso \n");
23             else   printf("\t Riscaldamento spento \n");}
24         ;
25  imposta_obiettivo: TOKOBIETTIVO TOKTEMP NUMERO
26         {printf("\t Temperatura impostata a %d \n", $3);}
27         ;
```

Problema

Il tipo delle variabili `$` è `YYSTYPE`, che di default è intero, ma se si vuole per esempio costruire l'albero di derivazione serve un altro tipo. Per risolvere il problema di inserisce nel sorgente Yacc il seguente codice:

```
1   %union{
2   nodeType   *expr;
3   int        value;
4   }
```

che dice che la variabile `yyval` e le variabili `$$,$1,...` possono essere di più tipi diversi, dando una lista di possibilità.

Dunque nella parte delle definizioni (prologo) del file Yacc,per ogni token si deve specificare quale tipo usare:

```
1   %token <value> INTEGER, CHARCON; /* terminali */
2   %token <expr>  expression; /* non temrinali */
```

2.7.2 Differenze tra Yacc e LALR

Yacc produce codice C che implementa un automa LALR, ma non esiste uno stretto controllo che la grammatica sia LALR, infatti esse vengono accettate, però:

- si costruisce un automa a pila dove alcune scelte sono non deterministiche, ovvero in alcuni casi si deve scegliere l'operazione da eseguire tra shift e reduce senza indicazioni precise
- il codice ne sceglie una, ma si possono fare scelte sbagliate rifiutando parole valide

Per risolvere il problema dell'ambiguità si possono usare automi a pila non deterministici, ma noi vogliamo usare un automa deterministico, per cui bisogna definire delle priorità per togliere le ambiguità (Riga 14 del codice dell'esempio a pagina 50).

2.7.3 Creazione del codice

```
> lex file.l
> yacc -d file.y
> cc lex.yy.c y.tab.c -o fileEseguibile
```

in alternativa:

```
> flex file.l
> bison -d file.y
> gcc lex.yy.c y.tab.c -o fileEseguibile
```

in alternativa inserisco

```
#include "lex.yy.c"
```

nel prologo Yacc, e uso il comando

```
> cc y.tab.c -o fileEseguibile
```

L'opzione `-d` forza la creazione del file `y.tab.h`

Capitolo 3

Nomi e ambiente

3.1 Astrazione attraverso i nomi

L'idea dell'astrazione sta alla base dello sviluppo del software, permette di gestire la complessità dei programmi e consiste sostanzialmente in nascondere i dettagli per mettere in evidenza ciò che è importante. Un tipo di astrazione è anche quello effettuato dalla macchina astratta, essa infatti mostra al programmatore solo le istruzioni che la macchina implementa senza richiedere allo stesso la conoscenza dei meccanismi utilizzati nell'implementazione. Il modo più diretto per astrarre consiste nell'utilizzare i **nomi**, presenti perfino in linguaggi di basso livello come assembly attraverso i tag e con i quali era possibile, in combinazione con l'utilizzo di goto, astrarre sulle procedure. I nomi consistono in una sequenza significativa di caratteri usata per denotare qualcos'altro, gli stessi permettono di rappresentare sinteticamente e mnemonicamente[1] elementi quali:

- valori (costanti)
- locazioni di memoria (variabili)
- pezzi di codice (procedure)
- operatori (funzioni base)
- etc...

Esempio:

```
1  const pi = 3.14;
2  int x = pi + 1;
3  void f(){...};
```

[1]Per mnemonicamente intendiamo che sono facili da ricordare dato che il nome generalmente rappresenta il significato dell'elemento astratto.

dove:

- `const`, `int` e `void` sono nomi che rappresentano il tipo
- `pi` è il nome identificatore della costante definita
- `x` è il nome identificatore della variabile x
- `f` è il nome della funzione con codice espresso all'interno delle parentesi
- `in` e `void` sono rispettivamente i nomi dei tipi integer e vuoto
- `+` è il nome della funzione somma

Bisogna notare anche come i nomi non debbano essere usati sempre come identificatori, per esempio i simboli semplici: +, -, <= e ++ sono anch'essi nomi siccome rappresentano delle funzioni.

Riguardo ai nomi, è importante distinguere tra il nome, ovvero la stringa di caratteri vera e propria, e l'oggetto rappresentato, oppure, come vengono chiamati in linguistica, tra significante e significato. Difatti uno stesso oggetto può avere più nomi (*aliasing*), mentre uno stesso nome può denotare oggetti diversi in momenti diversi.

Diciamo che un oggetto è denotabile se ad esso è associabile un nome, diversi linguaggi possono avere diversi tipi di oggetti denotabili.

3.2 Binding e ambiente

Per **legame** (binding) si intende l'associazione esistente tra un nome ed un oggetto, mentre per **ambiente** (environment) si intende l'insieme dei legami esistenti, quest'ultimo dipende dal punto in cui il programma si trova e può dipendere dalla storia del programma, ovvero il codice eseguito in precedenza.

Una prima distinzione per i binding può essere:

- nomi definiti dal linguaggio: tipi primitivi, operazioni primitive, costanti predefinite
- variabili, parametri formali, procedure (in senso lato), tipi definiti dall'utente, etichette, moduli, costanti definite dall'utente, eccezioni

Il binding può avvenire in diversi momenti, tra cui: durante la definizione del linguaggio, durante la scrittura del codice, durante il caricamento del programma in memoria, durante l'esecuzione, etc...

Il binding può essere dunque di due tipi:

- **statico**, se avviene prima dell'esecuzione della prima istruzione
- **dinamico**, se avviene durante l'esecuzione

Notare che un oggetto denotato può essere definito incrementalmente in vari momenti, ovvero alcune informazioni possono essere create a livello statico mentre altre a livello dinamico.

Ad ogni variabile `x` inoltre sono associati:

- **ambiente** definisce quale locazione di memoria contiene il dato di x
- **store** determina il dato effettivo

L'ambiente può essere modificato attraverso le dichiarazioni, creando nuovi nomi e nuovi legami, mentre lo store può essere modificato attraverso le istruzioni di assegnamento.

Alcuni linguaggi di programmazione non prevedono l'esistenza di uno store, come quelli funzionali puri tipo Scheme.

3.3 Dichiarazioni

Le dichiarazioni sono i meccanismi che permettono di creare i legami tra nomi e oggetti, modificando dunque l'ambiente.
Esempio:

```
1  int x = 0;          // dichiarazione di una variabile
2  typedef int T;      // dichiarazione di un tipo
3  int inc (T x){      // dichiarazione di una funzione
4      return x + 1;
5  }
```

Attraverso più dichiarazioni, lo stesso nome può denotare oggetti distinti in punti diversi del programma.

3.4 Blocchi

Nei linguaggi moderni l'ambiente è strutturato e l'elemento principale per gestire questa struttura è il **blocco**, che definiamo come regione del programma che può contenere dichiarazioni locali a quella regione ovvero che, quando si esce dal blocco, ciò che è stato dichiarato non è più valido.

I blocchi possono essere:

- **anonimi**, ovvero blocchi a cui non viene dato un nome
- **legati ad una procedura**, ovvero blocchi a cui viene dato un nome, tipicamente sono le procedure

La notazione varia in base all linguaggio:

`{...}`	`C, Java`
`begin ... end`	`Algol, Pascal`
`(...)`	`Lisp`
`let...in...end`	`Scheme, ML`

3.4.1 Vantaggi dei blocchi

I blocchi portano una serie di vantaggi, uno di questi è la gestione locale dei nomi.

Supponiamo di voler eseguire uno swap tra due variabili senza interferire con altri elementi del codice:

```
1  {
2      int tmp = x;
3      x = y;
4      y = tmp;
5  }
```

in questo modo, anche se la variabile `tmp` fosse già stata dichiarata in un'altra parte del codice, la variabile definita all'interno di questo blocco non interferirebbe con essa e viceversa.

Altri vantaggi dell'uso dei blocchi sono:

- strutturare meglio il programma
- migliorare l'occupazione di memoria: in caso di dichiarazione di variabili che occupano molta memoria, come le matrici, è possibile recuperare tale memoria nel momento in cui si esce dal blocco. Solitamente, però, è un vantaggio poco importante, in quanto la memoria nella maggior parte dei casi non è un grosso problema, in più, creare un blocco significa modificare l'ambiente, dunque eseguire una serie di operazioni rischiando di aumentare i tempi di computazione.
- permettono la ricorsione

3.4.2 Annidamento

I blocchi possono essere annidati:

```
1  {
2      int x = 0;
3      int y = 2;
4      {
5          int x = 1;
6          x = y;
```

```
7       }
8       write(x);
9   }
```

Per quanto riguarda la visibilità, il principio è che è valida la dichiarazione più interna. Le regole di visibilità sono:

- una dichiarazione è visibile nel blocco di definizione e in tutti quelli annidati, a meno di mascheramento
- una nuova dichiarazione per lo stesso nome nasconde, *maschera*, la precedente

Nell'esempio, il valore di `x` che viene scritto è `0`, in quanto la dichiarazione interna di x non interferisce con quella esterna.

Altro esempio

```
1   A:{
2       int a = 1;
3       B:{
4           int b = 2;
5           int c = 3;
6           C:{
7               int c = 4;
8               int d = a + b + c;
9               write(d);
10          }
11      }
12      D:{
13          int e = a + b + c;
14          write(e);
15      }
16  }
```

In questo esempio il blocco `C` maschera la dichiarazione della variabile `c` del blocco `B`. Il blocco `D` non vede i blocchi `B` e `C` e dunque da errore perché per lui `b` e `c` non sono definite.

3.4.3 Utilizzabilità di una dichiarazione

Dov'è utilizzabile una dichiarazione all'interno del blocco in cui essa compare?

- tipicamente a partire dalla dichiarazione e fino alla fine del blocco
- in alcuni linguaggi (Modula3, Python) in tutto il blocco

quindi dichiarazioni come:

```
{
    const a = b;
    const b = 3;
    ...
}
```

possono generare errore oppure no in base al linguaggio, perché `b` viene usata prima di essere dichiarata.

3.4.4 Validità di una dichiarazione

Una dichiarazione può essere valida:

- a partire dalla dichiarazione
- in tutto il blocco

Istruzioni del tipo:

```
{
    const a = 1;
    ...
    procedure foo;
        const b = a;
        const a = 2;
}
```

generano errore in Pascal, C#, dove validità (tutto il blocco) e utilizzabilità (dalla dichiarazione) non coincidono. Tipicamente a `b` viene assegnato `1` (C, Java), può però essergli assegnato `2` (Python).

3.5 Ambiente

L'ambiente (in uno specifico blocco) può essere suddiviso in:

- **ambiente locale**: fa riferimento a tutte le variabili dichiarate nel blocco più interno, al valore associato e ai parametri formali (se dentro una procedura)
- **ambiente globale**: fa riferimento alle variabili che sono visibili all'interno di tutto il programma, quindi quelle dichiarate nel main
- **ambiente non locale, non globale**: fa riferimento alle variabili dichiarate nei blocchi più interni che possono essere usate dai blocchi ancora più interni

3.5.1 Operazioni sull'ambiente

Le operazioni che si possono eseguire sull'ambiente sono:

- creazione di un'associazione nome–oggetto denotato (naming)
  - eseguita all'entrata in un blocco, dichiarazione locale
- distruzione di un'associazione nome–oggetto denotato (unnaming)
  - quando si esce dal blocco le dichiarazioni locali vengono cancellate
- riferimento ad un oggetto denotato mediante il suo nome (referencing)
- disattivazione di un'associazione nome–oggetto denotato
  - quando una dichiarazione di un blocco più esterno viene mascherata in un blocco interno, la variabile resta ma non ci può accedere
- riattivazione di un'associazione nome–oggetto denotato
  - quando si esce dal blocco in cui la variabile era stata mascherata, la variabile viene riattivata e diventa di nuovo usabile

3.5.2 Operazioni sugli oggetti denotabili

Le operazioni che si possono eseguire sugli oggetti denotabili sono:

- creazione (assegnargli una locazione di memoria)
- accesso
- modifica (se l'oggetto è modificabile)
- distruzione

Creazione e distruzione di un oggetto non coincidono con creazione e distruzione dei legami per esso. Alcuni oggetti denotabili, come costanti e tipi, non vengono né creati né distrutti.

3.5.3 Eventi base del binding

La vita dell'oggetto e del rispettivo nome sono caratterizzati dai seguenti eventi:

1. creazione di un oggetto
2. creazione di un legame per l'oggetto
3. riferimento all'oggetto, tramite il legame

4. disattivazione di un legame
5. riattivazione di un legame
6. distruzione di un legame
7. distruzione di un oggetto

Gli eventi da 1 a 7 denotano il tempo di vita di un oggetto, mentre quelli da 2 a 6 il tempo di vita dell'associazione, notare quindi come sia possibile che il tempo di vita di un oggetto non sempre coincida con quello dei suoi legami.

Mostriamo ora un esempio in cui la vita dell'oggetto dura più a lungo di quello del nome.

```
1  var A:integer;
2  procedure P (var X:integer);
3      begin ...
4      end;
5  ...
6  P(A);
```

Quando nella riga 6 viene chiamata la procedura `P(A)`, viene creato un legame tra `X` e l'oggetto legato anche con `A`. Una volta finita la procedura `P()`, l'oggetto rimane in vita dato che ha ancora un legame con A, mentre il legame con X viene distrutto.

Vediamo ora un esempio opposto rispetto al precedente, ovvero dove il tempo di vita dell'oggetto è inferiore a quello del legame.

```
1  int *X, *Y, z;
2  X = (int *) malloc (sizeof (int));
3  Y = X;
4  *Y = 5;
5  free (X);
6  z = *Y;
```

Nel codice proposto, `X` è un puntatore e come oggetto associato ha una locazione di memoria allocata in riga 2 con il comando `malloc()`, la stessa locazione poi viene associata a `Y`, anch'esso un puntatore, e per finire nella locazione di memoria associata ai puntatori viene inserito il valore 5. A questo punto viene eseguito il comando `free(X)` che va a de–allocare la locazione di memoria associata a `X`, quest'ultima però è l'oggetto associato ad entrambi i puntatori `X` e `Y` portando `Y` in una situazione in cui ad esso nessun oggetto è associato provocando un comportamento anomalo nella riga 6.

La situazione appena descritta non è comune nei linguaggi moderni, notare infatti come il codice proposto è scritto in C, e prende il nome di **dangling reference**, ovvero, **riferimenti pendenti**.

3.5.4 Regole di scope

In presenza di procedure, le regole di visibilità (scoping) diventano più complesse. Ci sono due possibilità in base al linguaggio di programmazione scelto:

- scoping statico
- scoping dinamico

Esempio: quale `x` viene incrementato?

```
1   int x = 10;
2   void incx(){
3       x++:
4   }
5   void foo(){
6       int x = 0;
7       incx();
8   }
9   int main(){
10      foo();
11      write(x);
12  }
```

Nel caso dello scoping statico, per i nomi non locali, ci si riferisce sempre alla struttura statica del codice, ovvero com'è scritto al momento della compilazione, dunque l'`x++` della riga 3 si riferisce sempre all'`x` della riga 1, indipendentemente da dove `incx()` viene chiamata, e quindi il risultato finale del programma è `11`.

Nel caso dello scoping dinamico, per i nomi non locali, ci si riferisce sempre alla dichiarazione visibile nel blocco più vicino al momento dell'esecuzione. A riga 7 viene chiamata la procedura `incx()` e in quel momento la dichiarazione più vicina è quella eseguita nella procedura `foo()`, dunque viene incrementata la `x` di riga 6. Il risultato finale del programma è quindi `10` in quanto la `x` globale non viene mai modificata.

Esempio

```
1   int x = 0;
2   void foo(int n){
3       x = x + n;
4   }
5   foo(2);
6   write(x);
7   {
8       int x = 0;
```

```
9       foo(3);
10      write(x);
11  }
12  write(x);
```

Nel caso di scoping statico, a riga 6 stampa 2; a riga 10 stampa 0 perché `foo(3)` non interferisce con la dichiarazione di riga 8 e `write(x)` si riferisce alla `x` più vicina; a riga 12 stampa 5 perché la `x` globale di riga 1 è stata incrementata a riga 5 e a riga 9.

Nel caso di scoping dinamico, a riga 6 stampa 2; a riga 10 stampa 3 perché `foo(3)` si riferisce alla dichiarazione più vicina in quel momento, ovvero quella di riga 8; a riga 12 stampa 2 perché la perché `x` globale di riga 1 è stata incrementata solo a riga 5.

Vantaggi e svantaggi

Lo scoping statico è la scelta più comune nei linguaggi di programmazione perché porta con sé i seguenti vantaggi:

- indipendenza dalla posizione: l'ambiente in cui viene chiamata una funzione non ne modifica il comportamento. Esempio:

```
1       int x = 0;
2       void incx(){
3           x++;
4       }
5       void foo(){
6           int x = 0;
7           incx();
8       }
9       foo();
```

  - `incx()` è dichiarata nello scope della `x` più esterna,
  - `incx()` è chiamata nello scope della `x` più interna,
  - `incx()` può essere chiamata in molti contesti diversi,
  - con lo scoping statico `incx()` si riferisce sempre alla `x` più esterna dunque avrà un comportamento uniforme indipendentemente da dove viene chiamata, mentre con lo scoping dinamico il comportamento varia

  Questo è un grande vantaggio, pensate se si dovesse sempre fare attenzione a dove viene chiamata una certa procedura, magari all'interno di un codice con centinaia di righe. Con lo scoping statico, dal momento della dichiarazione di una funzione, essa avrà sempre lo stesso comportamento ovunque verrà chiamata.

- indipendenza dai nomi locali: con lo scoping dinamico, se si cambia il nome, anche per sbaglio, si modificherà il comportamento del programma, non ha alcun effetto invece con lo scoping statico. Esempio:

```
1       int x = 0;
2       void incx(){
3           x++;
4       }
5       void foo(){
6           int y = 0; // y al posto di x
7           incx();
8       }
9       foo();
```

Lo scoping dinamico ha però un vantaggio: maggior flessibilità nell'uso delle procedure.
Esempio:

Supponiamo che `visualizza()` sia una procedura che

- rende a colore su video un certo testo
- usa come parametro una costante non locale `colore`

Con scoping dinamico è possibile modificare il suo comportamento simulando il meccanismo dei "valori di default":

```
1   {
2       const colore = rosso;
3       visualizza(testo)
4   }
```

dove a riga 2 viene ridefinita la costante colore e quindi la procedura `visualizza()` userà quella al posto della variabile non locale definita in precedenza.

In conclusione:

Scoping statico (statically scoped, lexical scope):

- informazione completa dal testo del programma
- le associazioni sono note a tempo di compilazione
- principi di indipendenza
- più complesso da implementare ma più efficiente

- Algol, Pascal, C, Java, Scheme, ...

Scoping dinamico (dynamically scoped):

- informazione derivata dall'esecuzione
- spesso causa di programmi meno "leggibili"
- più flessibile: modificare il comportamento di una procedura al volo
- più semplice da implementare, ma meno efficiente
  - esistono implementazioni efficienti, ma sono piuttosto complesse
- Lisp (alcune versioni), Perl, APL, Snobol, ...

3.5.5 Aliasing

Quando due o più nomi fanno riferimento allo stesso oggetto si dice che sta avvenendo un **aliasing**, tale fenomeno può avvenire per due diverse cause e comporta una maggiore complessità nel comportamento del programma.

La prima è il passaggio di parametri ad una procedura per riferimento, come mostrato nel codice seguente.

```
1  int x = 2;
2  int foo (ref int y){
3      y = y + 1;
4      x = x + y;}
5  foo(x);
6  write (x);
```

Attraverso il passaggio per riferimento (xedere paragrafo 6.3 a pagina 125) il nome `y` dichiarato in riga 2 prende come riferimento la locazione di memoria associata alla `x` dichiarata in riga 1, di conseguenza abbiamo che all'interno della procedura `foo()` l'oggetto inizialmente referenziato solo da `x` viene referenziato anche da `y`, si verifica quindi un aliasing.

Un'altra possibile fonte di aliasing è l'utilizzo dei puntatori.

```
1  int *X, *Y;
2  X = (int *) malloc (sizeof (int));
3  *X = 0;
4  Y = X;
5  *Y = 1;
6  write(*X);
```

In questo caso il motivo è più semplice da comprendere rispetto al primo caso, a riga 4 infatti assegniamo a `Y` il valore di `X`, essendo però dei puntatori avremo che sia `X` sia `Y` faranno riferimento alla stessa locazione di memoria e referenzieranno dunque lo stesso oggetto.

3.5.6 Overloading

Per **overloading** si intende quel fenomeno dove lo stesso nome ha diversi significati a seconda del contesto in cui viene utilizzato, si tratta inoltre di una forma di polimorfismo.

L'esempio più comune è il simbolo `+`, usato per la funzione predefinita somma, questo simbolo infatti viene usato per:

- somme di interi
- somme di floating-point
- concatenazione di stringhe
- etc...

3.5.7 Costrutto let su scheme

Scheme permette delle dichiarazioni tramite appositi costrutti, ciascuno con diversa validità:

- `let`: non ricorsiva, creazione in blocco del nuovo ambiente
- `let*`: non ricorsiva, creazione sequenziale di una serie di ambienti
- `letrec`: ricorsive e mutuamente ricorsive

Per capire la differenza tra il primo, `let`, ed il secondo, `let*`, prendiamo come esempio i seguenti codici:

```
1  (let ((a 1))
2     (let ((a 2)
3           (b a))
4       b))
5
6  (let ((a 1))
7     (let* ((a 2)
8            (b a))
9       b))
```

In entrambi viene eseguito un primo `let` che crea un ambiente in locale in cui `a = 1`, nello stesso ambiente viene eseguito un altro `let` in cui viene ridichiarata la variabile `a` con `a = 2` e dichiarata la variabile `b` con `b = a`. Le procedure differiscono però nel tipo di `let` utilizzato, nel primo, il `let` senza nulla, le dichiarazioni interne vengono eseguite in contemporanea e isolate, e le dichiarazioni fanno riferimento alle variabili dell'ambiente subito esterno, questo comporta che il valore assunto da `b` sia `1` siccome nell'ambiente subito esterno `a = 1`. Differente è la situazione del secondo codice, ovvero dove il

`let` utilizzato è il `let*`, con questo ad ogni dichiarazione l'ambiente viene modificato, è l'equivalente di fare `let(a 2)let(b a)` e quindi la dichiarazione `b = a` viene fatta con `a` già modificato da `(a 2)`, otteniamo che il risultato quindi è 2.

L'ultimo, `letrec`, permette di definire

3.5.8 Mutua ricorsione

Date due funzioni `f()` e `g()` diciamo che sono **mutualmente ricorsive** se si chiamano l'un l'altra, ovvero ad un certo momento dell'esecuzione di `f()` viene effettuata una chiamata a `g()` e viceversa ad un certo momento dell'esecuzione di `g()` viene effettuata una chiamata ad `f()`.

La mutua ricorsione è problematica per i linguaggi perché richiede di poter utilizzare dei nomi non ancora dichiarati, prendiamo l'esempio seguente.

```
1   {
2       void f(){
3           ...
4           g();   // g() non ancora dichiarata
5           ...
6       }
7       void g(){
8           ...
9           f();
10          ...
11      }
12  }
13  ...
```

Come si vede dall'esempio la funzione `f()` necessita di utilizzare `g()` ma `g()` non viene dichiarata prima della fine di `f()`, anche invertendo l'ordine con cui sono scritte le funzioni il problema persiste, otterremo in questo caso che `g()` chiama `f()` ma `f()` non viene dichiarata prima della fine di `g()`.

La soluzione consiste nel permettere un'eccezione al seguente vincolo:

- un nome deve essere dichiarato prima di essere utilizzato.

La mutua ricorsione può avvenire anche per la definizione di tipo, come nei seguenti esempi.

```
1   type lista = ^elem;
2        elem = record
3             info : integer;
4                 next : lista;
5            end
```

In questo codice pascal dichiariamo `lista` come puntatore ad un tipo `elem`, ma il tipo `elem` non viene dichiarato prima della riga successiva, e lo stesso `elem` utilizza `lista` all'interno della propria dichiarazione.

```
1   struct child {
2      struct parent *pParent;
3   };
4   struct parent {
5      struct child *children[2];
6   };
```

Analoga è la situazione in questo codice c dove la `struct child` ha al suo interno un puntatore a una struct `parent`, ma la `struct parent` ha un puntatore ad una `struct child`.

Per la mutua ricorsione di tipo alcuni linguaggi la permettono, per esempio il C permette di fare una dichiarazione parziale di tipo e dichiarare la struttura utilizzando quella dichiarazione.

```
1   typedef struct elem element;
2   struct elem {
3       int info;
4       element *next;
5   }
```

Analoga è la soluzione per la mutua ricorsione di funzioni, come vediamo nel prossimo segmento di codice.

```
1   void eval_tree(tree t);      /* solo dichiarazione */
2   void eval_forest(forest f){
3       ...
4   eval_tree(t2);
5   }
6   void eval_tree(tree t){      /* definizione completa */
7       ...
8   eval_forest(f2);
9   }
```

Quando ci troviamo in riga 4, al compilatore non importa come la funzione `eval_tree` è composta, ma soltanto che gli argomenti siano giusti e che il risultato ritornati sia trattato in maniera corretta, queste informazioni vengono date attraverso la definizione parziale scritta in riga 1, in questo modo il compilatore può controllare il giusto utilizzo di `eval_tree` senza conoscerne subito l'implementazione, permettendoci quindi di scriverla in seguito , dopo la fine di `eval_forest`.

3.5.9 Moduli

L'idea dei moduli è dividere il codice in componenti diverse, ciascuna contenente una serie di definizioni decidendo quali di esse sono disponibili all'esterno, nascondendo gli altri. Questi moduli in seguito possono essere utilizzati in un altro modulo, a seconda di quale serve si specifica all'inizio del nuovo modulo, il che permette di usare tutte e sole le funzioni e nomi necessari. Si tratta di un meccanismo più sofisticato dei blocchi annidati e delle procedure.

Capitolo 4

Gestione della memoria

La gestione della memoria si occupa di come il compilatore-interprete organizza i dati necessari all'esecuzione del programma. Nel codice macchina scritto a mano e in quello generato dai compilatori, la memoria ha una struttura molto simile.

4.1 Uso della memoria RAM

Nell'uso tipico, il codice ARM prevede una divisione della memoria nei seguenti intervalli consecutivi:

- `0 - 0xFFF`: riservata al sistema operativo
- `0x1000 - ww`: codice del programma (`.text`),
- `ww - xx`: costanti, variabili del programma principale (`.data`)
- `xx - yy`: heap per dati dinamici (liste, alberi), nel caso in cui si richieda memoria aggiuntiva tramite chiamate al sistema operativo
- `yy - zz`: stack di attivazione per le chiamate di procedura

Il registro `r13`, `sp` (stack pointer) punta alla cima dello stack. Il registro `r11`, `fp` (frame pointer) punta al "frame" della procedura in esecuzione.

4.2 Tipi di allocazione della memoria

Ci sono due metodi principali di allocazione della memoria:

- allocazione statica: memoria allocata a tempo di compilazione
- allocazione dinamica: memoria allocata a tempo d'esecuzione, divisa in:

  - pila (stack): oggetti allocati con politica LIFO (Last In First Out)
  - heap: oggetti allocati e de–allocati in qualsiasi momento

normalmente un programma usa tutti e tre i meccanismi.

4.2.1 Allocazione statica

Gli oggetti hanno un indirizzo assoluto, mantenuto per tutta la durata dell'esecuzione. Solitamente vengono allocati staticamente:

- variabili globali
- costanti determinabili a tempo di compilazione
- tabelle usate dal supporto a run-time (per type checking, garbage collection, etc. . .)

Allocazione statica completa

Alcuni linguaggi di programmazione (vecchie versioni di Fortran), usano l'allocazione statica per tutti i dati, in questo caso:

- ad ogni variabile (locale o globale) viene assegnato un indirizzo univoco per tutta l'esecuzione
- le variabili locali di una procedura mantengono il valore anche dopo la fine della procedura

Vantaggi dell'allocazione statica completa:

- semplicità dell'implementazione
- accesso ai dati più veloce, siccome ogni variabile sta sempre nella stessa posizione

Svantaggi dell'allocazione statica completa:

- non si può fare la ricorsione, in quanto le varie chiamate ricorsive di una stessa procedura devono avere ciascuna uno spazio privato di memoria per:
  - una copia delle variabili locali, ogni chiamata ricorsiva le può modificare in modo diverso
  - informazioni di controllo (indirizzo di ritorno)

  ma l'allocazione statica non permette la creazione di molteplici spazi per una sola funzione.

- forza ad usare più spazio del necessario, in quanto costringe ad allocare spazio per tutte le variabili e per tutte le funzioni di tutto il codice. Comporta uno spreco perché magari due funzioni che richiedono molto spazio non vengono mai chiamate contemporaneamente

4.2.2 Allocazione dinamica: stack di attivazione

In generale, a meno di allocazione statica completa, quando parte l'esecuzione del programma viene allocata la memoria che serve al programma principale e si esegue il codice finché c'è una chiamata ad una funzione. In questo caso deve essere allocato dello spazio nello stack di attivazione, che viene chiamato record di attivazione (RdA o frame). Nel momento in cui si esce dalla procedura questo spazio viene de–allocato.

Un record di attivazione di una procedura contiene:

- variabili locali
- parametri in ingresso e uscita
- indirizzo di ritorno (passaggio del controllo)
- link dinamico (al record di attivazione della procedura chiamante)
- link statico (al record di attivazione della procedura genitore, ovvero alla procedura a cui, in presenza di scoping statico, fa riferimento per le variabili non locali)
- risultati intermedi (uno spazio di memoria ausiliario che può essere necessario per valutare espressioni complesse)
- salvataggio dei registri (i registri della CPU contengono dati temporanei, quando il controllo passa ad un'altra procedura quei dati devono essere salvati in memoria)

Analogamente, ogni blocco che non è una procedura deve avere un suo record di attivazione anche se con meno informazioni di controllo:

- variabili locali
- risultati intermedi
- link dinamico (al record di attivazione della procedura chiamante)

Normalmente, nello stack di attivazione vengono occupati prima gli indirizzi alti e poi quelli più bassi. Ricordiamo inoltre che:

- il **frame pointer** (FP) punta all'indirizzo base dell'ultimo record e i dati dell'ultimo frame sono accessibili per offset (determinabile staticamente a tempo di compilazione) rispetto al frame pointer;
  `indirizzo_dato = FP + offset`

- per dati non locali è necessario determinare l'indirizzo base del record del dato, si usa il **link dinamico** che è il puntatore al precedente record sullo stack. Serve per risalire la catena dei link;
- per la gestione dello stack di attivazione si usa anche lo **stack pointer** (SP) che punta alla fine della pila, più precisamente al primo spazio di memoria disponibile.

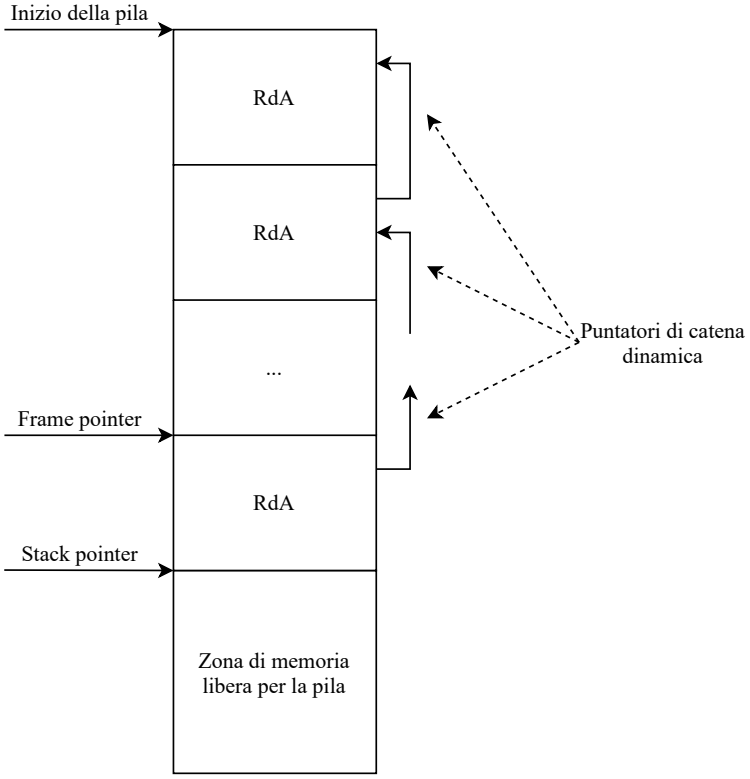


Figura 4.1: Stack di attivazione e i vari puntatori

Esempio: come funziona il meccanismo dei record di attivazione

```
1  {
2      int x = 0;
3      int y = x + 1;
4      {
5          int z = (x + y) * (x - y);
6      }
7  }
```

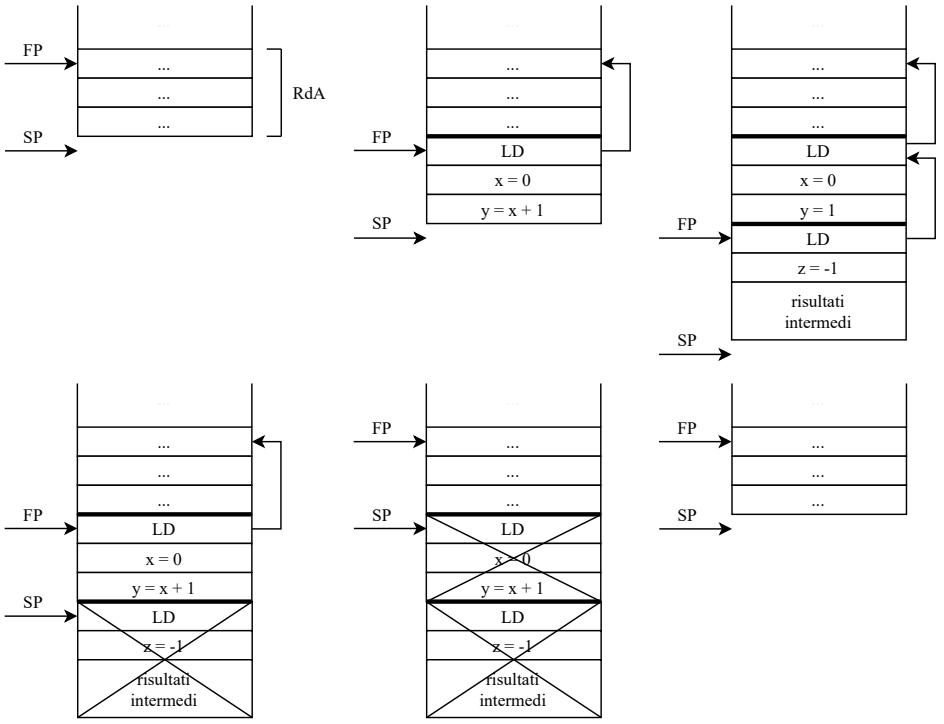


Figura 4.2: Evoluzione dello stack di attivazione

Passaggi eseguiti nello stack di attivazione per il precedente codice:

1. push del record con spazio per `x` e `y`
   (a) scrittura del link dinamico nello spazio di memoria apposito
   (b) link dinamico del nuovo RdA: `LD = FP`
   (c) `FP = SP`
   (d) `SP = SP + dimensione_nuovo_RdA`
2. assegnazione dei valori di `x` e `y`
3. push del record del blocco interno con spazio per `z` e per il risultato intermedio `x + y`
   (a) scrittura del link dinamico nello spazio di memoria apposito
   (b) link dinamico del nuovo RdA: `LD = FP`
   (c) `FP = SP`
   (d) `SP = SP + dimensione_nuovo_RdA`
4. assegnazione del valore di `z`
5. pop del record del blocco interno, elimina l'ultimo RdA, recupera spazio e ripristina i puntatori

   (a) `SP = FP`

   (b) `FP = link_dinamico` dell'RdA tolto dallo stack

6. pop del record del blocco esterno

   (a) `SP = FP`

   (b) `FP = link_dinamico` dell'RdA tolto dallo stack

Nella pratica, in molti linguaggi si preferisce evitare l'implementazione a pila per i blocchi anonimi perché magari per blocchi che devono essere eseguiti molte volte all'interno di altre procedure risulta dispendioso creare nuovi record nello stack di attivazione. Quello che si fa invece è, quando si crea il record per una procedura, si alloca anche lo spazio necessario per contenere i dati di tutti i blocchi anonimi che stanno all'interno della procedura. Questa tecnica potrebbe risultare più costosa in termini di memoria utilizzata, perché alcuni dei blocchi allocati potrebbero non essere usati o non essere usati contemporaneamente, ma sicuramente si ottiene un vantaggio in termini di velocità.

Esempio di ricorsione

```
1  int fact(int n){
2      if(n <= 1) return 1;
3      else return n * fact(n - 1);
4  }
```

Nell'RdA troviamo:

- parametri di input: `n`
- parametri di output: `fact(n)`
- link statici, dinamici e indirizzo di ritorno, back up registri del processore

ci saranno tanti RdA quanto il valore iniziale di `n`.

Nella seguente figura viene illustrato come lo stack si modifica durante l'esecuzione della ricorsione del codice esempio chiamata con `fact(3)`. n è il parametro, r sta per return, ra sta per return address, LD sta per link dinamico, pp sta per programma principale.

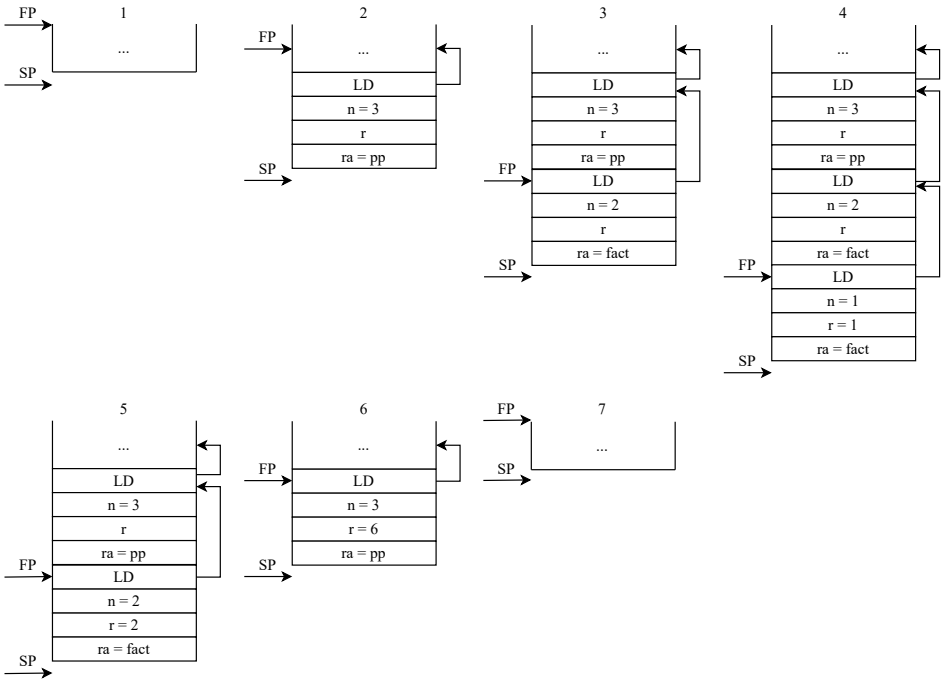


Figura 4.3: Utilizzo dello stack di attivazione durante la ricorsione chiamata con fact(3)

I passi che vengono compiuti sono:
Chiamata della procedura

1. allocazione dello spazio del RdA
2. inserire nel RdA appena creato i valori dei parametri passati
3. inizializzare le variabili locali
4. impostare link statici e dinamici
5. salvare i registri

Uscita dalla procedura

1. passare il risultato, tipicamente inserendolo in una locazione di memoria dove chi lo riceve sa dove trovarlo
2. cambiare FP e SP
3. ripristinare i registri
4. deallocare il RdA

Alcune di queste operazioni possono essere indifferentemente svolte dal chiamante o dal chiamato. Un'osservazione da fare è che inserire il codice nella procedura chiamata porta a generare meno codice, perché, nel caso una procedura venisse chiamata più volte, il codice non deve essere ripetuto per ogni chiamata.

4.2.3 Allocazione dinamica: heap

Come anticipato, la heap è una parte di memoria dove i blocchi possono essere allocati e deallocati a seconda delle necessità.

Necessaria quando il linguaggio permette:

- tipi di dato dinamici
- oggetti di dimensioni variabili (per esempio un vettore con dimensione variabile)
- oggetti che sopravvivono alla procedura che li ha creati

In questi casi non è possibile usare lo stack perché la quantità di spazio allocato nello stack non è variabile e una volta conclusa la funzione lo spazio viene deallocato.

I problemi della heap sono:

- gestione poco efficiente dello spazio: allocare spazio senza seguire un ordine provoca il fenomeno della frammentazione
- velocità di ricerca di spazio libero

Gestione della heap: suddivisione in blocchi di dimensione fissa

In questo tipo di heap i blocchi sono collegati tra loro come una lista libera, ovvero c'è un puntatore al primo blocco della lista e in ogni blocco c'è un puntatore al successivo. Quando c'è bisogno di memoria si possono dunque aggiungere blocchi alla lista e quando non ce n'è più bisogno si possono togliere. In questo modo la gestione della memoria è relativamente efficiente, ma il vincolo della dimensione fissa rende il tutto troppo rigido: non fornisce blocchi di elevata dimensione per strutture dati contigue di dimensione elevata. Dividere una struttura dati in più blocchi non è una strada praticabile perché rende la gestione troppo complessa in quanto richiedono una porzione sequenziale di memoria. Neanche l'opzione di usare blocchi grandi va bene perché porta a sprecare molto spazio.

Gestione della heap: suddivisione in blocchi di dimensione variabile

Inizialmente, quando si esegue un programma, la heap è formata da un unico blocco libero, poi sarà una lista formata da blocchi di dimensione

variabile. Quando serve dello spazio, si alloca all'interno di un blocco e quando non serve più, lo si aggiunge alla lista dei blocchi liberi.
Problemi:

- le operazioni devono essere efficienti (perché l'allocazione di dati dinamici avviene a run time)
- bisogna evitare i fenomeni di frammentazione della memoria

Ci sono due tipi di frammentazione:

- **frammentazione interna**: si verifica quando si alloca un blocco di dimensione strettamente maggiore di quella richiesta dal programma: la porzione di memoria non utilizzata, interna al blocco, evidentemente andrà sprecata fino a che il blocco non sarà restituito alla lista libera.
- **frammentazione esterna**: è ben peggiore della frammentazione interna, si verifica quando la lista libera è composta di blocchi di dimensione (relativamente) piccola per cui, anche se la somma della memoria libera totale presente è sufficiente, non si riesce ad usare effettivamente la memoria libera.

Esistono più tecniche di allocazione della memoria che cercano di evitare la frammentazione.

Unica lista libera Ad ogni richiesta di allocazione la ricerca di un blocco di dimensione opportuna può avvenire in due modi:

- first fit: si cerca nella lista il primo blocco di dimensione sufficiente
- best fit: si cerca il blocco di dimensione minima fra tutti quelli di dimensione sufficiente

La prima tecnica privilegia il tempo di gestione a scapito dell'occupazione di memoria, mentre la seconda l'occupazione di memoria a scapito della velocità. Per entrambe comunque il costo di allocazione è lineare rispetto al numero di blocchi presenti nella lista libera. Quando un blocco è deallocato, viene restituito alla lista libera. Un'ovvia ottimizzazione da fare consiste nel cercare nella lista libera un blocco contiguo a quello appena deallocato al fine di fonderli insieme in un unico blocco.

Liste libere multiple Per ridurre il costo di allocazione di un blocco alcuni metodi di gestione della heap usano più liste libere in cui ogni lista contiene blocchi liberi di dimensione simile. L'allocazione consiste nel scegliere il primo blocco disponibile nella lista opportuna.

Buddy system La memoria libera viene divisa in varie liste, ogni lista contiene blocchi della stessa dimensione, che è una potenza di k. La lista k-esima ha dunque blocchi di dimensione 2^k.

Se si desidera un blocco di dimensione 2^j:

- si cerca nella j-esima lista, se si trova si usa quello
- se invece la lista è vuota, si cerca un blocco, nella lista successiva (la j+1-esima), di dimensione doppia e lo si divide in due blocchi di dimensione 2^j, uno lo si usa, l'altro lo si collega alla lista j-esima
- se anche la lista successiva è vuota, la procedura si ripete ricorsivamente

Quando un blocco di 2^k viene deallocato, viene riunito alla sua altra metà (buddy), se disponibile. Ciò significa che se nella k-esima lista trovo un altro blocco, lo unisco a quello che sto deallocando e lo inserisco nella $k+1$-esima lista.

Fibonacci heap Variante molto simile alla buddy in cui, invece di usare potenze di 2, vengono usati i numeri di Fibonacci. Questo perché anche i numeri di Fibonacci possono essere divisi in due parti con dimensione un numero di Fibonacci, ma con il vantaggio che crescono più lentamente, ottenendo quindi una minore frammentazione interna.

4.2.4 Allocazione dinamica: regole di scope

Mentre si sta eseguendo una procedura, può essere necessario recuperare dati non locali nello stack di attivazione. Il modo per farlo dipende dalla regola di scope utilizzata, vediamo dunque come esse vengono implementate.

Scope statico

Oltre al link dinamico, in questo caso è necessario anche il link statico, che punta alla procedura genitore, ovvero quella dove la procedura in esecuzione è stata dichiarata.
Consideriamo il programma:

```
1   int x = 10;
2   void incx(){
3       x++;
4   }
5   void foo(){
6       int x = 0;
7       incx();
8   }
9   foo();
10  incx();
```

in cui `incx()` viene chiamata due volte: indirettamente in riga 9 tramite `foo()` e direttamente in riga 10. Non è possibile dunque utilizzare il link dinamico perché nel primo caso non punta alla dichiarazione di `x` che vogliamo,

bensì a quella della procedura `foo()`. Serve quindi un link apposito, il link statico, che punti alla dichiarazione corretta.

I legami validi sono quelli definiti:

- nella procedura corrente
- nella procedura genitore
- nel genitore del genitore

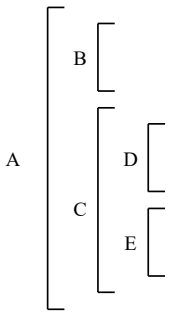


Figura 4.4: Struttura esempio

In D, i legami validi sono quelli definiti in D, C, A. Le altre dichiarazioni non sono visibili. Quindi quando la procedura D sarà in esecuzione, i record di attivazione che si cercheranno saranno quelli di C o quelli di A. Durante la ricerca si risalgono i link statici finché si trova la dichiarazione. Partendo da D dunque il primo link statico porta a C, se la dichiarazione non si trova in C si userà il link statico di C che porta ad A.

In pratica, a tempo di compilazione, per ogni variabile `x` in ogni procedura `P` si determina:

- quale procedura antenato `Q` contiene la dichiarazione di `x` a cui far riferimento
- di quanti livelli `Q` è antenato di `P`
- la posizione relativa di `x` nei record di attivazione di `Q`

A tempo di esecuzione per accedere ad `x` dalla procedura `P` bisogna:

- accedere all'ultimo RdA di `Q` attivo
- partendo dall'indirizzo base del RdA di `Q`, determinare la posizione di `x`

Per poter accedere ai RdA degli antenati e raggiungere `Q`, ogni RdA contiene un puntatore all'ultimo RdA del genitore, il link statico. I link statici definiscono la catena statica: la lista dei RdA degli antenati. Per

ogni variabile dunque è sufficiente risalire la catena di tanti link quanti sono stati determinati a tempo di compilazione, questo numero rimane costante per tutto il run time. Anche la posizione di `x` nel RdA è fissa dunque ci si può accedere con l'offset determinato a tempo di compilazione e non serve memorizzare il nome delle variabili negli RdA.

Riassumendo:

- link dinamico (procedura chiamante) dipende dalla sequenza di esecuzione del programma, definisce la catena dinamica
- link statico (procedura genitore) dipende dalla struttura delle dichiarazioni di procedure, definisce la catena statica

Data la struttura in Figura 4.5a, la sequenza di chiamate A, B, C, D, E, C, E genera la pila in Figura 4.5b:

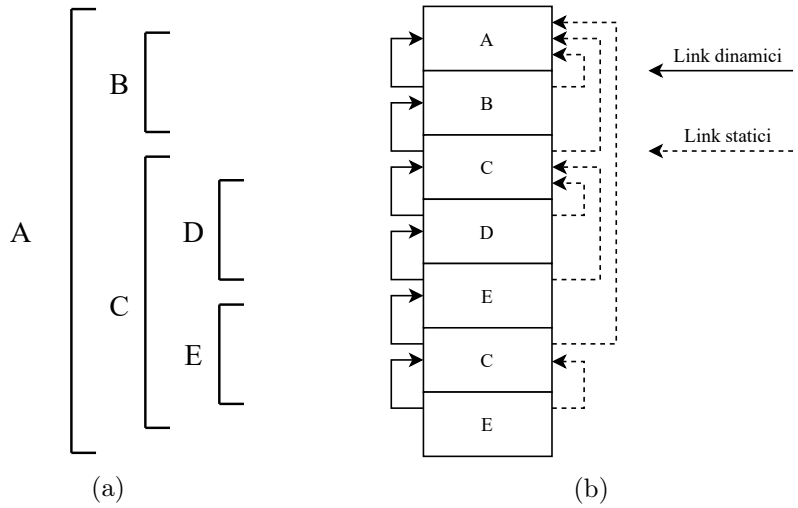


Figura 4.5: Struttura del programma e stack dopo l'esecuzione

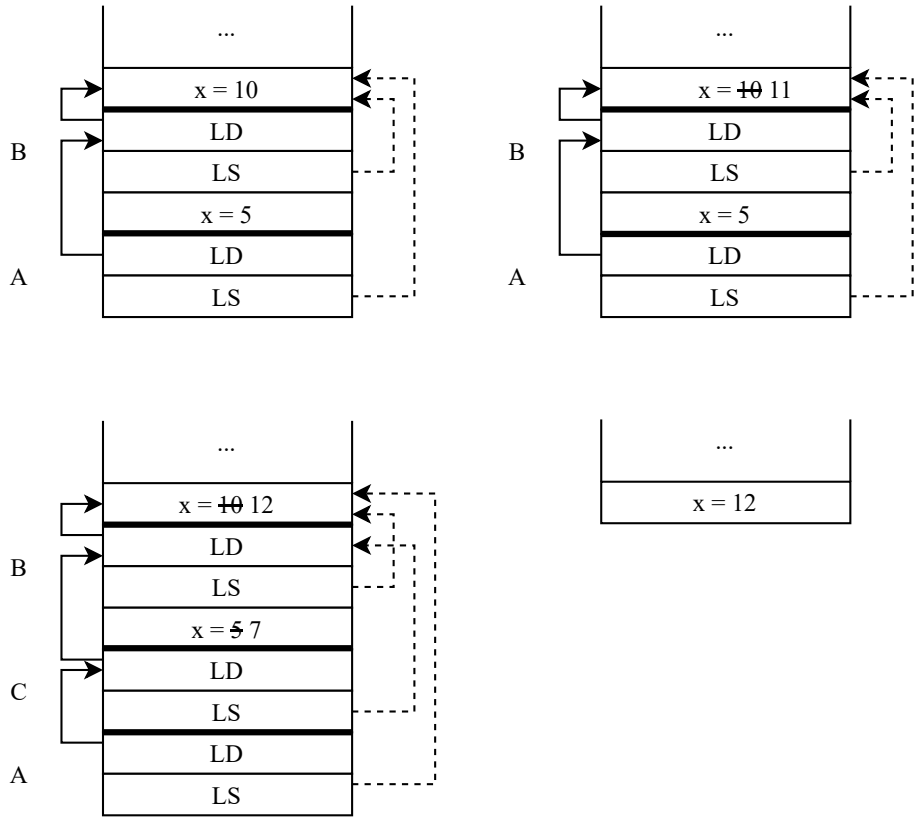
Notare come il link statico di E punta all'ultimo RdA di C.

Esempio di esecuzione

```
1  int x = 10;
2  void A(){
3      x = x + 1;
4  }
5  void B(){
6      int x = 5;
7      void C(){
8          x = x + 2;
```

```
9           A();
10      }
11      A();
12      C();
13  }
14  B();
```

per svolgere l'esercizio non è necessario scrivere tutto ciò che contengono i RdA, basta ciò che è strettamente necessario per la comprensione.



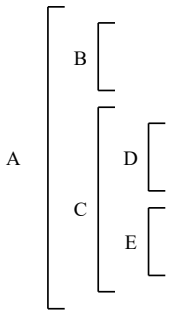
Come si determina il link statico? Prendiamo come esempio una procedura chiamante `Ch` che deve determinare il link statico della procedura chiamata `S` prima di passarglielo in modo che lo inserisca nel suo RdA.

Per prima cosa `Ch` vede dove sta nel suo ambiente la dichiarazione di `S`. Ci sono più possibilità:

- `S` è una procedura locale a `Ch`. In questo caso la profondità viene detta 0 perché è una procedura che si trova immediatamente. Il valore da inserire come link statico è il frame pointer attuale di `Ch`

- `S` è una procedura dichiarata in un n-esimo avo di `Ch`. In questo caso la profondità è n e il valore da inserire come link statico viene determinato risalendo la catena statica per n passi
- in altre posizioni `S` non sarebbe visibile

Esempio di profondità relativa k Nel caso di un programma con struttura



relativamente alla procedura C, all'interno di essa si possono chiamare tutte le altre procedure, e ciascuna ha un profondità relativa rispetto a C. D ed E hanno profondità 0, perché sono dichiarate all'interno di C, B e C hanno profondità 1 perché sono allo stesso livello di C e A ha profondità 2.

Ridurre i tempi per raggiungere la variabile non locale Normalmente, quando bisogna accedere ad una variabile che non è locale, il compilatore sa esattamente a quale dichiarazione deve fare riferimento, ovvero sa a che distanza (in termini di link statici da seguire) rispetto alla procedura corrente si trova il RdA della procedura in cui c'è la dichiarazione della variabile e sa accedere direttamente alla variabile perché conosce la sua posizione all'interno del RdA. L'ottimizzazione possibile è fare in modo di poter accedere direttamente al RdA giusto senza dover fare la discesa nella catena statica, evitando dunque un costo proporzionale alla profondità.

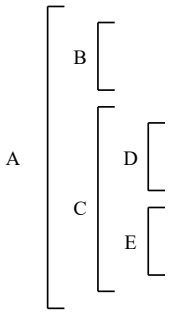
Display La prima tecnica che si può usare è il display, che sostanzialmente è un vettore che contiene dei puntatori a RdA attivi e visibili in un dato istante. L'i-esimo elemento dell'array contiene il puntatore al RdA della procedura a livello di annidamento statico i, dunque il primo elemento dell'array punterà al programma principale, le procedure definite all'interno avranno profondità 1 e così via. Questo risolve il problema perché non bisogna più scendere la catena statica, ma se la procedura corrente deve accedere ad una variabile `x` con annidamento statico i, basterà usare l'i-esimo puntatore nel display per determinarne il RdA che la contiene.

Come si aggiorna il display Il display deve essere aggiornato quando si chiama una procedura e quando eventualmente la si conclude. Supponiamo che la procedura `P` venga chiamata:

- `P` conosce la sua profondità statica, dunque sa dove va inserito il suo RdA nel display
- nel suo RdA, salva il puntatore che c'è attualmente nel display in quella posizione
- inserisce, sempre in quella posizione nel display, il puntatore al suo RdA
- al termine della chiamata, `P` ripristina il valore originario nel display

L'invariante che si vuole mantenere è che ad ogni istante il display contenga la lista di RdA corretti per l'ambiente della procedura.

Esempio di evoluzione dell'array del display con la sequenza di chiamate A C1 D C2 E con la solita struttura:



All'inizio l'array contiene solo 0, che indicano puntatori vuoti.

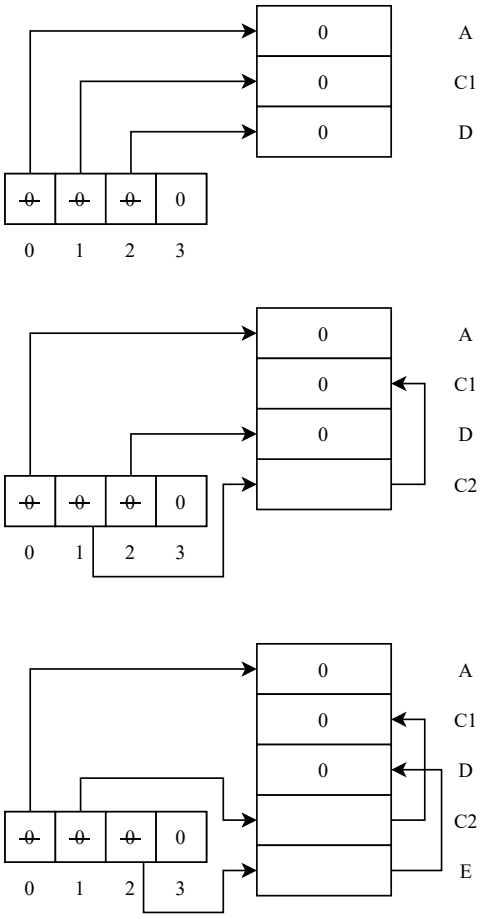


Figura 4.6: Esempio dell'evoluzione del display

Nel primo passo gli zeri vengono salvati nei RdA delle procedure e sostituiti con puntatori a questi RdA. A è a profondità statica 0 quindi sostituisce l'elemento in posizione 0, C1 è a profondità statica 1 quindi sostituisce l'elemento in posizione 1 e D è a profondità statica 2 quindi sostituisce l'elemento in posizione 2.

Nel secondo passo abbiamo un'altra chiamata a C che, di nuovo, si trova a profondità statica 1. Quello che viene fatto è dunque salvare in C2 il puntatore che c'era in posizione 1 e sostituirlo con un puntatore al suo RdA.

Quello che succede nel terzo passo è che viene chiamata E che si trova a profondità statica 2, dunque viene salvato l'elemento in posizione 2, ovvero il puntatore a D, e sostituto con un puntatore al suo RdA.

Infine, nel momento in cui si concludono le procedure, vengono eseguiti tutti i passi al contrario fino a ritornare alla situazione iniziale.

Nella pratica la tecnica display è poco utilizzata perché le catene statiche con lunghezza > 3 sono rare. Di fatto però non è molto più costosa rispetto alle

sole catene statiche, l'unico svantaggio è che bisogna eseguire delle operazioni anche all'uscita dalle procedure.

Bisogna porre attenzione al fatto che se ci sono funzioni che vengono passate come parametro non basta aggiornare il livello corrente del display, ma bisogna aggiornare anche quelli prima, altrimenti i link statici da seguire non sono corretti; questo complica abbastanza il meccanismo. Negli esercizi, se verrà chiesto il display, non ci saranno funzioni passate come parametro.

Scope dinamico

Nello scope dinamico, i link non sono più determinati a tempo di compilazione, ma dipendono dal flusso del controllo a run time e dall'ordine con cui i sottoprogrammi sono chiamati. La regola generale è che il legame valido per il nome `n` è determinato dall'ultima dichiarazione del nome `n` eseguita, quindi per nomi non locali è sufficiente scendere la catena dinamica.

L'implementazione è più semplice (anche se un po' più costosa) rispetto allo scope statico, motivo per cui alcuni linguaggi più vecchi usavano questa tecnica. L'implementazione consiste nel memorizzare i nomi delle variabili negli RdA (a differenza dello scope statico dove non è necessario memorizzarli). La ricerca viene eseguita scendendo livello dopo livello nello stack di attivazione seguendo la catena dinamica e si conclude quando la dichiarazione viene trovata.

I due principali svantaggi sono dunque:

- bisogno di memorizzare il nome oltre che il valore delle variabili
- bisogna scendere nella catena dinamica senza sapere quanto

Esempio: struttura del programma e chiamate A B C D

```
1  A(){
2      x = ...
3      y = ...
4      B(){
5          x = ...
6          v = ...
7      }
8      C(){
9          w = ...
10         x = ...
11         D(){
12             w = ...
13         }
14     }
15 }
```

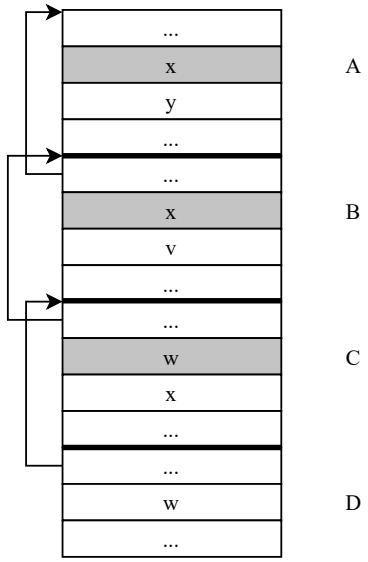


Figura 4.7: Stack di attivazione

Lo stack di attivazione contiene le dichiarazioni delle variabili dentro ogni RdA delle procedure. Quando siamo in D, possiamo accedere a tutte le variabili dichiarate in precedenza seguendo la catena dinamica, tranne quelle che sono sovrascritte, denotate con il colore scuro.

A-List È possibile usare un'altra tecnica di implementazione che usa le A-List, in cui le associazioni sono raggruppate in una struttura dati apposita, invece di usare lo stack di attivazione.

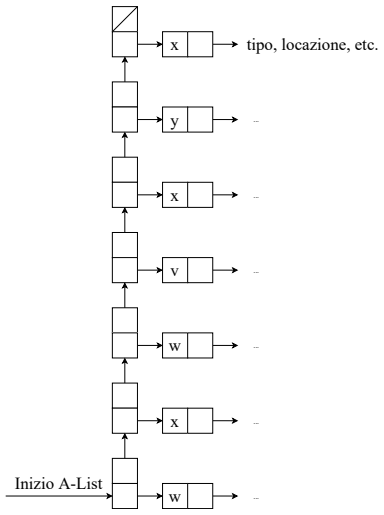


Figura 4.8: A-List

Ogni volta che si cerca una variabile si parte dall'inizio e si scandisce la lista fino a trovarla. È di fatto molto simile all'implementazione normale, cambia solo che le informazioni sulle variabili sono separate dallo stack di attivazione.

Tuttavia, entrambe le soluzioni precedenti impiegano un tempo lineare per la ricerca delle variabili.

Tabella centrale dei riferimenti – CRT Tecnica per ottenere un accesso in tempo costante alla posizione delle variabili. All'inizio, il compilatore vede quali sono tutte le variabili dichiarate e costruisce una tabella che contiene tutti i nomi. (Esiste un'altra versione delle CRT, in cui alcuni nomi potrebbero essere inseriti a run time e per cui sono necessarie delle tabelle hash). Ad ogni nome viene associata una lista, dove il primo elemento, se esiste, è il riferimento corrente di quella variabile, mentre gli altri elementi sono le dichiarazioni della variabile che sono nascoste. Quando si esce da una procedura bisogna aggiornare la lista relativa alle variabili definite all'interno di essa togliendo il primo elemento della lista.

Esempio:

```
1  A(){
2      x = ...
3      y = ...
4      B(){
5          x = ...
6          v = ...
7      }
8      C(){
9          w = ...
10         x = ...
11         D(){
12             w = ...
13         }
14     }
15 }
```

Il pedice indica la procedura dentro la quale la variabile è stata definita.

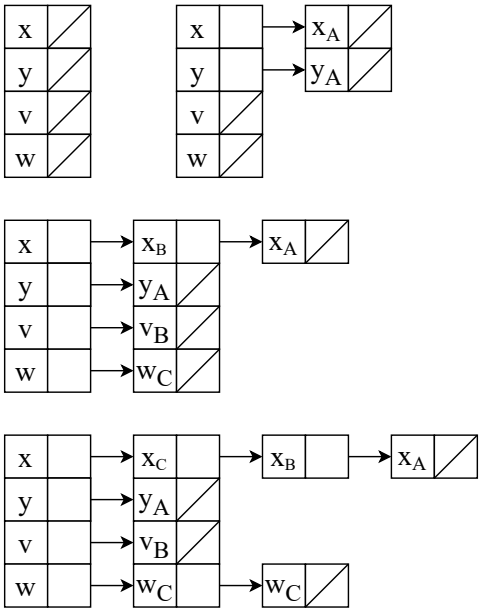


Figura 4.9: Evoluzione della tabella centrale dei riferimenti

Lo svantaggio è che ogni volta che si entra e si esce da una procedura bisogna aggiornare la tabella, ma ha il grosso vantaggio che poi l'accesso alle variabili viene fatto in tempo costante.

Stack dei riferimenti nascosti Tecnica in cui si costruisce una tabella in cui ogni riga è occupata da una variabile. A differenza delle CRT non si usa una lista per tenere in memoria il valore precedente, bensì si usa uno stack. Ogni volta che si entra in una procedura si inserisce il valore vecchio nello stack e ogni volta che si esce si ripristina il valore precedente.
Esempio

```
1   A(){
2       x = ...
3       y = ...
4       B(){
5           x = ...
6           v = ...
7       }
8       C(){
9           w = ...
10          x = ...
11          D(){
12              w = ...
13          }
14      }
```

```
15  }
```

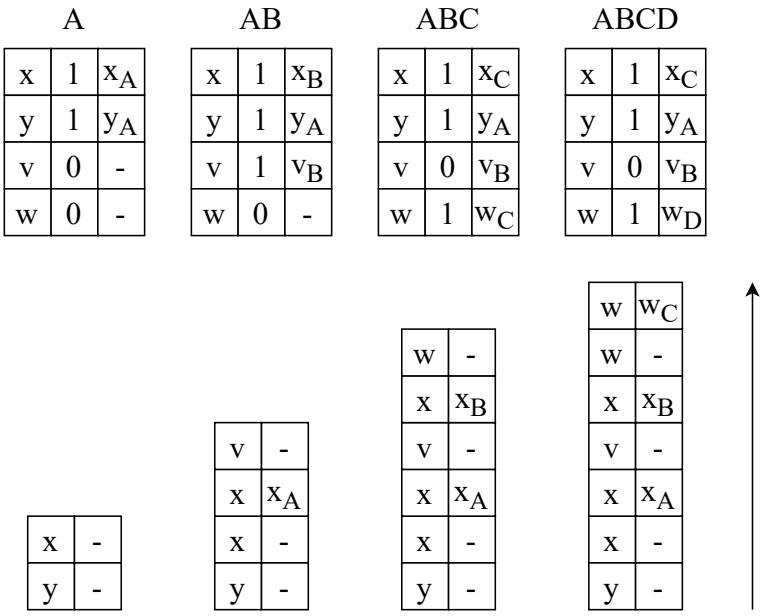


Figura 4.10: Evoluzione dello stack dei riferimenti nascosti

La colonna centrale è un booleano che indica se la dichiarazione è visibile nel punto corrente oppure no. La colonna più a destra indica la procedura alla quale fare riferimento per la variabile attiva in un dato momento. Nello stack dell'esempio i valori vengono inseriti dall'alto.

Capitolo 5

Strutture di controllo

In questo capitolo ci occupiamo di stabilire quali sono i meccanismi attraverso cui, nei linguaggi di programmazione, si determinano le operazioni da eseguire.

Nel codice macchina abbiamo una sequenza di istruzioni elementari e in più, per aggiungere un po' di complessità, delle istruzioni salto, che permettono di simulare i meccanismi che si trovano ad alto livello. Il problema di queste istruzioni è che sono poco espressive, dunque nei linguaggi di programmazione ad alto livello si preferisce astrarre sul controllo, ottenendo delle istruzioni più strutturate, compatte e leggibili.

Le strutture per il controllo del flusso sono:

- espressioni
- comandi
- sequenzializzazione di comandi
- test
- iterazione
- ricorsione

Esistono inoltre altri meccanismi di controllo più sofisticati che permettono di fare cose più ad alto livello:

- chiamate di funzione
- gestione delle eccezioni
- esecuzione concorrente
- scelta non deterministica

che vedremo più avanti.

I paradigmi di programmazione (imperativo, funzionale) differiscono principalmente nei meccanismi di controllo adottati:

- imperativo: assegnazione, sequenzializzazione, iterazione
- dichiarativo: valutazione di espressioni, ricorsione

5.1 Espressioni

Le espressioni sono costruite a partire da identificatori (variabili, costanti), letterali (numeri), operatori (`+`, `-`, ...) e funzioni. Tipicamente, la valutazione di un'espressione restituisce un valore, ma può anche avere effetti collaterali (modifiche di locazioni di memoria) e possono divergere (non terminare mai e il controllo non viene restituito).

5.1.1 Notazione

C'è una certa varietà nelle notazioni che vengono usate nei linguaggi di programmazione, le principali differenze sono:

- posizione dell'operatore rispetto agli argomenti: infissa, prefissa, postfissa
- uso delle parentesi

Tipicamente si usa la notazione infissa, dove l'operatore si inserisce tra gli argomenti, in quanto più simile alla notazione aritmetica a cui siamo abituati.

Diverse notazioni possibili	Esempi
infissa	`a + b * c`
funzione matematica	`add(a, mult(b, c))`
linguaggi funzionali (Cambridge polish)	`(+ a (* b c)`
omissione di alcune parentesi (ML, Haskell)	`+ a (* b c)`

In scheme si usa la notazione Cambridge polish e l'uso delle parentesi è forzato in quanto necessario per la valutazione, in altri invece, come Haskell, l'uso delle parentesi è libero perché servono solo per definire un ordine di valutazione.

Nei linguaggi di programmazione viene spesso usato dello "zucchero sintattico" per ottenere delle scritture alternative di un'espressione per migliorare la leggibilità.

- Haskell, Ada: `a + b` al posto di `'+' a b` e `'+' (a, b)`
- Ruby, C++: `a + b` al posto di `a.+ b` e `a.operator+ (b)`

Notazione polacca

La notazione polacca ha due forme:

- prefissa (polacca diretta): `+ a * b c`
- postfissa (polacca inversa): `a b c * +`

In entrambi i casi viene eseguita la somma tra `a` e la moltiplicazione tra `b` e `c`.

La caratteristica più interessante della notazione polacca è che non necessita di parentesi se l'arietà delle funzioni (ovvero il numero di argomenti) è prefissata. Un esempio di linguaggio in cui ci sono operatori con arietà non fissa è Scheme, infatti si può avere un operatore seguito da un lista di argomenti di lunghezza arbitraria (es: `(+ 1 2 3)`).

La notazione polacca inversa è utilizzata all'interno di alcuni linguaggi intermedi come il java bytecode. Questo perché questi linguaggi intermedi non si basano su registri come ARM, bensì su uno stack degli operandi, ovvero uno stack in cui si depositano valori e su cui si chiamano le operazioni aritmetiche. Se per esempio viene chiamata la funzione di somma, essa viene eseguita sui primi due valori dello stack, togliendoli e sostituendoli con il risultato.
Esempio: `2 + 3 * 5` diventa

```
2 3 5 * +

push 2;
push 3;
push 5;
mult;
add;
```

Ogni volta che si trova una costante si fa il push nello stack e ogni volta che si trova un operatore si chiama la funzione sullo stack.

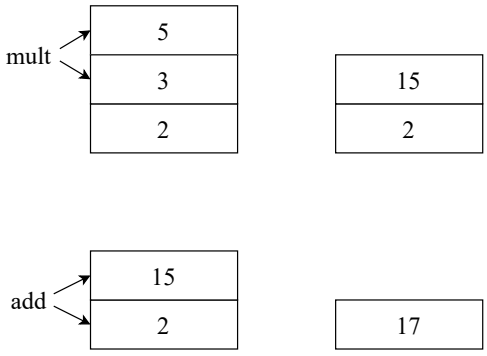


Figura 5.1: Evoluzione dello stack degli operandi durante la valutazione dell'espressione

Notazione infissa

I linguaggi di programmazione tendono a usare la notazione infissa in quanto più simile alla scrittura matematica, tuttavia è ambigua e non sempre il risultato della valutazione di un'espressione è ovvio perché in base al linguaggio possono cambiare le regole di precedenza degli operatori. Per esempio, nella seguente espressione

```
a + b * c ** d ** e / f
```

l'operatore `**` indica l'elevamento a potenza che non è un'operazione associativa, dunque ogni linguaggio può decidere se associarla a sinistra o a destra. Si può dunque interpretarla come $(c^d)^e$ oppure c^{d^e}, ma normalmente si adotta la seconda interpretazione, associando a destra.

Ogni linguaggio fissa le sue regole di precedenza tra operatori, ma normalmente gli operatori aritmetici hanno precedenza su quelli di confronto, e quelli di confronto hanno precedenza su quelli logici. Su queste regole ci sono vari livelli di complessità, per esempio il C e i suoi derivati (C++, Java, C#) hanno 15 livelli di precedenza, il Pascal ne ha 3, in APL e SmallTalk tutti gli operatori hanno la stessa precedenza e quindi le parentesi sono obbligatorie e in Haskell si possono definire nuove funzioni assegnandogli come identificatore un simbolo di operazione che può essere utilizzato con una notazione infissa e si può specificarne precedenza e associatività.

Per quanto riguarda invece l'ordine con cui eseguire operazioni con operatori dello stesso livello di priorità, normalmente si associa a sinistra: `15 + 4 - 3 = (15 + 4) - 3`

5.1.2 Rappresentazione ad albero

Data un'espressione, il modo migliore e più naturale di rappresentarne il significato è un l'albero, esse infatti vengono linearizzate per necessità di scrittura. Per esempio l'espressione

```
(a + f(b)) * (c + f(b))
```

genera il seguente albero

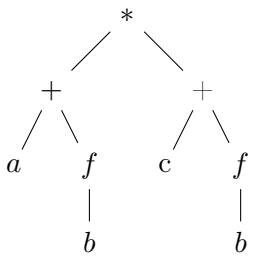


Figura 5.2: Albero generato dall'espressione

Il problema di questa rappresentazione è che non dice con che ordine eseguire le operazioni, per esempio se eseguire prima la somma di sinistra o di destra. Nel caso in cui non ci siano effetti collaterali, l'ordine di esecuzione non fa alcuna differenza, se non di ottimizzazione, ma nel caso ci siano, la valutazione potrebbe portare risultati diversi in base all'ordine applicato.

5.1.3 Effetti collaterali ed ordine di valutazione

Diciamo che abbiamo un effetto collaterale se la valutazione di un'espressione porta ad una modifica dello stato del programma, con lo stato inteso come l'insieme input, output e memoria nel momento in cui l'espressione viene valutata. Un classico esempio di tale fenomeno è quando la valutazione di un'espressione porta alla chiamata di funzioni e tali funzioni modificano la memoria, come nel seguente esempio:

```
(a + f(b)) * (c + f(d))
```

In questa espressione una valutazione da destra a sinistra può differire dalla valutazione della stessa espressione ma da sinistra a destra, di seguito riportiamo un codice C che mostra questo evento.

```
1  int a = 3; //parametri usati come input
2  int b = 5;
3  int c = 7;
4  int d = 9;
5
6  int brokenFunction(int); //dichiarazione funzione
7
8  int param; //parametro di stato
9
10 int main()
11 {
12     int x;
13     param = 1;
14
15     x = (a + brokenFunction(b)) * (c + brokenFunction(d));
           // (a + f(b)) * (c + f(d))
16
17     printf("%d", first);
18     return 0;
19 }
20
21 int brokenFunction(int number){ //
22     param++;
23     return number*param;
```

```
24  }
```

Eseguendo il codice si ottiene che l'output è 442, vogliamo vedere però se c'è una differenza se la valutazione viene fatta da destra a sinistra e vice versa, allora possiamo forzare una o l'altra sostituendo al posto della riga 15 i seguenti frammenti di codice.

```
1   x = (a + brokenFunction(b)); //valutazione da sinistra a
        destra
2   x = x * (c + brokenFunction(d));
3
4   x = (c + brokenFunction(d)); //valutazione da destra a
        sinistra
5   x = x * (a + brokenFunction(b));
```

L'output da sinistra a destra risulta essere sempre 442, mentre per la valutazione da destra a sinistra risulta essere 450.

Chiamiamo **ordine di valutazione** l'ordine con cui le sotto-espressioni di un'espressione vengono valutate.

Alcuni linguaggi definiscono quest'ordine chiaramente, ne è un esempio Java (da sinistra a destra), mentre altri linguaggi, tra cui il c, non lo esprimono formalmente, lasciando la scelta al compilatore, il quale può portare a risultati diversi come per il codice seguente:

```
1   int x=1;
2   printf("%d \n", (x++) * (++x)); //ritorna 3
3   x=1;
4   printf("%d \n", (++x) * (x++)); //ritorna 4
```

La formalizzazione dell'ordine di valutazione ha alcuni vantaggi, tra cui una maggiore chiarezza nel codice, ma porta anche degli svantaggi tra cui la minore efficienza.

5.1.4 Ottimizzazione e ordine di valutazione

Alcuni compilatori possono modificare l'ordine di valutazione per ragioni di efficienza, per esempio:

```
1   a = b + c
2   d = c + e + b
```

può essere riarrangiato in

```
1   a = b + c
2   d = b + c + e
```

ed eseguito come

```
1   a = b + c
2   d = a + e
```

In alcuni casi, modifiche come queste possono portare ad alterazioni del risultato finale. Per evitare le ambiguità dovute all'ordine di valutazione, in alcuni linguaggi non sono ammesse funzioni con effetti collaterali nelle espressioni (Haskell). Altri linguaggi invece specificano l'ordine di valutazione (Java). Infine, in altri linguaggi ancora, dove non ci sono restrizioni, si può evitare che il risultato dipenda da scelte del compilatore forzando un ordine di valutazione, come nel seguente esempio: l'espressione

```
    y = (a + f(b)) * (c + f(d))
```

può essere spezzata riscrivendola come

```
1   x = a + f(b);
2   y = x * (c + f(d))
```

Per concludere, gli effetti collaterali possono essere utili in caso di:

- gestione di strutture dati di grandi dimensioni, funzioni che operano su matrici
- definizione di funzioni che generano numeri casuali, `rand()`

ma la loro assenza permette alla valutazione del codice di essere:

- indipendente dall'ordine di valutazione
- più vicina all'intuizione matematica
- spesso più facile da capire
- più facile verificare, provare la correttezza
- più facile da ottimizzare per il compilatore (preservando il significato originale)

5.1.5 Aritmetica finita

In un calcolatore l'insieme dei numeri rappresentabili è finito:

- numeri interi: insieme limitato
- numeri float: insieme limitato e precisione limitata

Questo porta alcune conseguenze importanti tra cui la possibilità di overflow, errori di arrotondamento ed alcune proprietà matematiche perdono la valenza, per esempio:

$$(a + (b + c)) \text{ può essere diverso da } ((a + b) + c)$$

Una delle due può perfino produrre un overflow quando l'altra non lo fa, per esempio se prendiamo come parametri $a = -2$, $b = max\_int$ e $c = 2$ allora l'output della prima genera un errore dovuto alla somma tra max_int e 2, la seconda invece non lo fa e ritorna max_int.

Se prendiamo $a = 10^{15}$, $b = -10^{15}$ e $c = 10^{-15}$ la valutazione differisce a causa dell'errore di arrotondamento.

5.1.6 Valutazione eager e lazy

Per valutazione **eager**, si intende che gli argomenti di un'espressione o di una funzione vengono valutati prima dell'inizio della valutazione della stessa, viceversa per **lazy** gli argomenti vengono valutati solo se necessario. Nelle operazioni aritmetiche in genere si usa sempre la eager dato che se ho degli argomenti prima o poi li uso, diversa è la situazione per le espressioni booleane.

Nelle espressioni IF THEN ELSE (diverse dal comando IF THEN ELSE, nelle espressioni le clausole non contengono comandi, ma altre espressioni da valutare e ritornare come risultato) infatti la valutazione è di solito lazy: prima viene valutata l'espressione, di solito booleana, all'interno della guardia, e poi, in base al risultato, solo una delle due clausole.

Queste espressioni sono presenti in molti linguaggi di programmazione.

C, Java	`a == 0 ? b : b/a`
Scheme	`(if (= a 0) b (/ b a))`
Python	`b if a == 0 else b/a`

Notare come la valutazione lazy è fondamentale nell'esempio mostrato, nel caso in cui `a = 0`, se entrambe le espressioni venissero valutate si otterrebbe un errore!

Un altro caso in cui è ammessa una valutazione lazy è con alcuni operatori booleani, in particolare se si verifica un fenomeno chiamato **valutazione corto circuito**, ovvero quando la valutazione di un primo argomento permette di determinare con certezza il risultato dell'intera espressione.

In questi casi l'ordine di valutazione risulta fondamentale per determinare il risultato.

Prendiamo per esempio la seguente espressione booleana:

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

se valutiamo il primo argomento ed otteniamo `true` non ci serve valutare anche il secondo siccome il risultato di `true || qualsiasi = true`, inoltre valutare anche il secondo argomento porterebbe ad un errore dato che se il primo è true allora nel secondo dovremmo valutare una divisione per 0, motivo per cui una valutazione eager non sarebbe possibile. Nell'esempio abbiamo dato per scontato che la valutazione fosse da sinistra a destra, ma se così non fosse l'espressione porterebbe comunque ad un errore nonostante la valutazione lazy, dato che dovremmo valutare prima la divisione, eventualmente con `a = 0`, ed in seguito valutare l'argomento successivo. In genere valgono le seguenti regole:

- se il primo argomento di un or `||` è `true`, restituisce `true`
- se il primo argomento di un and `&&` è `false`, restituisce `false`

Alcuni linguaggi ADA mettono a disposizione due diverse versioni degli operatori booleani:

- short circuit: and then or else
- eager: and or, utili se la valutazione delle espressioni ha un effetto collaterale necessario alla computazione

Mostriamo un esempio in cui lo stesso codice si comporta in maniera diversa a seconda del linguaggio.

```
1  p = lista
2  while (p && p->valore != 3)
3  p = p->next
```

Il codice è scritto in C, la valutazione nella guardia del while avviene correttamente dato che il secondo argomento, `p->valore != 3`, viene valutato solo quando il primo non è false (ricordiamo che in C il false è rappresentato come 0 e il puntatore a vuoto è 0).

```
1  p := lista;
2  while (p <> nil ) and (p^.valore <> 3) do
3      p := p^.prossimo;
```

Nel caso del pascal viene generato un errore a causa della valutazione eager che porta a valutare anche il secondo argomento quando la lista è nulla.

5.2 Comandi

Se, come dicevamo prima, le espressioni sono presenti in tutti i linguaggi di programmazione, non è così per i comandi, che sono costrutti presenti tipicamente nei linguaggi cosiddetti imperativi.

Un comando è una entità sintattica la cui valutazione non necessariamente restituisce un valore, ma può avere un effetto collaterale. Il comando più comune è l'assegnamento, che inserisce in una locazione di memoria un valore ottenuto valutando un'espressione.
Esempio:

```
1  x = 2;
2  y = x + 1;
```

notare il diverso ruolo svolto da `x` nei due assegnamenti:

- se si trova a sinistra dell'uguale, `x` denota una locazione di memoria e viene detto **l-value**
- se si trova a destra dell'uguale, `x` denota il contenuto della locazione e viene detto **r-value**

In generale,

```
exp1 := exp2
```

- si valuta `exp1` per ottenere un l-value (locazione)
- si valuta `exp2` per ottenere un r-value (valore memorizzabile)
- si inserisce il valore nella locazione

Un l-value può essere definito da un'espressione complessa, esempio in C:

```
(f(a) + 3)->b[c] = 2
```

dove

- `f(a)` è un puntatore ad un elemento in un array di puntatori a strutture `A`
- la struttura `A` ha un campo `b` che è un array
- viene inserito `2` nel campo c-esimo dell'array

La parte sinistra di un'assegnazione è detta tipicamente variabile. Tralasciando l'uso matematico del termine "variabile", in informatica, a seconda dei contesti, esistono più significati:

- nei linguaggi imperativi: identificatore a cui è associata una locazione di memoria dove si trova il valore modificabile
- nei linguaggi funzionali: un identificatore a cui è associato un valore non modificabile
- nei linguaggi logici: alle variabili non viene associato alcun valore, per esempio nel predicato $x > 5$, x non ha alcun valore e la computazione cerca la corretta istanziazione, ovvero quella che rende vera il predicato

5.2.1 Modello a valore

È il modello più semplice, l'idea è che ad una variabile viene associata una locazione di memoria, e dentro quella locazione si inserisce il valore.

5.2.2 Modello a riferimento

Nel modello a riferimento, alla variabile viene associata una locazione di memoria, ma, in questo caso, dentro quella locazione non si trova direttamente il dato, ma un riferimento (un puntatore) ad un'altra locazione di memoria in cui si trova il valore. La differenza più evidente è nelle operazioni di assegnamento, per esempio, se `Y` contiene il riferimento ad un valore, dopo l'assegnamento `X = Y`, `X` e `Y` fanno riferimento alla stessa locazione di memoria, contenente quindi un valore condiviso tra le due variabili.

5.2.3 Differenze di implementazione tra i due modelli

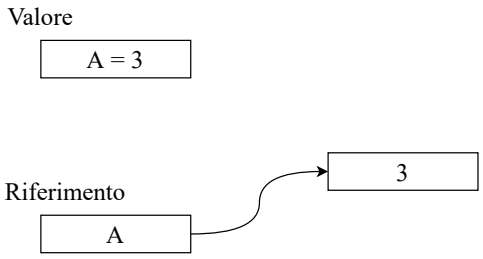


Figura 5.3: Modello a valore e a riferimento

È importante notare che dopo un'assegnazione per esempio di una lista, con il modello a valore le due variabili sono distinte e le modifiche sono indipendenti, mentre con il modello a riferimento le modifiche sono condivise. Questo però non è sempre vero, perché, per oggetti più semplici di una lista verrebbero comunque assegnati due oggetti distinti, per quali dipende dal linguaggio di programmazione.

In alcuni linguaggi, vengono usati entrambi i modelli a seconda del tipo della variabile. Per esempio in Java i tipi primitivi (interi, booleani, ...) usano il modello a variabile e l'assegnamento copia un valore nella memoria, mentre i tipi riferimento (tipi classe, array) usano il modello a riferimento e l'assegnamento crea una condivisione dell'oggetto.

La stessa cosa accade in Python, in particolare divide i tipi in due categorie:

- immutabili, tipi semplici: interi, booleani, enuple
- mutabili, tipi complessi: vettori, liste, insiemi

e l'assegnamento si comporta di conseguenza:

- per oggetti immutabili: viene creata una nuova istanza dell'oggetto, non si modifica la memoria
- per oggetti mutabili: viene condivisa la memoria occupata dalla stessa istanza, eventualmente modificata

Esempio in Python

```
1  tuple1 = (1,2,3) # le tuple sono immutabili
2  list1 = [1,2,3] # le liste sono mutabili
3
4  tuple2 = tuple1
5  list2 = list1
6
7  # operazione che aggiunge elementi in coda
8  tuple2 += (9,9,9)
9  list2 += [9,9,9]
10
11 print 'tuple1 = ', tuple1 #output: (1,2,3)
12 print 'tuple2 = ', tuple2 #output: (1,2,3,9,9,9)
13 print 'list1 = ', list1  #output: [1,2,3,9,9,9]
14 print 'list2 = ', list2  #output: [1,2,3,9,9,9]
```

con la tupla, si crea una nuova istanza che viene aggiornata, mentre l'istanza originale rimane immutata. Con la lista invece anche la lista originale viene modificata.

Vantaggi del modello a riferimento:

- non si duplicano strutture dati complesse, minor costo
- tutte le variabili sono puntatori, utile nelle funzioni polimorfe, ovvero funzioni che lavorano su tipi di dato diversi senza dover creare una funzione per ogni tipo

Svantaggi:

- si crea aliasing, le modifiche si ripercuotono anche su altre variabili, rende più complessa la comprensione e l'uso di un programma
- costa di più l'accesso al dato, in quanto bisogna fare un passaggio aggiuntivo

I linguaggi funzionali si dividono in puri, dove le variabili sono sempre immutabili, e quelli non puri, dove le variabili possono anche essere mutabili.

Esempio linguaggio funzionale non puro (ML)

In ML si distingue tra locazione e contenuto. Quando si definisce una variabile, la si dichiara una volta sola e il suo valore è fisso per sempre. Se si volesse implementare la mutabilità si può definire una variabile non come un valore fisso, ma come un riferimento ad una locazione di memoria che eventualmente si può modificare.

```
1  val x = ref 2
2  val x2 = x
3  val x3 = !x
4  val _ = x := (!x) + 7
```

dove:

- a riga 1, `x` denota una locazione di memoria contenente il valore `2`
- a riga 2, `x2` punterà alla stessa locazione di memoria di `x`
- a riga 3, `x3` prende il valore `2` e non potrà essere modificato
- a riga 4, nella locazione puntata da `x` viene messo il valore di `x + 7`, quindi il contenuto di `x` e `x2` è ora `9`, mentre `x3` ha sempre valore `2`. L'underscore serve solo per forzare la valutazione dell'espressione con effetto collaterale assegnando il risultato ad una variabile che non esiste e che verrà poi buttata via

5.2.4 Valori denotabili, memorizzabili, esprimibili

Nei linguaggi imperativi distinguiamo tra:

- ambiente: nomi $\rightarrow$ valori denotabili, normalmente definito tramite le dichiarazioni
- memoria: locazioni $\rightarrow$ valori memorizzabili

distinguiamo inoltre tre classi di valori:

- valori **denotabili**: cosa può essere associato ad un identificatore (interi, locazioni di memoria, procedure)
- valori **memorizzabili**: tutti i dati che si possono inserire nella memoria tramite assegnamenti
- valori **esprimibili**: risultato della valutazione di un'espressione

I linguaggi di programmazione devono, esplicitamente o implicitamente, definire per ogni classe di valori quali possono essere.

Esempio:

- le procedure sono solitamente denotabili con un identificatore, esprimibili nei linguaggi funzionali tramite un'espressione, memorizzabili se si può memorizzare una procedura in una locazione di memoria
- le locazioni sono denotabili nei linguaggi imperativi, esprimibili con i puntatori, memorizzabili salvando in una locazione di memoria un puntatore

Nei linguaggi funzionali puri non esistono valori memorizzabili e le locazioni non sono denotabili o esprimibili. Nei linguaggi funzionali non puri le funzioni sono valori esprimibili.

5.2.5 Operazioni di assegnamento

In molti linguaggi sono definiti degli operatori di assegnamento più sintetici:

```
x = x + 1 diventa x += 1 (C, Java, ...)
x := x + 1 diventa x +:= 1 (Algol, Pascal, ...)
```

questo perché un'operazione standard di incremento come per esempio

```
A[index(i)] = A[index(i)] + 1
```

dove `index(i)` è una funzione che calcola un indice, bisognerebbe chiamare due volte questa funzione per valutare l'espressione e nel caso in cui la funzione causasse effetti collaterali, questi verrebbero ripetuti, inoltre, con la sintassi sintetica si otterrebbe una migliore leggibilità.

In C, Java e altri linguaggi ci sono molti operatori di assegnamento, incremento, decremento:

```
+=  -=  *=  /=  %=  &=  |=
```

che sono rispettivamente somma, sottrazione, moltiplicazione, divisione, resto, bit–wise and, bit–wise or. Di incremento e decremento di un'unità:

```
x++ x--
```

nel caso in cui la variabile incrementata sia un puntatore C ad un array di oggetti, l'incremento viene moltiplicato per la dimensione degli oggetti puntati.

5.3 Espressioni e comandi

Riassumendo, ricordiamo che sintatticamente si distingue tra comandi ed espressioni:

- comandi: l'effetto collaterale è importante e non restituisce valori

- espressioni: il valore restituito è importante e non ha effetti collaterali

in alcuni linguaggi però la distinzione tra comando ed espressione risulta sfumata: i due aspetti, effetto collaterale e risultato coesistono e dove è previsto un comando si possono inserire espressioni e viceversa.

Normalmente, i comandi sono separati dalle espressioni, nel senso che non sono completamente intercambiabili, per esempio in C, Java e C# ci sono due tipi di test if then else:

```
if (a == b) {x = 1} else {x = 0};
x = (a == b) ? 1 : 0;
```

il primo si basa su comandi, il secondo è un'espressione, ma si possono inserire espressioni dove ci si aspetta un comando o una sequenza di comandi, per esempio tra le parentesi graffe del primo test. Se lì mettessimo un'espressione, avremmo che essa verrà valutata e il risultato verrà buttato via.

Un'altra cosa che si ha in C è che i comandi restituiscono anche dei valori, per esempio un assegnamento restituisce il valore assegnato, dando la possibilità di fare cose come `a = b = 5`, che viene interpretato come `a = (b = 5)`, proprio perché `b = 5` restituisce il valore `5`, oppure `if(a = b)` che risulta falso se `b = 0` e vero se `b > 0`. Per fare un altro esempio, ci ricolleghiamo al secondo tipo di test, in cui si può sostituire all'espressione un comando, per esempio `x = (a == b) ? x = 1 : x = 0` il che è lecito in quanto un singolo comando può essere visto come un'espressione visto che restituisce il valore assegnato. Ciò non vale invece per blocchi di comandi: `x = (a == b) ? {x = 1} : {x = 0}` genera un errore.

Un'altra osservazione è che il comando di incremento `++` può essere visto come un'espressione con effetti collaterali. Questo comando ha due versioni:

- `++x`: pre–incremento, esegue `x = x + 1` e restituisce il valore incrementato
- `x++`: post–incremento, restituisce il valore di `x` e poi esegue `x = x + 1`

analogamente, esistono anche `x--` e `--x`.

Ci sono dei linguaggi in cui si decide in principio di non fare alcuna distinzione tra comandi ed espressioni.
Esempio: Algol68

```
1  begin
2      a := begin f(b); g(c) end;
3      g(d);
4      2 + 3;
5  end
```

dove `begin` ed `end` definiscono un blocco in cui c'è una sequenza di comandi. A riga 2 si fa un'assegnazione su `a` in cui la sequenza di comandi viene vista come un'espressione che ottiene gli effetti collaterali di `f(b)` e restituisce il valore dell'ultima funzione, in questo caso `g(C)`. A riga 3 c'è una chiamata di funzione il cui valore restituito verrà buttato via, ma l'importante è l'effetto collaterale. A riga 4 viene valutata l'espressione che restituisce il valore `5`.

5.4 Comandi per il controllo sequenza

Comandi per il controllo sequenza esplicito:

- ;
- blocchi
- GOTO

Comandi condizionali:

- if
- case

Comandi iterativi (cicli):

- iterazione indeterminata (while)
- iterazione determinata (for, foreach)

Comandi sequenziali:

```
C1; C2;
```

la composizione sequenziale di comandi è il costrutto base dei linguaggi imperativi, nella sintassi ci sono due possibili scelte:

- ; separatore di comandi, non serve inserirlo nell'ultimo comando
- ; terminatore di comandi, bisogna inserirlo anche dopo l'ultimo comando. C e derivati usano questa sintassi

5.4.1 Blocchi

Sintassi:

```
{                   begin
    ...                 ...
}                   end
```

servono per trasformare una sequenza di comandi in un singolo comando e/o per introdurre variabili locali.

5.4.2 GOTO – istruzione salto

```
1  if a < b GOTO 10
2  ...
3  10: ...
```

si introduce un meccanismo di etichette con cui si possono etichettare porzioni di codice e poi con l'istruzione GOTO si può saltare del codice per raggiungere una certa etichetta. È il meccanismo base in assembly, permette una notevole flessibilità, ma rende i programmi poco leggibili e nasconde gli errori. Già da molti anni il GOTO è considerato dannoso e contrario ai principi della programmazione strutturata.

Il teorema di Boehm–Jacopini dice che il GOTO è sostituibile dai costrutti while e test. Nella pratica, la rimozione del GOTO non porta a una grossa perdita di flessibilità espressiva. In molti linguaggi moderni ci sono comunque alcune istruzioni di salto, ma sono specifiche di alcuni contesti, per esempio:

- uscita alternativa da un loop: `break`
- ritorno da sottoprogramma: `return`
- gestione eccezioni: `raise exception`

Il return non è sostituibile dal GOTO perché il return gestisce anche i record di attivazione. Nei linguaggi che prediligono la sicurezza e la chiarezza (Java), il GOTO non è presente.

Programmazione strutturata

Metodologia introdotta negli anni 70, per gestire la complessità del software, i cui principi sono i seguenti:

- progettazione gerarchica, top–down (il problema viene scomposto in sottoproblemi)
- modularizzare il codice (mantenere la struttura top–down anche nel codice, un sottoprogramma per un sottoproblema)
- uso di nomi significativi (migliore manutenibilità)
- uso estensivo dei commenti
- tipi di dati strutturati
- uso dei costrutti strutturati per il controllo
  - ogni costrutto, pezzo di codice, ha un unico punto di ingresso e di uscita

- le singole parti della procedura sono modularizzate
- i diagrammi di flusso non sono più necessari, in quanto il programma si auto-descrive poiché strutturato

5.4.3 Comandi condizionali – if then else

```
if(B) {C_1};
if(B) {C_1} else {C_2};
// oppure, in base al linguaggio
if B then C_1 else C_2
```

Questo costrutto è potenzialmente ambiguo:

```
if(i == 2) if(i == 1) printf("%d \n", i); else printf(...)
```

non è esplicito a quale if l'else fa riferimento. Per risolvere si introducono varie regole in base al linguaggio di programmazione specifico, per esempio in C l'else fa riferimento all'if più vicino, in Algol68 e Fortran77 si usano le parole chiave `endif` o `fi` per marcare la fine. È possibile definire rami multipli espliciti con il comando `else if`

```
1  if(Bexp1) {C1}
2      else if(Bexp2) {C2}
3      ...
4      else if(Bexpn) {Cn}
5      else {Cn + 1}
```

5.4.4 Condizioni in Scheme

La possibilità di avere delle espressioni condizionali è presente anche nei linguaggi funzionali, in Scheme è infatti presente la seguente espressione:

```
(if test-expr then-expr else-expr)
```

L'espressione viene eseguita valutando prima `test-expr` e in seguito, a seconda dell'esito della valutazione precedente, `then-expr` oppure `else-expr`. Notare che il risultato della valutazione di `test-expr` non deve essere per forza un booleano, `then-expr` verrà eseguita in ciascun caso tranne quando la valutazione di `test-expr` non sia `#f`. Esempi:

```
1  > (if (positive? -5) (error "doesn't get here") 2)
2  2
3  > (if (positive? 5) 1 (error "doesn't get here"))
4  1
5  > (if 'we-have-no-bananas "yes" "no")
6  "yes"
```

Comando condizionale in scheme

In Scheme è presente anche uno speciale comando condizionale `cond` che prende come input una serie di coppie `[(guardia) (espressione)]` e valuta l'espressione in coppia con la prima guardia che risulta essere vera.

```
(cond
    [(positive? -5) (error "doesn't get here")]
    [(zero? -5) (error "doesn't get here, either")]
    [(positive? 5) 'here])
```

5.4.5 Case

Si tratta di una versione più complessa del "if then else", dove la guardia non è un booleano bensì un'espressione qualsiasi, come mostrato nel seguente esempio:

```
case exp of                 {* exp: espressione a
                               valori discreti *}
    | const_1 : C_1
    | const_2 : C_2
    ...                     {* const valori costanti,
                               disgiunti *}
    | const_n : C_n         {* di tipo compatibile con exp *}
    else C_n+1
```

Nei vari linguaggi ha diversi nomi, ma la struttura è spesso simile a quella dell'esempio, dove si indica l'inizio del comando e un'espressione da valutare. In seguito si indicano i vari casi denotandoli con un `|`, indicando il valore (o i valori quando può essere espresso un range) per cui l'espressione va valutata e l'espressione da valutare. Si può descrivere anche un caso default che viene valutato se l'esito della valutazione dell'espressione non rientra tra i casi possibili.

In alcuni linguaggi funzionali, tra cui Haskell, il costrutto `case` diventa piuttosto sofisticato e prende il nome di **pattern matching**. Esempio:

```
case (n, xs) of
    | (0, _) => []
    | (_, []) => []
    | (n, (y:ys)) => y : take (n-1) ys
```

in questo caso, l'espressione da valutare consiste in una coppia dove il primo valore è un numero naturale e il secondo è una lista. Il `_` indica che l'elemento può assumere qualsiasi valore.

Il costrutto case non è presente solo nei linguaggi funzionali ma anche nei linguaggi imperativi, infatti è presente anche in C e nei suoi derivati, dove ha la seguente sintassi:

```
int i ...
switch (i){
    case 3:
        printf("Case3 ");
        break;
    case 5:
        printf("Case5 ");
        break;
    default:
        printf("Default ");
}
```

come si può notare, in C il case prende il nome di switch, l'inizio del caso si indica con `case` e non con `|` e viene esplicitato il comando `break` alla fine di ogni caso.

Il `break` non è strettamente necessario, se non lo si mettesse infatti l'esecuzione procederebbe con i comandi successivi. In C può anche essere utile per evitare di scrivere più volte lo stesso comando, come nell'esempio seguente, nel quale il caso 2 sfrutta il comando presente nel caso 3:

```
switch (i){
    case 1:
        printf("Case1 ");
        break;
    case 2:
    case 3:
        printf("Case2 or Case 3");
        break;
    default:
        printf("Default ");
}
```

Questo tipo di utilizzo può portare però ad errori. Per evitare questa situazione, alcuni linguaggi preferiscono vietarlo o permetterlo se indicato in modo esplicito come nel caso del C#, dove se si omette il `break` si ottiene un errore, ma si può continuare l'esecuzione esplicitandone l'intenzione usando il comando `continue`.

Nei linguaggi più recenti è possibile anche esplicitare un range di valori:

```
switch (arr[i]){
    case 1 ... 6:
        printf("%d in range 1 to 6\n", arr[i]);
```

```
4           break;
5       case 19 ... 20:
6           printf("%d in range 19 to 20\n", arr[i]);
7           break;
8       default:
9           printf("%d not in range\n", arr[i]);
10          break;
```

5.4.6 Implementazione del costrutto case

Volendo, il case si potrebbe scrivere come una serie di if controllando tutti i casi, ma facendo in questo modo il codice è più prolisso e inefficiente, questo perché con l'if bisogna fare tutti i confronti e quindi ha un costo lineare sul numero di confronti, mentre il case valuta l'espressione e usa il risultato come entry a una tabella di istruzioni di salto, dove, per ogni valore dell'espressione tra quelli ammessi, trova un'istruzione di salto che porta al codice macchina da eseguire, ottenendo quindi un accesso diretto.

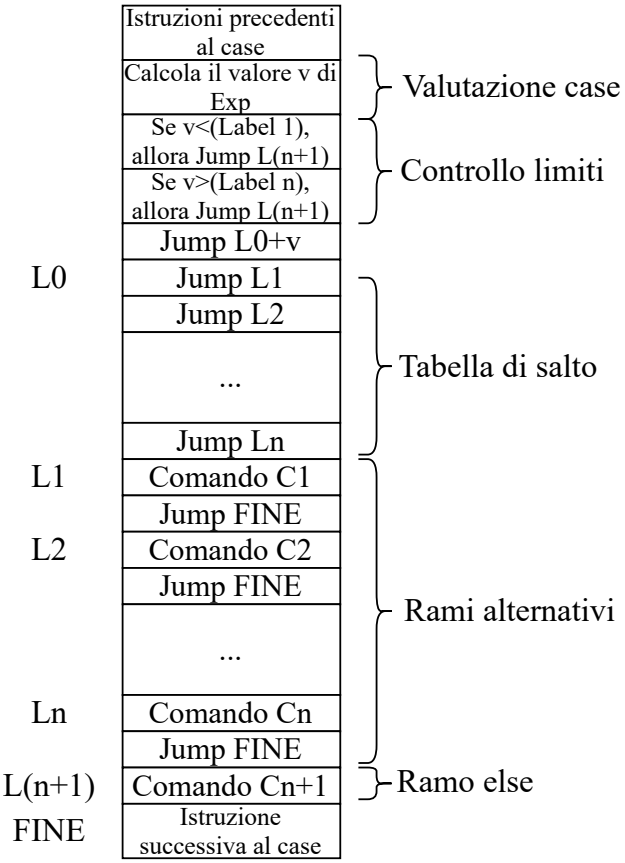


Figura 5.4: Struttura del codice macchina generato per lo switch

Nell'immagine vediamo che all'inizio viene valutata l'espressione che compare come argomento del case; il valore ottenuto è quindi usato come offset (indice) per calcolare la posizione, nella tabella di salto, dell'istruzione che permette il salto al comando del ramo scelto. Il ramo else è invece fatto sulla base del controllo esplicito dei limiti delle etichette. I Jump FINE dopo ogni comando sono il corrispettivo del comando `break`.

Lo schema precedente funziona bene in quanto, come detto, il tempo di esecuzione è costante e l'utilizzo di memoria è limitato, ma solo se anche il range di valori è limitato e non disperso. Con range troppo ampi, infatti, l'occupazione di memoria è troppo alta perché la tabella deve avere tante entry quanti sono i possibili valori che può assumere l'espressione. Un altro caso in cui c'è uno spreco di memoria è quando i range (`case n1 ... n`) sono ampi, perché bisogna ripetere la stessa istruzione di salto per ogni valore nel range. È possibile ridurre l'occupazione di memoria con una ricerca binaria oppure con una tabella hash, ottenendo un tempo di esecuzione rispettivamente logaritmico o costante, avendo però un codice macchina più complesso.

5.5 Iterazione

L'iterazione permette di creare linguaggi Turing completi, ovvero permettono di definire tutte le funzioni calcolabili, senza questo meccanismo non sarebbe possibile la ripetizione di comandi e la computazione terminerebbe sempre in un numero limitato di passi. Possiamo dividere l'iterazione in due gruppi:

- iterazione indeterminata: i cicli vengono controllati logicamente (while, repeat, ...)
- iterazione determinata: i cicli vengono controllati numericamente (do, for, foreach, ...), con numero di ripetizioni del ciclo determinate al momento dell'inizio del ciclo

5.5.1 Iterazione indeterminata

La sintassi più usata è: `WHILE condizione DO comando`.

```
1  // in Java
2  while (counter > 1){
3      factorial *= counter--;
4  }
5
6  // in Pascal
7  while Counter > 0 do
8  begin
9      Factorial := Factorial * Counter;
```

```
10      Counter := Counter - 1
11  end;
```

L'idea del `WHILE` consiste nel voler eseguire un numero di ripetizioni di comandi maggiore o uguale a zero in base alla valutazione della guardia. In alcuni casi si vorrebbe obbligare l'iterazione dei comandi per almeno una volta, per queste situazioni è presente una versione detta **post test**. Con questo costrutto semplicemente si esegue il test dopo la prima iterazione cosicché, indipendentemente dall'esito della valutazione, almeno un'esecuzione venga effettuata.

```
1   // In c, c++, Java, ...
2   do {
3       factorial *= counter--; // Multiply, then decrement
4   } while (counter > 0);
5
6   // In Ruby
7   begin
8       factorial *= counter
9       counter -= 1
10  end while counter > 1
11
12  // In Pascal si ripete finché la guardia non diventa true
13  repeat
14      Factorial := Factorial * Counter;
15      Counter -:= 1
16  until Counter = 0
```

I due cicli sono equivalenti, è facile sostituire l'uno con l'altro ma in generale uno dei due risulta più sintetico in base al contesto di utilizzo.

```
1   do {
2       do_work();
3   } while (condition);
4
5   equivalente a:
6
7   do_work();
8   while (condition) {
9       do_work();
10  }
```

Nell'iterazione indeterminata il numero di ripetizioni non è noto a priori, nonostante questo permette di definire tutte le funzioni calcolabili.

L'implementazione è abbastanza semplice e diretta dato che basta trasformare ogni ciclo in un paio di istruzioni di salto.

5.5.2 Iterazione determinata

Nell'iterazione determinata all'inizio viene imposto il numero di ripetizioni da eseguire.

Il costrutto per l'iterazione determinata è:

```
1  FOR indice := inizio TO fine STEP passo DO
2      ....
3  END
```

con i seguenti vincoli:

- i valori `indice`, `fine` e `passo` vengono valutati e salvati all'inizio dell'esecuzione e non possono essere modificati dall'interno del loop.
- il valore di `indice` può essere preesistente oppure essere creato dal `FOR` stesso; viene modificato sempre dal `FOR` in base al valore di `passo` che, se non espresso, di solito ha valore 1
- `passo` è una costante diversa da 0
- ad ogni iterazione viene eseguito il test `indice > fine` se `passo < 0`, altrimenti `indice < fine` se `passo > 0`

Se queste regole vengono rispettate il potere espressivo è minore delle iterazioni indeterminate dato che non permette di esprimere computazioni che non terminano, ma risulta comunque preferibile al `while` siccome garantisce la terminazione (il for è simulabile con il while ma la terminazione non è comunque garantita) e spesso ha una scrittura più semplice ed è più compatto.

Bisogna tenere presente che in C ed i suoi derivati il `for` disponibile non è quello appena descritto siccome è permessa la modifica degli indici.

Diversi linguaggi possono differire leggermente nell'implementazione del `for`:

- possibilità di modificare indice, valore finale, passo nel loop (se sì, non si tratta di vera iterazione determinata)
- se il ciclo termina quando l'indice ha il valore `fine` oppure quando lo supera
- numero di iterazioni (dove avviene il controllo indice $<$ `fine`)
- possibilità incremento negativo
- valore indice al termine del ciclo: indeterminato, `fine`, `fine + 1`

Iterazione determinata in C, C++, Java

La sintassi è:

```
for(initialization; condition; increment/decrement)
    statement
```

per esempio:

```
int sum = 0;
for(int i = 1; i < 6; ++i){
    sum += i;
}
```

Altri linguaggi

Python:

```
for counter in range(1, 6): # 1, 2, 3, 4, 5
    # statements
```

Ruby:

```
for counter in 1..5
    # statements
```

Foreach

L'idea del Foreach è di ripetere il ciclo su tutti gli elementi di un oggetto enumerabile, come array, liste, insiemi, alberi. Di seguito un esempio in Java (dalla versione 5):

```
int [] numbers = {10, 20, 30, 40, 50};

for(int x : numbers){
    System.out.print(x + ",");
}
```

dove la variabile `x` assume tutti i valori della lista.

Il potere espressivo è limitato, ma a volte è preferibile perché è più chiaro, inoltre separa l'algoritmo di scansione della struttura dati (che viene generato in automatico dal compilatore) dalle operazioni da svolgere sui singoli elementi. Può essere utilizzato su strutture dati che mettano a disposizione funzioni implicite per determinare il primo elemento, l'elemento successivo e il test di terminazione.
Altri esempi:
In Python

```
1  pets = ['cat', 'dog', 'fish']
2  for f in pets:
3      print f
```

In Ruby, per fare la stessa cosa ci sono due metodi:

```
1  pets = ['cat', 'dog', 'fish']
2  pets.each do |f|
3      f.print
4  end
5  #oppure
6  for f in pets
7      f.print
8  end
```

nel primo viene usato `each`, che esegue una funzione `f` per ogni elemento dell'oggetto. Tra `| |` si inserisce il nome della funzione da eseguire e poi si scrive il corpo della funzione. Il secondo metodo è invece più simile alla controparte in Python.
In JavaScript:

```
1  var numbers = [4, 9, 16, 25];
2  function myFunction(item, index){
3      ...;
4  }
5  numbers.forEach(myFunction)
```

in riga 5 al foreach viene passata la funzione come argomento.

5.6 Ricorsione

La ricorsione è un meccanismo base ed alternativo all'iterazione per implementare la ripetizione di comandi nei linguaggi funzionali, dove l'iterazione con cicli non è possibile direttamente, permettendo di ottenere lo stesso potere espressivo dei linguaggi imperativi.
Si dice che una funzione è ricorsiva se è definita in termini di sé stessa.

Le funzioni matematiche di natura induttiva sono facilmente traducibili in una funzione ricorsiva, come nel caso della funzione fattoriale, definita matematicamente come:

$$fattoriale(0) = 1$$

$$fattoriale(n) = n * fattoriale(n-1)$$

che viene tradotta in:

```
1  int fatt (int n){
2      if (n <= 1)
3          return 1;
4      else
5          return n * fatt ( n-1 );
6  }
```

5.6.1 Definizione induttiva di funzioni

In matematica ricorsione ed induzioni hanno senso in strutture che vengono chiamate induttive, l'esempio più semplice è l'insieme dei numeri naturali. Nel caso dei numeri naturali, l'insieme è una struttura ricorsiva dato che è definito come il minimo insieme X che rispetta i seguenti assiomi:

- 0 è in X
- se n è in X allora succ(n) è X

Data una struttura ricorsiva è naturale definire un principio di induzione, nel caso dei numeri naturali abbiamo che una proprietà P è vera se:

- $P(0)$ è vera
- $\forall n \in N : P(n)\ vera \Rightarrow P(n+1)\ vera$

Analogamente al principio di induzione, si può definire il principio di ricorsione (definizione induttiva di funzioni): se $g : (Nat\ x\ A) \rightarrow A$ totale allora esiste una unica funzione totale $f : Nat \rightarrow A$ tale che

- $f(0) = a$
- $f(n+1) = g(n, f(n))$

In matematica si può dimostrare che una funzione definita seguendo la struttura precedente risulta ben definita, ovvero definisce univocamente una funzione totale.

Tale struttura risulta però limitante nella pratica data la difficoltà di codificare alcune funzioni in tale modo, allora si può generalizzare la definizione induttiva per avere uno schema più flessibile mantenendo una buona definizione.

A differenza delle funzioni in matematica, i linguaggi di programmazione permettono definizioni non corrette, prendiamo per esempio la seguente definizione:

$$foo(0) = 1$$

$$foo(n) = foo(n+1) - 1$$

Tale definizione non è corretta e non definisce quindi nessuna funzione, se la traduciamo in codice otteniamo un programma corretto anche se per alcuni input potrebbe non terminare.

```
1  int foo (int n){
2  if (n == 0)
3      return 1;
4  else
5      return foo(n+1) - 1
6  }
```

La ricorsione è possibile in ogni linguaggio che permetta:

- funzioni (o procedure) che possono chiamare sé stesse
- gestione dinamica della memoria (pila)

Ogni programma ricorsivo può essere tradotto in uno equivalente iterativo e viceversa.

5.6.2 Ricorsione ed iterazione

Se entrambe sono possibili, usare la ricorsione risulta essere più naturale per lavorare con strutture dati ricorsive come gli alberi, e quando la struttura del problema è ricorsiva. Viceversa, in altre strutture come matrici e vettori l'iterazione è più efficiente. Generalmente la ricorsione si usa nei linguaggi funzionali dove è una scelta obbligata a causa dell'assenza dell'iterazione, nei linguaggi imperativi invece si preferisce usare l'iterazione.

La ricorsione, se implementata in maniera naive, risulta inefficiente rispetto ad un programma equivalente scritto utilizzando l'iterazione, questo avviene perché ad ogni chiamata della funzione a sé stessa bisogna allocare un nuovo record di attivazione, operazione che costa in termini di spazio e tempo. Per risolvere il problema esistono due vie:

- usare un compilatore ottimizzato
- usare la ricorsione di coda

5.6.3 Ricorsione di coda

Una chiamata di `g` in `f` di si dice "in coda" (o tail call) se `f` restituisce il valore restituito da `g` senza nessuna ulteriore computazione.

`f` è tail recursive se contiene solo chiamate in coda a sé stessa.

```
1
2  function tail_rec (n: integer, m): integer
3  begin ... ; return tail_rec(n-1, m1) end
4
```

```
5
6   function non_tail rec (n: integer): integer
7   begin ... ; x:= non_tail_rec(n-1); return g(x) end
```

A differenza della ricorsione normale, nella ricorsione di coda non è necessario mantenere in memoria tutti i record di attivazione delle varie chiamate, dato che a ciascuna di esse lo stack utilizzato precedentemente non è più utile e può quindi essere rimosso e lo spazio da esso occupato può essere usato dalla nuova chiamata. Questo rende la ricorsione di coda efficiente quasi quanto un'iterazione.

5.6.4 Chiave di lettura della ricorsione di coda

La ricorsione di coda simula in un linguaggio funzionale un ciclo WHILE:

- per ogni funzione `f` che in un linguaggio imperativo avremmo implementato tramite un ciclo definiamo una funzione `f-helper` avente parametri extra
- questi parametri extra svolgono il ruolo delle variabili modificabili nel ciclo
- `f-helper` chiama sé stessa aggiornando i parametri extra, come avviene in un ciclo
- `f` chiama `f-helper` inizializzando i parametri extra (come la funzione imperativa)

In questo modo simuliamo uno stato in maniera locale e controllata, senza introdurre uno stato globale. La ricorsione di coda andrebbe utilizzata solo nelle funzioni più critiche (per la velocità d'esecuzione globale del programma).

5.6.5 Esempi di ricorsione di coda

Il primo esempio consiste nel calcolo del fattoriale, prendiamo innanzitutto il caso di un'implementazione con ricorsione normale:

```
1   int fatt (int n){
2       if (n <= 1)
3           return 1;
4       else
5           return n * fatt ( n-1 );
6       }
```

Lo stack di attivazione per calcolare la funzione `fatt(n)` risulta essere:

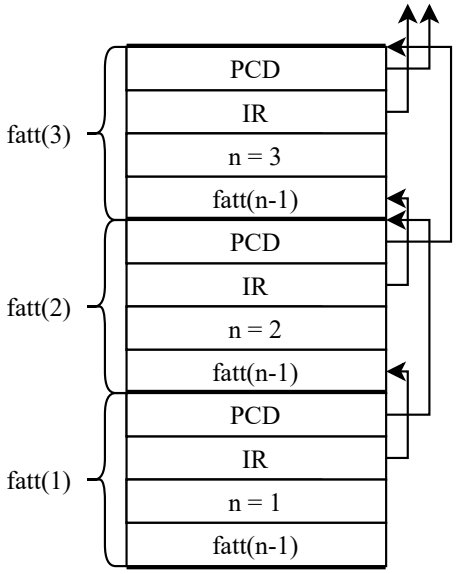


Figura 5.5: Evoluzione dello stack di attivazione

Come si vede, ad ogni chiamata ricorsiva viene occupata una nuova locazione di memoria che non potrà essere rilasciata prima della fine delle chiamate ad essa consecutive. Analizziamo cosa succede nel caso del fattoriale con ricorsione di coda.

```
1   int fattrc (int n, int res){
2   if (n <= 1)
3       return res;
4   else
5       return fattrc ( n - 1, n * res);
6   }
7
8   int fatt (int n){
9       fattrc (n, i)
10  }
```

In questo caso si utilizzano due procedure, la prima implementa la ricorsione di coda, attraverso il parametro `res` si memorizza il risultato delle chiamate ricorsive fino al caso attuale, ritornato nel caso base.

Dato che la prima è una procedura a due argomenti, è necessaria la seconda procedura per poter utilizzare il fattoriale come procedura a un argomento.

Si nota immediatamente che è necessario solo un RdA per la procedura, ogni chiamata ricorsiva non è più necessaria una volta chiamata la successiva, può essere quindi eliminato e lo spazio riutilizzato per la chiamata successiva.

Fibonacci

La funzione di Fibonacci è definita come:

```
Fib(0) = 1
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2)
```

che è facilmente traducibile in una procedura scritta in Scheme:

```
1  (define (fib n)
2      (if (< n 2)
3          1
4          (+ (fib (- n 1)) (fib (- n 2)))))
```

La funzione di Fibonacci implementata in questo modo rende la procedura estremamente inefficiente: ad ogni esecuzione della procedura (con `n` maggiore o uguale a 2) vengono chiamate altre due procedure, la complessità cresce quindi in tempo e spazio esponenziale (più correttamente ha una crescita proprio alla Fibonacci). Anche la funzione di Fibonacci può essere trasformata utilizzando la ricorsione di coda:

```
1  (define (fibHelper n a b)
2      (if (= n 0)
3          b
4          (fibHelper (- n 1) b (+ a b)))
5
6  (define (fib n)
7      (if (= n 0)
8          1
9          (fibHelper (- n 1) 1 1))
```

In questo caso la complessità diventa lineare per il tempo, per lo spazio invece diventa costante dato che si usa un solo RdA.

Capitolo 6

Astrarre sul controllo

In questo capitolo verranno approfondite le procedure e le funzioni, definendo le modalità di passaggio dei parametri, le funzioni di ordine superiore, che possono essere usate come parametri o come risultato, e i gestori delle eccezioni.

Riprendiamo brevemente il concetto di astrazione. Come abbiamo detto è un meccanismo per gestire la complessità:

- identifica le proprietà importanti di cosa si vuole descrivere
- nasconde e ignora gli aspetti secondari
- permette di descrivere e concentrarsi solo sulle questioni rilevanti

in generale, permette di evidenziare tratti comuni in strutture diverse (matematica) e spezzare un sistema complesso in sottosistemi descritti astrattamente, senza entrare nel dettaglio. In informatica permette di avere delle istruzioni più potenti e astratte, e dei dati che, se non sono dei semplici interi ma sono più complessi, possono essere visti in modo più strutturato.

La definizione di una procedura–funzione principale è un meccanismo di astrazione sul controllo, per esempio definendo la procedura:

```
1  float log(float x){
2      double z;
3      /* CORPO DELLA FUNZIONE */
4      return expr;
5  }
```

possiamo usare `log` conoscendo le sue specifiche, ma senza conoscerne i dettagli di implementazione (per esempio come il logaritmo viene approssimato). Possiamo specificare `log` senza conoscere l'implementazione. Possiamo implementare `log` (scriverne il codice) rispettando le specifiche e senza conoscere il contesto in cui verrà usato.

6.1 Parametri

Un po' di terminologia per i parametri. Nella seguente dichiarazione/definizione:

```
int f(int n) {return n + 1;}
```

`n` è detto **parametro formale**, che è il parametro che appare nella dichiarazione e che sarà usato all'interno del corpo della funzione, dove rappresenta l'istanza di argomento con cui `f` verrà chiamata.

Nella seguente chiamata di funzione:

```
x = f(y + 3);
```

`y + 3` è detto **parametro attuale** ed è il valore che verrà sostituito al parametro formale.

Possiamo distinguere tra:

- funzioni: restituiscono un valore
- procedure: non restituiscono nulla

Questa distinzione è molto simile a quella tra espressioni e comandi. In generale, con le funzioni siamo più interessati al valore restituito, mentre con le procedure siamo più interessati all'effetto collaterale. Nel caso di C e derivati, tutto ciò che viene definito sono funzioni e, se non dovesse restituire niente, viene comunque restituito un valore di tipo `void`.

6.2 Comunicazione chiamato – chiamante

Una funzione chiamante può scambiare dei dati con la funzione chiamata e viceversa attraverso tre metodi principali:

- valore restituito con `return`
- passaggio di parametri, per cui esistono varie modalità
- effetti collaterali che provocano modifiche all'ambiente non locale. Questo riduce il livello di astrazione perché chi usa la funzione deve conoscere anche quali modifiche vengono fatte (nei linguaggi funzionali questa possibilità non è presente, per cui sono più astratti)

6.3 Modalità di passaggio dei parametri

Si può distinguere tra:

- parametri d'ingresso: main $\rightarrow$ proc

- parametri d'uscita: main $\leftarrow$ proc
- parametri d'ingresso – uscita: main $\leftrightarrow$ proc

Ci sono varie modalità di passaggio con diversa semantica, implementazione e costo, ma ci concentreremo sulle 2 principali:

- passaggio per **valore** (call by value): il parametro formale diventa una variabile locale, il parametro attuale viene valutato e assegnato al formale. In questo modo abbiamo una comunicazione main $\rightarrow$ proc perché le modifiche al formale non passano all'attuale. Il parametro attuale è un r-value che può essere qualsiasi espressione
- passaggio per **riferimento** (call by reference): il parametro attuale non è un r-value, bensì un l-value, ovvero la locazione di memoria dove sta effettivamente il dato. Al momento del passaggio, il parametro attuale e formale fanno riferimento allo stesso dato. L'effetto è che tutto ciò che accade all'interno della procedura chiamata ha effetto anche sul chiamante, abbiamo dunque una comunicazione main $\leftrightarrow$ proc

6.3.1 Passaggio per valore

```
1  void foo(int x){x = x + 1;}
2  ...
3  y = 1;
4  foo(y + 1);
5  foo(y);
```

Quando `foo` viene chiamata si crea un RdA in cui il parametro formale `x` è una variabile locale; viene valutata l'espressione `y + 1` ottenendo il valore `2` che viene assegnato a `x` nel RdA ed incrementato. Il valore di `y` non subisce modifiche.

Questo meccanismo si basa sull'assegnamento, dunque può essere costoso per dati di grande dimensione in quanto ne viene fatta una copia. L'implementazione consiste nell'allocare una variabile nel RdA associata al parametro formale e al momento della chiamata, il parametro attuale viene valutato e nel RdA inserito il suo valore.

6.3.2 Passaggio per riferimento

```
1  void foo(reference int x){x = x + 1;}
2  ...
3  y = 1;
4  foo(y);
5  foo(V[y + 1]);
```

Il parametro attuale `y` viene valutato come l-value, dunque non come valore `1`, ma come locazione di memoria associata a `y`, che viene passata alla procedura chiamata e assegnata a `x`. La modifica che `foo` fa al dato ha quindi effetto anche su `y` nella procedura chiamante. Abbiamo perciò una comunicazione bidirezionale main $\leftrightarrow$ proc. L'implementazione consiste nell'inserire un puntatore nello stack di attivazione che va dal parametro formale a quello attuale. Dal punto di vista dell'efficienza è ovviamente migliore in quanto è sufficiente passare un puntatore e non un dato che può essere di grande dimensione. Dal punto di vista del codice invece è meno efficiente perché per accedere al dato bisogna prima seguire il puntatore per trovarlo.

Raramente nei linguaggi di programmazione questo è il meccanismo di default, Fortran è probabilmente l'unico linguaggio in cui lo è. In altri linguaggi come C++, PHP, Visual Basic, .NET, C#, REALbasic e Pascal questo meccanismo è offerto come opzione usando parole chiave come `ref` in C++ e C#

```
foo(ref int x)
```

In altri linguaggi come in C, ML e Rust invece è simulabile passando un puntatore/riferimento/indirizzo di memoria (call by address) ottenendo lo stesso effetto del passaggio per riferimento. In Java può essere simulato usando i tipi riferimento (call by sharing).
Simulazione in C:

```
1  void foo(int *x){*x = *x + 1;}
2  ...
3  y = 1;
4  foo(&y);
```

dove, ricordiamo, se `x` è un puntatore, allora `*x` indica l'oggetto puntato da `x`. Allo stesso modo, nella chiamata non bisogna passargli `y`, ma il puntatore alla locazione di memoria associata a `y` con `&y`.

6.3.3 Call by sharing

Tipicamente usato nei linguaggi di programmazione ad oggetti, con call by sharing si intende che quando c'è una chiamata a una procedura, il chiamante non passa una copia del dato, ma un riferimento ad esso. Questo perché parecchi dati in Java, Python, Ruby e JavaScript sono visti come dei riferimenti, ricordiamo la distinzione tra tipo di dato mutabile e immutabile, dove, se il dato è mutabile, viene passato un riferimento e quindi dopo un'assegnazione si crea una condivisione del dato tra due variabili; stessa cosa accade con il passaggio dei parametri.

Parametro attuale e formale devono avere lo stesso tipo riferimento, che normalmente sono più complessi di un intero e sono tipi strutturati, array, classi. Le modifiche nella funzione del parametro formale si ripercuotono sul parametro attuale perché sono due riferimenti alla stessa copia dell'oggetto.

Esempio in Python di call by sharing:

```
1  def f(a_list):
2      a_list.append(1)
3
4  m = []
5  f(m)
6  print(m)
```

stampa `[1]` perché la lista è un oggetto mutabile e dunque viene passato il riferimento e la lista viene condivisa, ma

```
1  def f(a_list):
2      a_list = [1]
3
4  m = []
5  f(m)
6  print(m)
```

stampa `[]`, perché nella procedura viene creato un nuovo vettore ed assegnato ad `a_list`.

Esempio in Java:

```
1  class A{
2      int v;
3  }
4  A y = new A(3) // y.v = 3
5  ...
6  void foo(A x){x.v = x.v + 1;}
7  ...
8
9  foo(y);
```

Le classi sono un tipo riferimento, l'assegnamento avviene copiando il riferimento (puntatore). La procedura `foo` dell'esempio porta ad una modifica al campo `v` dell'oggetto `y` in quanto viene passato un riferimento ad esso.

Lo stesso esempio con la classe `Integer` avrebbe un comportamento diverso: la chiamata avrebbe comunque creato una condivisione, ma l'istruzione `x = x + 1;` crea un nuovo intero e lo assegna a `x`. **Il comportamento dipende dal tipo di dato.**

Il nome call by sharing non è standard, infatti in ambito Java si parla di call by value; il nome call by sharing mette in evidenza che

- il passaggio del parametro crea una copia condivisa tra parametro attuale e formale
- eventuali modifiche nel chiamato si possono ripercuotere sul chiamante

Riassumendo, è importante sapere:

- se il passaggio del parametro crea una copia condivisa o una nuova copia
- se l'assegnamento modifica l'oggetto puntato o crea un nuovo oggetto

quindi:

- copia condivisa e oggetti modificabili: comportamento simile al call by reference
- copia condivisa e oggetti non modificabili (l'assegnazione crea una nuova istanza): comportamento simile al call by value

6.3.4 Riassunto: value, reference, sharing

Passaggio per valore:

- semantica semplice: si crea una copia locale e le modifiche non hanno effetto all'esterno, comportamento prevedibile
- implementazione abbastanza semplice
- costoso il passaggio per dati di grande dimensione
- efficiente il riferimento al parametro formale (accesso diretto al dato)
- necessità di altri meccanismi per comunicare main $\leftarrow$ proc

Passaggio per riferimento:

- semantica complessa: si crea aliasing, comportamento meno prevedibile
- implementazione semplice
- efficiente il passaggio, non bisogna fare una copia
- un po' più costoso il riferimento al parametro formale (accesso prima al puntatore, poi al dato, indiretto)

Call by sharing:

- a seconda dei casi, simile a call by value o a call by reference

6.3.5 Passaggio per costante (o read only)

Il passaggio per costante ha i vantaggi del passaggio per riferimento, in cui quando bisogna passare un dato di grosse dimensioni basta passare un riferimento ad esso, ma ha anche la chiarezza del passaggio per valore. Per ottenere ciò bisogna definire l'argomento della procedura (il parametro formale) come una costante, dunque si dichiara che quel valore non verrà mai modificato all'interno della procedura, nello specifico non si potrà fare l'assegnamento o passarlo per riferimento ad altre procedure che possono modificarlo; questo controllo viene fatto dal compilatore (per esempio, C dà solo un warning, invece Java dà un errore). Il risultato è che passare quel dato per valore o per riferimento ha lo stesso identico comportamento, dunque nell'implementazione la scelta viene lasciata al compilatore: di solito i parametri "piccoli" vengono passati per valore, mentre quelli "grandi" per riferimento. Per farlo si usano delle parole chiave specifiche per ogni linguaggio di programmazione.

In C:

```
1  int foo(const char *a1, const char *a2){
2      /* le stringhe a1 e a2 non possono essere modificate */
3  }
```

In Java:

```
1  void foo(final A a){
2      // qui a non può essere modificato
3  }
```

6.3.6 Passaggio per risultato

Il passaggio per risultato è l'esatto duale del passaggio per valore. Si tratta di una modalità che realizza una comunicazione di sola uscita dalla procedura main ← proc. L'ambiente locale della procedura è esteso con un'associazione tra il parametro formale e una nuova variabile. Il parametro attuale deve essere un'espressione l-value. Al momento della terminazione della procedura, subito prima della distruzione dell'ambiente locale, il valore corrente del parametro formale viene assegnato alla locazione ottenuta mediante il l-value dell'attuale.

```
1  void foo(result int x){x = 8;}
2  ...
3  y = 1;
4  foo(y);
```

Non c'è nessun legame tra la `y` iniziale del chiamante e la `x` nel corpo di `foo` (infatti `x` viene ridefinita assegnandogli un valore), dunque non è possibile trasmettere dati alla procedura mediante il parametro. Nella pratica, questo metodo è usato in alcuni linguaggi di programmazione nei casi in cui sia utile che una funzione possa ritornare più di un singolo risultato.

6.3.7 Passaggio per valore-risultato

La combinazione del passaggio per valore e del passaggio per risultato dà luogo alla modalità detta per valore-risultato. Si tratta di una modalità che realizza una comunicazione bidirezionale main $\leftrightarrow$ proc senza però che ci siano gli effetti di aliasing. Il parametro attuale deve essere un'espressione l-value. Alla chiamata, il valore dell'attuale è assegnato al formale e al ritorno, il valore del formale è assegnato all'attuale.

```
1   void foo(value-result int x){x = x + 1;}
2   ...
3   y = 8;
4   foo(y);
```

Durante l'esecuzione di `foo`, dentro a `x` non viene messo un riferimento a `y`, ma viene creata una variabile indipendente a cui viene assegnato il valore di `y`. Nel corpo di `foo` non esiste nessun legame tra `x` e `y`. Una volta terminata la procedura il valore finale di `x` viene copiato dentro `y`. In questo esempio se si fosse usato il passaggio per riferimento non ci sarebbero state differenze, ma è facile pensare ad un esempio in cui, a causa dell'aliasing, si avrebbe avuto un risultato differente. Esempio:

```
1   void foo(ref int x, ref int y, ref int z){
2       y = 2;
3       x = 4;
4       if(x == y) z = 1;
5   }
6   ...
7   int a = 3;
8   int b = 0;
9   foo(a, a, b);
```

Con il passaggio per riferimento abbiamo che `y` e `x` puntano alla stessa locazione di memoria e dunque nell righe 2 e 3 si modifica due volte la stessa cosa, di conseguenza a riga 4 abbiamo che `x` e `y` hanno entrambe valore 4 e a `z` viene assegnato il valore `1`.

```
1   void foo(value-result int x, value-result int y, ref int z){
2       y = 2;
3       x = 4;
```

```
4       if(x == y) z = 1;
5   }
6   ...
7   int a = 3;
8   int b = 0;
9   foo(a, a, b);
```

Con il passaggio per valore-risultato invece abbiamo che `x` e `y` sono due copie distinte e per cui avranno due valori diversi in riga 4, dunque `foo` termina senza assegnare il valore `1` a `z`.

Questa modalità funziona bene per dati di piccole di dimensioni, altrimenti, come nel caso di Ada, si usa il passaggio per riferimento per ovviare ai problemi di costo tipici del passaggio per valore.

6.3.8 Passaggio per nome

Questa modalità è stata introdotta in Algol-W in cui è il default, ora non viene usata da nessun linguaggio imperativo, ma è classica dei linguaggi funzionali.

```
1   int sel(name int x, y){
2       return (x == 0 ? 0 : y);
3   }
4   z = sel(w, z/w);
```

L'espressione del parametro attuale viene passata, ma non valutata. Ogni volta che nel corpo della funzione `sel` trova `x` o `y` valuta le espressioni `w` e `z/w`. Se nell'esempio avessimo dato a `w` valore `0`, avremmo avuto una divisione per zero nel secondo argomento, dunque con il passaggio per valore avremmo ottenuto un errore, mentre con il passaggio per nome quell'espressione non verrebbe mai valutata e la procedura ritornerebbe `0` (con il passaggio per riferimento avremmo ottenuto un errore a priori perché `z/w` non è un l-value).

Questa modalità può simulare il passaggio per riferimento:

```
1   int y;
2   void foo(name int x){x = x + 1;}
3   ...
4   y = 1;
5   foo(y);
```

In questo esempio, come abbiamo detto prima, `foo` viene eseguita sostituendo ogni istanza di `x` con `y` dunque alla fine anche `y` viene modificata. Nei casi in cui il parametro formale compaia come l-value, il compilatore controlla che anche il parametro attuale sia un l-value.

Cattura delle variabili

```
1  int y = 1;
2  void fie(name int x){
3      int y = 2;
4      x = x + y;
5  }
6  fie(y);
```

Il problema della cattura delle variabili si presenta quando ci sono più variabili con lo stesso nome, come la `y` dell'esempio. Quello che succede è che in riga 4 otteniamo `y = 2 + 2` dunque `y` avrà valore `4`, che non è ciò che di norma si vuole ottenere da un passaggio per nome, infatti la `y` che si vuole venga valutata è quella del parametro attuale, ovvero quella definita in riga 1.

Per ovviare a questo problema si procede nel seguente modo: al momento della chiamata, oltre all'espressione del parametro attuale bisogna fornire il suo ambiente di valutazione. La coppia `<exp,env>` prende il nome di **chiusura** ed è costituita da

- `exp`: un puntatore al testo di `exp`
- `env`: un puntatore di catena statica (sullo stack di attivazione) al record di attivazione del blocco di chiamata. In pratica diciamo che la valutazione dell'espressione nel corpo della procedura deve essere fatta usando come ambiente il record di attivazione puntato, facendo quindi in modo che alcune dichiarazioni più recenti che sono state fatte vengano nascoste.

Valutazione multipla

Se un parametro formale appare più volte nel codice, esso viene ovviamente valutato più volte. Nel caso in cui ci siano degli effetti collaterali nell'espressione, essi verranno quindi ripetuti ad ogni valutazione.

```
1  int V[] = {0, 1, 2, 3};
2  int y = 1;
3  void fie(name int x){
4      int y = 2;
5      x = x + y;
6  }
7  fie(V[y++]);
```

Nell'esempio, abbiamo che ad ogni valutazione del parametro formale, il parametro attuale `y` viene incrementato, ottenendo quindi un risultato inaspettato.

I linguaggi funzionali si dividono in eager e lazy, in quelli eager le espressioni dei parametri vengono valutate subito, mentre nei lazy viene usato il passaggio per nome. Nei linguaggi funzionali lazy in cui viene usato questo metodo, si adotta un accorgimento per evitare questo problema: il parametro viene comunque passato per nome e non valutato subito, ma all'interno della procedura, quando si presenta il parametro formale per la prima volta, esso viene valutato e il risultato viene salvato e riutilizzato per le successive occorrenze, questa variante del metodo prende il nome di **call by need**. Spesso, i linguaggi funzionali lazy che usano questo meccanismo sono anche linguaggi funzionali puri, in cui le espressioni non possono avere effetti collaterali, questo per evitare inconsistenze tra call by need e call by name.

Simulazione del passaggio per nome

Nei linguaggi funzionali eager come Scheme, è possibile simulare il passaggio per nome. Il vantaggio è di avere migliori prestazioni oppure evitare errori come con la valutazione corto-circuito: valutare le espressioni solo se necessario. L'idea è che, invece di passare un'espressione che verrebbe per forza valutata, si passa una funzione con zero argomenti che viene considerata come un valore e dunque non viene valutata subito.
Esempio in Scheme:

```
1  (define (doublePlusOne e)
2      (+ (e) (e) 1)
3
4  (define x 2)
5
6  (doublePlusOne (lambda () (+ x 3)))
```

Nell'esempio si vuole passare l'espressione `x + 3`, ma senza forzarne la valutazione, quindi viene inglobata in una lambda espressione senza argomenti, chiamata in questo caso **thunk**; l'effetto è che verrà valutata solamente all'interno del corpo della procedura e solo se richiesto. Il termine thunk indica semplicemente il meccanismo generale per simulare la valutazione lazy (per nome) in linguaggi con valutazione eager (per valore). Nel corpo della procedura `doublePlusOne` alle due variabili `e` verrà sostituito `(+ x 3)` che avrà valore 5; l'espressione viene dunque valutata due volte.

6.4 Parametri di default

In alcuni linguaggi come Ada, C++, C#, Fortran, Python è possibile definire procedure con valori di default per alcuni argomenti, questo comporta il fatto che sia possibile fornire meno argomenti nella chiamata, nel cui caso la procedura userà gli argomenti di default definiti.
Esempio in Python:

```
1  def printData(firstname, lastname = 'Mark', subject =
       'Math'):
2      print(firstname, lastname, 'studies', subject)
3
4  #1 positional argument
5  printData('John')
6  # 2 positional arguments
7  printData('John', 'Gates')
8  printData('John', 'Physics')
9
10 # Output:
11 John Mark studies Math
12 John Gates studies Math
13 John Physics studies Math
```

A riga 1, il valore di default viene assegnato al parametro formale. Nel caso in cui si volesse modificare il terzo parametro lasciando il secondo di default, bisogna specificare i parametri nella chiamata alla funzione:

```
1  def printData(firstname, lastname = 'Mark', subject =
       'Math'):
2      print(firstname, lastname, 'studies', subject)
3
4  # Mixed passing is possible
5  printData(firstname = 'John', subject = 'Physics')
6  printData('John', subject = 'Physics')
7
8  # Output
9  John Mark studies Physics
10 John Mark studies Physics
```

6.5 Funzioni di ordine superiore

Le funzioni di ordine superiore sono funzioni che ricevono come argomento o restituiscono come risultato delle altre funzioni. Nei linguaggi funzionali queste tecniche sono piuttosto diffuse. Passare funzioni come argomenti ad altre funzioni è relativamente semplice da implementare, motivo per cui è presente anche nei linguaggi imperativi, in questo caso le funzioni sono considerate oggetti di secondo livello (normalmente sarebbero di terzo livello). Restituire funzioni come risultato di funzioni è invece più raro nei linguaggi imperativi, infatti solo in quelli più recenti è presente (Java, C#), dove le funzioni vengono considerate oggetti di primo livello.

6.5.1 Semantica

Nel caso più comune abbiamo il chiamante `main` che passa una funzione `f` che verrà valutata, eventualmente più volte, nel chiamato `g`. Il problema principale è sapere in quale ambiente viene valutata `f`. Normalmente si seguono le politiche di scope statico e dinamico, ma entrano in gioco anche le politiche di **deep** e **shallow binding**.
Esempio:

```
1   int x = 1;
2   int f(int y){return x + y;}
3   int g(int h (int i)){
4       int x = 2;
5       return h(3) + x;
6   }
7   ...
8   int x = 4;
9   int z = g(f);
```

dove la funzione `g` a riga 3 è una funzione di ordine superiore che ha come argomento una funzione che prende in input un intero e restituisce un intero. A riga 9 la funzione `f` viene data come argomento alla funzione `g`. Possiamo dire che la funzione `f` ha 3 momenti di vita:

- `f` viene dichiarata (riga 2), dove nel caso di scoping statico, la `x` fa riferimento alla dichiarazione in riga 1
- `f` viene passata come argomento alla funzione `g` (riga 9), dove la variabile non locale `x`, sempre nel caso di scoping statico, fa riferimento alla dichiarazione in riga 8
- `f` viene chiamata nel corpo di `g` (riga 5), dove `x` fa riferimento alla dichiarazione in riga 4

abbiamo dunque 3 alternative tra cui scegliere per decidere l'ambiente in cui valutare la funzione.

6.5.2 Politiche di binding

Le politiche di binding determinano quale ambiente non locale si applica al momento dell'esecuzione di `f`.

- scope statico: usa, come al solito, l'ambiente dove `f` è stata dichiarata (`x = 1`, riga 1)
- scope dinamico: ci sono due alternative:
  - **deep binding**: l'ambiente usato è quello in cui `f` viene passata come argomento, dunque quello a riga 9 con `x = 4`

– **shallow binding**: il più coerente con l'idea di scope dinamico, usa l'ambiente più vicino a dove viene chiamata, dunque dove a riga 5 `f` viene chiamata con il nome `h`, verrà usata la definizione `x = 2` a riga 4

6.5.3 Implementazione con scope statico

L'implementazione è molto simile al passaggio per nome: quando si passa l'argomento, si passa sia il codice della funzione tramite un puntatore, sia l'ambiente in cui valutarla. Si inserisce dunque nel RdA di `g`, in corrispondenza di `f`, la closure `<code, env>`. Alla chiamata della procedura `f` dentro `g`, si alloca il record di attivazione e si usa come puntatore alla catena statica il riferimento all'ambiente fornito dalla chiusura `env`.

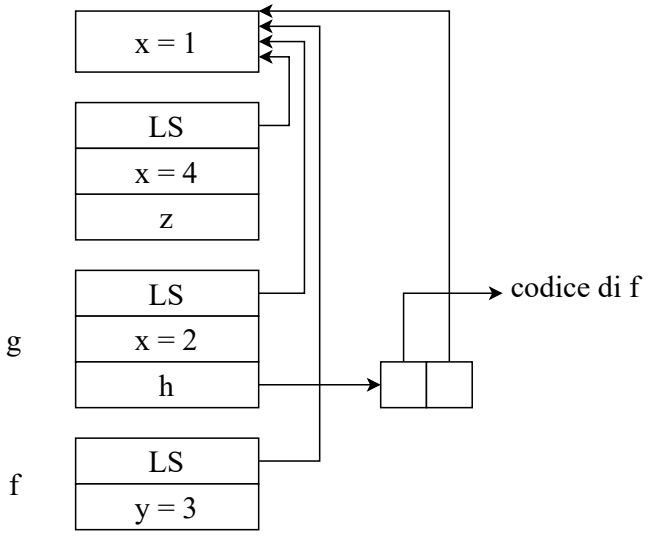


Figura 6.1: Stack di attivazione con chiusura e scope statico

6.5.4 Implementazione con scope dinamico: shallow binding

Non serve passare la chiusura perché la funzione verrà valutata nell'ambiente in cui viene utilizzata.

6.5.5 Implementazione con scope dinamico: deep binding

Si passa anche la chiusura in cui viene specificato l'ambiente dove la funzione è stata passata come argomento. Rispetto allo scope dinamico normale, non si risale lo stack e dunque parte di esso viene saltato, praticamente viene usato un link statico.

6.5.6 Deep vs shallow binding con scope statico

A prima vista deep o shallow binding non fa differenza con scope statico perché è la regola di scope statico a stabilire quale ambiente usare, ma in realtà non è così: potrebbero esistere più istanze del blocco che dichiara la procedura passata come parametro (la procedura viene dichiarata più volte), in questo caso la politica di binding cambia la dichiarazione da prendere in considerazione. Questo scenario accade in presenza di ricorsione. Per coerenza, viene sempre usato il deep binding implementato con chiusure, vediamo un esempio:

```
1   void foo(int f(), int x){
2       int fie(){
3           return x;
4       }
5       int z;
6       if(x == 0) z =f();
7       else foo(fie, 0);
8   }
9   int g(){
10      return 1;
11  }
12  ...
13  foo(g, 1);
```

dove `foo` è una funzione ricorsiva che definisce al suo interno la funzione `fie`, dunque ogni volta che `foo` viene chiamata, `fie` viene dichiarata di nuovo con un ambiente diverso, in questo caso con due `x` non locali diversi.

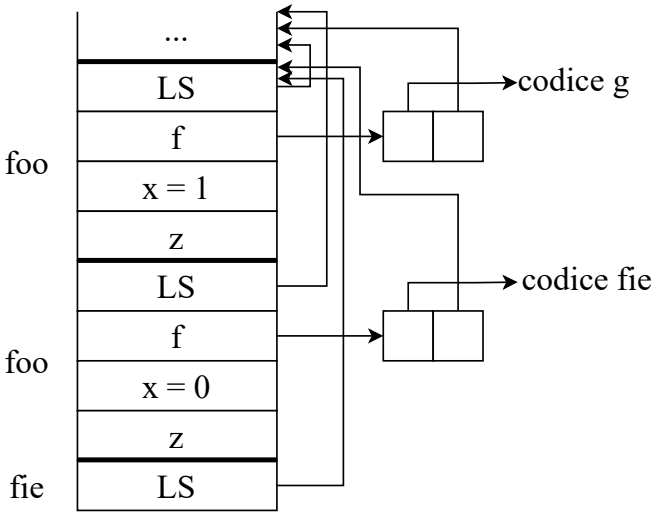


Figura 6.2: Stack di attivazione con deep binding

6.5.7 Funzioni come argomento in C

Le funzioni come argomento, come anticipato, possono essere presenti anche nei linguaggi imperativi, come nel seguente esempio in C:

```
1   void mapToInterval ( void (*f)(int), int max ) {
2       for ( int ctr = 0 ; ctr < max ; ctr++ ) {
3           (*f)(ctr);
4       }}
5
6   void print ( int x ) {
7       printf("%d\n", x); }
8
9   void main () {
10      ...
11      mapToInterval(print, 100)    /* notare print senza
            parentesi */
```

Per C la situazione è abbastanza semplice, un vincolo di tale linguaggio è che non è possibile definire funzioni all'interno di altre funzioni, dunque esistono solo due ambienti: l'ambiente globale e l'ambiente locale della funzione stessa. Quando bisogna usare una variabile all'interno di una funzione, se non appartiene all'ambiente locale allora per forza di cose deve essere valutata in quello globale, questo rende non necessaria la chiusura. Per utilizzare le funzioni come argomento basta utilizzare un puntatore alla funzione, la quale verrà valutata nell'ambiente globale.

6.5.8 Funzioni come risultato

Per funzioni come risultato si intende la possibilità di generare funzioni come risultato di altre funzioni, tali funzioni ritornate come risultato non potranno in generale essere rappresentate solamente dal loro codice ma è necessario anche l'ambiente nel quale la funzione verrà valutata.

Consideriamo un esempio semplice:

```
1   int x = 1;
2
3   void->int F() {
4       int g() {
5           return x + 1;
6       }
7       return g;
8   }
9
10  void->int gg = F();
11  int z = gg();
```

In questo caso `F` ha al suo interno una dichiarazione di un'altra funzione `g`, che viene ritornata come risultato. In seguito dichiariamo una variabile `gg` di tipo `void->int`, ovvero una funzione senza argomenti che ritorna un intero e gli assegniamo l'output di `F`, che non a caso è una funzione da `void` ad `int`. In questo modo possiamo usare `gg` come fosse una funzione, in questo caso `gg` è la funzione `g`. La procedura `F` ritorna una chiusura.

Una seconda versione del codice precedente risulta essere più complessa.

```
1  void -> int F () {
2      int x = 1;
3      int g () {
4          return x + 1;
5      }
6      return g;
7  }
8  void -> int gg = F();
9  int z = gg();
```

Per capire il motivo della maggiore complessità dobbiamo analizzare lo stack di attivazione della funzione.

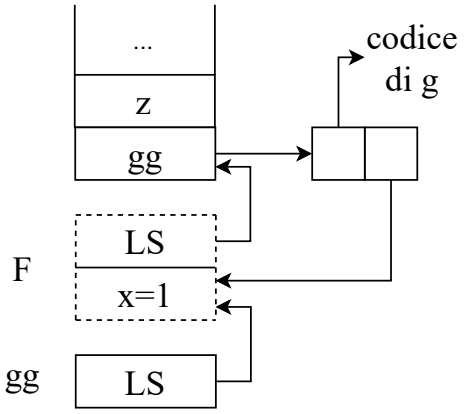


Figura 6.3: Stack di attivazione della funzione che ritorna un'altra funzione

Il programma principale contiene le dichiarazioni di `z` e di `gg`, alla quale deve essere assegnato un valore, in questo caso utilizzando una chiamata ad `F`. Tale chiamata porta alla creazione di un nuovo record di attivazione per `F`, il quale contiene una sola variabile locale `x`, inoltre la chiamata ad `F` porta alla creazione di `g` come chiusura. L'ambiente nel quale quest'ultima deve essere valutata è quello di `F`.

La funzione `gg` quindi farà riferimento all'ambiente di `F`, in questo caso allora non è possibile rilasciare il record di terminazione una volta terminata `F`.

Nei linguaggi che permettono funzioni come risultato si hanno alcune possibilità. Nei linguaggi funzionali si rinuncia completamente al meccanismo dello stack di attivazione, in questi linguaggi infatti ogni volta che viene chiamata una funzione il suo record di attivazione viene inserito in una struttura dati di tipo heap, alla chiusura di una funzione in questo modo la memoria utilizzata non viene comunque liberata evitando problemi di riferimenti pendenti. In questo caso però si crea un problema di occupazione di memoria dato che ad ogni chiamata di funzione viene allocato un record di attivazione nella heap che però non viene recuperato. Diventa necessario quindi un garbage collector a cui viene assegnato il compito di deallocare la memoria nella heap che non è più necessaria. Il garbage collector in generale può applicare due politiche, la prima è una gestione manuale dove viene espresso attraverso opportuni comandi quando un determinato elemento della heap non è più necessario e quindi eliminarlo. Tale approccio non è ottimale dato che elementi della heap possono essere condivisi da diversi elementi, nel caso di un puntatore, per esempio, potrebbe essere che l'elemento puntato sia puntato anche da altri puntatori e quindi una cancellazione del primo non sempre permette di cancellare anche l'elemento puntato. Nei linguaggi funzionali più sofisticati ci sono dei meccanismi automatici che analizzano la heap e cercano dei dati o record di attivazione non più accessibili e quindi che possano essere eliminati. Tali meccanismi sono complessi e abbastanza costosi.

Nel caso di linguaggi imperativi invece non si rinuncia allo stack di attivazione e si gestiscono i casi particolari che possano generare problemi con meccanismi appositi e più sofisticati.

6.6 Eccezioni

Le eccezioni sono un meccanismo usato nei linguaggi di programmazione per gestire errori o situazioni non previste, ma anche per terminare in anticipo la computazione perché si sa già il risultato. Quello che si vuole ottenere è di non far gestire al solito modo gli errori, ma passare il controllo a qualche parte del codice che si aveva previsto per quella determinata situazione.

I primi linguaggi gestivano le eccezioni con il `GOTO` ma, come abbiamo visto, ci sono alcuni esempi che evidenziano che il `GOTO` è meno potente rispetto a cicli e procedure. Con il `GOTO`, inoltre, è difficile implementare correttamente l'uscita da una procedura perché non si occupa di rimuovere i RdA delle funzioni invocate precedentemente all'eccezione. Per una gestione corretta si introduce un costrutto apposito.

6.6.1 Gestione delle eccezioni

Si definiscono blocchi protetti, ovvero parte del codice in cui si possono generare delle eccezioni. All'interno di questo blocco si definisce l'insieme delle

eccezioni che possono essere generate e per ciascuna eccezione il suo gestore, ovvero la parte di codice che deve essere eseguita se si verifica l'eccezione.

All'interno del blocco può essere sollevata un'eccezione (raise an exception), in tal caso la normale computazione viene interrotta e viene cercato il gestore dell'eccezione relativo all'eccezione sollevata.

I blocchi protetti possono essere innestati e all'interno del blocco possono essere chiamate delle funzioni, quindi può capitare che nel momento in cui viene sollevata un'eccezione e la si deve gestire, si debba uscire da tutte le funzioni che sono state chiamate. Da dentro a fuori, in ogni corpo delle funzioni viene cercato un blocco protetto e controllato se c'è il gestore relativo finché viene trovato, al limite si arriva al programma principale e, se ancora non viene trovato, si termina il programma mandando un messaggio di errore. Riassumendo, abbiamo tre costrutti:

- definizione delle eccezioni
- definizione di blocchi protetti, con relativi gestori delle eccezioni
- sollevamento dell'eccezione

Esempio in Java

- la funzione `average` calcola la media di un vettore
- se il vettore è vuoto, solleva l'eccezione
- le eccezioni sono sottoclassi di una classe `Throwable`
- bisogna dichiarare `throws` se una funzione può generare un'eccezione

```
1  class EmptyExcp extends Throwable {int x = 0;};
2
3  int average(int[] V) throws EmptyExcp(){
4      if(length(V) == 0) throw new EmptyExcp();
5      else {int s = 0; for(int i = 0; i < length(V); i++) s =
           s èV[i];}
6      return s / length(V);
7  };
8  ...
9  try{...
10     average(W);
11     ...
12 }
13 catch (EmptyExcp e) {write('Array vuoto');}
```

A riga 1 c'è la dichiarazione dell'eccezione personalizzata che, come detto, eredita da `Throwable`. Da riga 9 a riga 12 abbiamo il blocco protetto dentro il quale possono essere chiamate funzioni che sollevano eccezioni. A riga 13 viene specificato il gestore delle eccezioni, che è sempre legato al blocco di codice protetto. L'esecuzione del gestore rimpiazza la parte di blocco protetto che doveva ancora essere eseguita. Ai gestori volendo si possono passare dei parametri per esempio assegnando il parametro `x` in `EmptyExcp` (non mostrato nell'esempio). Se l'eccezione non è gestita nella procedura corrente:

- la routine termina, l'eccezione è risollevata al punto di chiamata
- se l'eccezione non è gestita dal chiamante, l'eccezione è propagata lungo la catena dinamica: si toglie un RdA e si passa il controllo al chiamante
- fino a quando si incontra un gestore o si raggiunge il top-level, che fornisce un gestore di default (ferma un programma con un messaggio di errore)

Di conseguenza non si può sapere a tempo di compilazione quale sarà il gestore di un'eccezione sollevata dentro una funzione appunto perché si segue la catena dinamica, che viene seguita anche se si usa lo scope statico.

6.6.2 Uso delle eccezioni per aumentare l'efficienza

Presenteremo un esempio di un programma ricorsivo in Scheme che moltiplica tutti i nodi di un albero binario in cui ogni nodo contiene un intero. Sappiamo che è sufficiente un singolo nodo con valore 0 per restituire 0 come risultato. Se volessimo fare questa cosa con i meccanismi standard sarebbe molto complicato a causa della presenza delle chiamate ricorsive che non permettono un'uscita anticipata. Sfruttiamo invece il meccanismo delle eccezioni per implementare questo controllo e per migliorare l'efficienza:

- definiamo una funzione ausiliaria di visita dell'albero che genera l'eccezione `found-zero` se trova uno 0
- la funzione principale chiama l'ausiliaria con un gestore dell'eccezione, se è stata generata l'eccezione `found-zero`, restituisce `0`, altrimenti il risultato della funzione ausiliaria

in questo modo evitiamo di dover continuare la visita quando si sa già il risultato finale. In Scheme non serve dichiarare le eccezioni prima di poterle usare.

Codice:

```
1   (define-struct node (left right))
2
```

```
3   (define tree (make-node (make-node 10 9)
4                           (make-node 0 (make-node 1 5))))
5
6   (define (fold-product-aux nd)
7       (cond
8       [(number? nd)
9           (if (equal? nd 0) (raise "zero-found") nd)]
10      [(node? nd)
11          (* (fold-product-aux (node-left nd))
12             (fold-product-aux (node-right nd)))]))
13
14  (define (fold-product nd)
15      (with-handlers
16          ([(lambda (e) (equal? e "zero-found"))
17                        (lambda (e) 0)])
18          (fold-product-aux nd)))
```

A riga 1 si definisce una struttura dati `node` che ha una componente `left` e una `right`. A riga 3 si definisce l'albero usando la funzione `make-node` che viene generata in automatico dopo aver definito `node` e che prende in input due interi, può essere chiamata ricorsivamente. Definiamo la funzione `fold-product` che prende un albero e calcola il prodotto di tutti i nodi. A riga 6 definiamo una funzione ausiliaria `fold-product-aux)` che controlla se un nodo ha valore zero e in quel caso genera un'eccezione. A riga 14 definiamo la funzione principale che chiama la funzione ausiliaria in un blocco protetto `with-handlers` in cui il primo argomento è un controllo che l'eccezione generata `(e)` sia quella cercata, e poi c'è il gestore dell'eccezione che in quel caso restituisce `0`; il secondo argomento è la chiamata alla funzione ausiliaria che può generare l'eccezione.

6.6.3 Implementazione delle eccezioni

I blocchi protetti sono semplicemente dei blocchi che vengono valutati in un ambiente in cui sono definiti i gestori delle eccezioni. Se si seguono le normali regole di implementazione dei blocchi, all'inizio di un blocco protetto si alloca un nuovo RdA sullo stack di attivazione che contiene la lista dei gestori, e ogni volta che si esce bisogna rimuoverlo. Quando un'eccezione viene sollevata, si cerca il gestore nel RdA corrente, se non lo si trova si scende nella lista dei RdA alla ricerca di esso. Il problema di questa implementazione è che ogni volta che si esce o entra da un blocco protetto bisogna modificare lo stack di attivazione, operazione che è inefficiente se nella maggior parte dei casi l'eccezione non si verifica. Tipicamente la soluzione che viene implementata è quella di non creare RdA con i blocchi anonimi, ma solo quando c'è una chiamata di procedura, ovvero inserire tutti i gestori delle eccezioni dichiarati

in blocchi anonimi nel corpo della procedura all'interno del suo RdA. Inoltre si costruisce una mappa dei blocchi protetti che contiene tutti i gestori delle eccezioni, dove trovarli e le eccezioni che gestisce, in modo da eseguire un accesso diretto al gestore per passare il controllo.

Capitolo 7

Strutture dati

In questo capitolo analizzeremo il sistema di tipi nei linguaggi di programmazione.

7.1 I tipi

Un tipo di dato è definito come una collezione di valori omogenei ed effettivamente presentati, dotata di un insieme di operazioni che manipolino tali valori.

Analizziamo il significato della definizione: si tratta quindi di un insieme di valori, come i numeri naturali, che possiamo rappresentare nel calcolatore, questo non è possibile per esempio per i numeri reali, non rappresentabili a causa dell'esistenza di alcuni numeri appartenenti a tale insieme infiniti e non rappresentabili attraverso nessun algoritmo. Inoltre un elemento essenziale è la presenza di operazioni che manipolino tali elementi.

I tipi possono essere visti a diversi livelli:

- livello di progetto: servono a dare un'organizzazione concettuale dei dati, i quali vengono divisi in diverse categorie alle quali viene associato un tipo
- livello di programma: identificano e prevengono errori, permettono di verificare se un dato è usato coerentemente al loro tipo
- livello di implementazione: determinano l'allocazione dei dati in memoria

Ogni linguaggio di programmazione ha un proprio sistema di tipi, ossia il complesso delle informazioni e delle regole che governano i tipi in quel linguaggio. Più precisamente, un sistema di tipi è costituito da:

1. l'insieme dei tipi predefiniti dal linguaggio
2. i meccanismi che permettono di definire nuovi tipi

3. i meccanismi relativi al controllo dei tipi, tra i quali distingueremo:
    - le regole di equivalenza, che specificano quando due tipi formalmente diversi corrispondono allo stesso tipo (equivalenza tra tipo della variabile ed espressione assegnata)
    - le regole di compatibilità, che specificano quando un valore di un certo tipo può essere utilizzato in un contesto nel quale sarebbe richiesto un tipo diverso
    - le regole e le tecniche di inferenza dei tipi, che specificano come il linguaggio attribuisce un tipo ad un'espressione complessa, partendo dalle informazioni delle sue componenti

7.2 Categorie di sistemi di tipo

I sistemi di tipo si suddividono in più categorie.

7.2.1 Statici e dinamici

In base a quando avviene il controllo di tipo (**type checking**):

- **statico**: a tempo di compilazione come in C, Java. Si dichiara sempre il tipo delle variabili e il compilatore si occupa di controllare che vengano usate coerentemente con il loro tipo
- **dinamico**: durante l'esecuzione del codice come in Python, JavaScript e Scheme. Nel codice non si fanno dichiarazioni di tipo, ma i controlli vengono fatti a tempo di esecuzione verificando le compatibilità.

Questa separazione non è netta, quasi tutti i linguaggi fanno dei controlli dinamici e alcuni controlli di tipo (dimensione degli array) possono essere fatti solo a tempo di esecuzione.

Vantaggi e svantaggi

Sistemi di tipo statici:

- il controllo viene fatto prima e quindi vengono anticipati gli errori
- c'è meno carico di lavoro perché non bisogna effettuare controlli durante l'esecuzione
- a volte sono più prolissi perché ad ogni dichiarazione di variabile bisogna definirne il tipo
- tendono ad essere più restrittivi rispetto ai dinamici, perché per prevenire possibili errori di tipo, si impedisce codice perfettamente lecito nella pratica, esempio:

```
1  (define (f g) (cons (g 7) (g #t)))
2  (define pair_of_pairs (f (lambda (x) (cons x x))))
```

dove si applica la funzione `g` prima a `7`, poi a `true`, quindi dal punto di vista del compilatore la funzione dovrebbe poter prendere in input entrambi i tipi, ma non ci sono funzioni con questo grado di polimorfismo e quindi viene rifiutata. Il codice però funziona comunque perché la funzione lambda semplicemente prende i dati e ci costruisce un coppia, cosa sempre possibile. Questo codice verrebbe rifiutato da un controllo statico, ma uno dinamico lo accetterebbe.

Sistemi di tipo dinamici:

- bisogna eseguire il codice per trovare l'errore
- i dati occupano più memoria perché oltre al dato bisogna avere informazioni rispetto al tipo e bisogna fare i controlli a tempo di esecuzione (test non pesanti, ottimizzazione possibile)
- più concisi (meno definizioni di tipo nel codice)
- più flessibili

7.2.2 Strong e weak

- **Strong type system**: è difficile che un errore di tipo non venga rilevato nel codice a tempo di compilazione oppure durante l'esecuzione (**type safe**)
- **Weak type system**: permettono una maggiore flessibilità a costo della sicurezza, si possono scrivere operazioni che dal punto di vista dei tipi non hanno alcun senso e dovrebbero generare un errore, ma il programma va avanti lo stesso senza avvisare e magari si pianta in un altro momento. Esempio: usare una sequenza di 4 caratteri come se fosse un numero intero.

7.2.3 Concetti indipendenti

È importante sottolineare che la staticità/dinamicità e strong/weak sono due dimensioni ortogonali attraverso le quali si possono catalogare i vari linguaggi di programmazione. Queste dimensioni non sono assolute, cioè un linguaggio di programmazione può essere più o meno statico o più o meno dinamico e più o meno strong/weak.

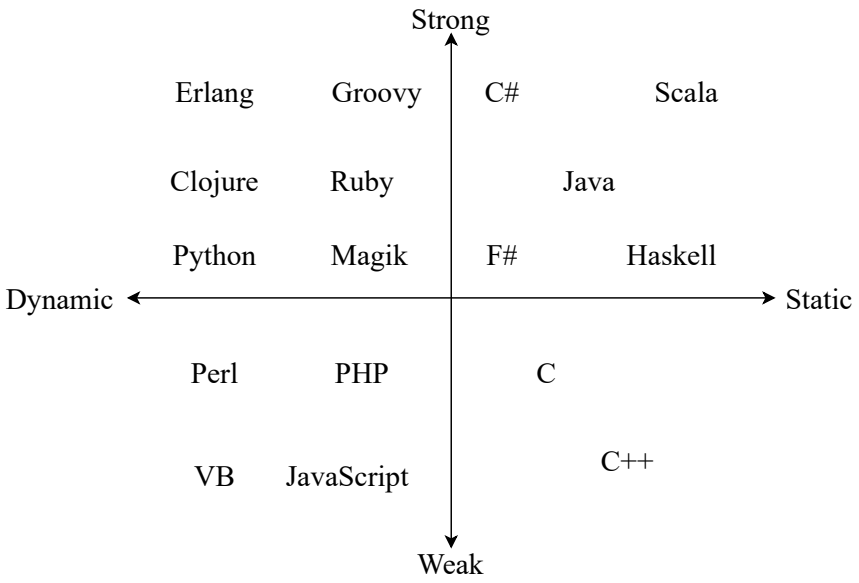


Figura 7.1: Categorizzazione di alcuni linguaggi di programmazione

7.3 Catalogazione dei tipi e dei loro valori

Fissato un linguaggio di programmazione, possiamo classificare i suoi tipi a seconda di come i suoi valori possono venir manipolati e del genere di entità sintattiche che corrispondono a tali valori. Seguendo la classificazione che abbiamo già visto nel sottoparagrafo 5.2.4 a pagina 105, abbiamo valori:

- denotabili, se possono essere associati ad un nome (ovvero rappresentabili mediante identificatori)
- esprimibili, se possono essere il risultato di un'espressione complessa (ovvero rappresentabili mediante espressioni (diverse da identificatori e dichiarazioni))
- memorizzabili, se possono essere memorizzati in una variabile (ovvero salvati in memoria, assegnati ad una variabile)

Dato un linguaggio di programmazione e dato un tipo, il tipo può essere catalogabile nelle tre categorie sopra. Prendiamo per esempio le funzioni, che sono un tipo perché possono per esempio essere `int -> int`, normalmente sono denotabili perché avranno un nome che può essergli associato. Sono esprimibili perché, quando sono oggetti di primo livello, possiamo avere un'espressione il cui risultato diventa una funzione, normalmente con una lambda espressione. Sono memorizzabili se c'è la possibilità di avere delle locazioni di memoria dove depositare una funzione e da dove, volendo, si può modificare.

7.3.1 Tipi predefiniti o scalari

Vengono chiamati scalari perché i valori possono essere messi in ordine (in scala) e sono:

Booleani

Sono scalari perché false è più "piccolo" rispetto a true.

- valori: `true`, `false` (Java), (Scheme: `#t`, `#f`)
- operazioni sul tipo: or, and, not, condizionali
- rappresentazione a basso livello: un byte
- note: C non ha un tipo esplicito bool, ma usa gli interi (0 per false, qualsiasi altro valore per true)

Caratteri

- valori: `a, A, b, B, ..., è, é, ..., ;, ', ...`
- operazioni sul tipo: uguaglianza; code/decode (conversione da numero a carattere e viceversa); qualche modo per passare da un carattere al successivo e/o al precedente (nella codifica fissata); dipendenti dal linguaggio e dalla codifica che usa
- rappresentazione a basso livello: un byte (ASCII - C), due byte (UNICODE - Java), lunghezza variabile (UTF-8 - opzione in Java)

Interi

- valori: `0, 1, -1, 2, -2, ..., maxint`
- operazioni sul tipo: confronti, `+, -, *, mod, div, ...`, più un buon numero di funzioni definite in varie librerie
- rappresentazione a basso livello: alcuni byte (1, 2, 4, 8, 16) scritti in complemento a due
  - in alcuni linguaggi esistono più tipi di interi che si differenziano per il numero di byte della rappresentazione (`byte` (1) o `char` (1), `short` (2), `int` (4), `long` (4), `long long` (16)). Specificare la lunghezza porta ad un comportamento del programma più uniforme, perché se non la si specificasse, compilatori diversi potrebbero usare rappresentazioni diverse degli interi e alcune potrebbero portare ad overflow ed altre no
  - C contiene anche la versione unsigned: binario puro
  - Scheme e altri linguaggi dispongono di interi di dimensione illimitata

Floating point - numeri reali

- valori: numeri razionali in un certo intervallo
- operazioni sul tipo: `+, -, *, /, ...`, più un buon numero di funzioni definite in varie librerie
- rappresentazione a basso livello: alcuni byte (4, 8, 16) mantissa + esponente, notazione che ha il vantaggio di poter rappresentare più numeri a parità di bit; (`float`, `double`, `Quad` o `float128`), notazione standard IEEE-754, virgola mobile

Alcuni linguaggi non specificano la lunghezza della rappresentazione usata per interi e floating-point, questo può creare problemi nella portabilità tra compilatori diversi.

L'hardware mette a disposizione delle primitive per le operazioni aritmetiche (per interi e floating point), mentre altri tipi non sono supportati.

Tipi numerici

In alcuni linguaggi di programmazione, oltre ai tipi classici, se ne aggiungono altri, che però non sono supportati dall'hardware:

- numeri complessi: vengono rappresentati come una coppa di numeri floating point, presenti in Scheme, Ada, e altri linguaggi in cui vengono definiti tramite librerie
- numeri fixed point: numeri con la virgola in una certa posizione, sono razionali, ma non con la notazione esponenziale. Vengono rappresentati con alcuni byte (2 o 4) in complemento a due o BCD (Binary Code Decimal), virgola fissa. Presenti in Ada
- razionali: rappresentazione esatta dei numeri frazionari. Vengono rappresentati come coppie di interi numeratore-denominatore. Con questa rappresentazione si possono eseguire le operazioni in aritmetica esatta evitando gli errori di arrotondamento. Presenti in Scheme

Esempio: tipi numerici in Scheme

Scheme considera 5 insiemi di numeri

- `number`
- `complex`
- `real`
- `rational`

- `integer`

ciascuno sovrainsieme dei sottostanti. La rappresentazione interna è nascosta (non è un linguaggio tipato), lasciata all'implementazione. Per gli interi normalmente viene usata una rappresentazione a lunghezza arbitraria. Si distingue tra `exact` (`integer, rational`) e `inexact` (`real, complex`).

Void

Il tipo `void` è stato introdotto in C

- valori: uno solo (`null`)
- operazioni sul tipo: nessuna operazione
- rappresentazione a basso livello: nessun bit

permette di trattare procedure/comandi come caso particolare di funzioni/espressioni in cui il risultato restituito è ininfluente.

```
void f(...) {...}
```

la procedura `f` vista come una funzione deve restituire un valore (e non nessuno), in questo modo restituisce un valore di tipo `void`. Siccome il valore restituito da `f`, di tipo `void`, è sempre lo stesso, non è possibile e non serve specificarlo nell'istruzione `return`.

7.3.2 Tipi ordinali (o discreti)

Chiamiamo ordinali quei tipi tali che datone uno esiste un valore successivo ed un precedente, fatta eccezione per il primo e l'ultimo, non sono ordinali quei numeri come i floating point anche se a causa della rappresentazione fisica è possibile trovare un successore. Alcuni ordinali predefiniti sono:

- interi
- caratteri
- booleani

Ha senso fare una distinzione dato che con i numeri ordinali è possibile eseguire un'iterazione, per esempio per un ciclo `for` generalmente vengono usati gli interi, ma è possibile usare i caratteri, inoltre gli ordinali possono essere usati anche come indici di array (anche se tale possibilità dipende dal linguaggio).

Esistono due tipi particolari di ordinali che possono essere definiti dagli utenti: le enumerazioni e gli intervalli (subrange).

Enumerazioni

L'idea delle enumerazioni consiste nel definire un tipo elencando tutti i suoi possibili valori.

```
type Giorni = (Lun,Mar,Mer,Giov,Ven,Sab,Dom);
```

Dall'esempio si nota anche la migliore leggibilità del codice, i valori di giorno rendono immediato capire di cosa si stia parlando rispetto che associare per ciascuno un numero e utilizzare quello. I valori sono anche ordinati in base all'ordine in cui sono definiti (`Mar < Ven`) ed è possibile eseguire un'iterazione sui valori, come per esempio: `for i:= Lun to Sab do`.

Sono definite le operazioni `succ`, `pred` e di confronto. Sono rappresentati con un byte.

Sono presenti sia in C sia in Java, anche se in C la situazione è diversa. Data la seguente enumerazione:

```
enum giorni = {Lun,Mar,Mer,Giov,Ven,Sab,Dom};
```

In C è equivalente a scrivere:

```
1 typedef int giorni;
2 const giorni Lun=0, Mar=1, Mer=2, ...,Dom=6;
```

In Java scrivere 1 e Mar non è la stessa cosa dato che sono due tipi distinti, Mar è di tipo `giorni` e 1 di tipo intero, in C questo non è il caso e scrivere Mar e 1 ha lo stesso effetto dato che sono entrambi interi.

Intervalli

Sono stati introdotti in Pascal e non sono presenti in C, Scheme e Java (in realtà in quest'ultimo sono possibili da definire come sottoggetti). Consistono nel definire un sotto intervallo di un ordinale base già definito:

```
1 type MenoDiDieci = 0..9;
2 type GiorniLav = Lun..Ven;
```

La rappresentazione è la stessa del tipo base da cui sono definiti.

In genere sono utili per avere una documentazione "controllabile", ovvero è possibile eseguire un type checking dinamico, oppure in alcuni casi per risparmiare memoria.

7.3.3 Tipi composti, strutturati, non scalari

Il più comune metodo di costruzione di tipi composti è il **record** (o struct), presente in quasi tutti i linguaggi, il quale consiste in una collezione di campi con un nome, ciascuno con un tipo (non per forza uguale).

Un altro tipo sono i **record varianti** (o union), record dove solo alcuni campi (mutuamente esclusivi) sono attivi in un dato istante.

Un altro costrutto sono gli **array**, nei quali tutti gli elementi sono dello stesso tipo e sono identificati da degli indici.

Altri tipi composti sono:

- insiemi: sottoinsieme di un tipo base
- puntatore: riferimento ad un oggetto di un altro tipo
- funzioni, procedure, metodi, oggetti

Tipi record e struct

Permettono di raggruppare dati di tipo eterogeneo, sono presenti in: C, C++, CommonLisp, Algol68. In Java non esistono dei record veri e propri ma possono essere rappresentati usando le classi, un record in Java è una classe senza metodi.

Esempio in C:

```
1  struct studente{
2      char nome[20];
3      int matricola;
4  };
```

Per selezionare i campi:

```
1  studente s;
2  s.nome = "Mario"
3  s.matricola=343536;
```

É possibile creare dei record annidati. In Java la situazione è simile:

```
1  class Studente {
2      public String nome;
3      public int matricola;
4  }
5
6  Studente s = new Studente();
7  s.nome = "Mario";
8  s.matricola = 343536;
```

Esiste una differenza tra i due, in C la notazione "`studente s`" fa si che `s` sia una variabile a cui è associata la locazione di memoria contenente i dati di `studente`, in Java con la stessa notazione viene allocato lo spazio per il riferimento `s`, ma non per l'oggetto vero e proprio a cui `s` fa riferimento, per cui è necessario un passaggio in più utilizzando la funzione "`new Studente()`" che crea il nuovo oggetto nell'heap e poi associarlo a `s`.

I record sono memorizzabili, denotabili ma non sempre esprimibili, dato che non è possibile (spesso) scrivere delle espressioni complesse il cui valore è un record. Nel caso di C è possibile a livello di inizializzazione.

L'uguaglianza non è in genere definita, in Java verificare un'uguaglianza significa controllare che due variabili puntino ad esattamente la stessa struttura.

Anche in Scheme è possibile implementare i record in maniera indiretta, definendo delle funzioni che:

- costruiscono record
- accedono ai campi
- testano il tipo di record

Prendiamo come esempio il record libro composto dai campi nome e autori.

```
1  (define (book title authors) (list 'book title author))
       //'book perchè book èun identificatore
2  (define (book-title b) (car (cdr b)))
3  (define (book-? b) (eq? (car b) 'book))
```

La prima funzione dato un titolo ed un autore ritorna una lista con tre elementi: book, il titolo e l'autore.

Con la seconda ritorniamo il secondo elemento della lista, se chiamata su un tipo libro allora ritornerà il titolo dello stesso.

Con la terza implementiamo un controllo per sapere se si tratta di un libro semplicemente controllando se il primo elemento è 'book. Le funzioni definite poi possono essere usate per definire dei libri specifici:

```
1  (define bazaar
2      (book
3          "The Cathedral and the Bazaar"
4          "Eric S. Raymond" ))
5
6  (define titolo_bazar (book-title bazaar)
7
8  (book-? bazar)
```

In racket (MIT-Scheme) è possibile ottenere lo stesso risultato usando define-struct:

```
1  (define-structure book title authors)
2
3  (define bazaar
4      (make-book
```

```
5           "The Cathedral and the Bazaar"
6           "Eric S. Raymond" ))
7
8   (define titolo_bazar (book-title bazaar)
9
10  (book? bazar)
```

Con questa notazione book rappresenta la struttura e tale nome verrà utilizzato automaticamente per creare le funzioni costruttori e distruttori: nel caso di `book` verranno create le funzioni `make-book` e `book?`. In racket si può ottenere lo stesso risultato usando `(struct ...)`.

In memoria i record vengono rappresentati allocando un numero sufficiente di byte per rappresentare ciascun campo, esiste però un problema dovuto all'allineamento dei vari campi. La rappresentazione in memoria è composta da parole, le quali sono a loro volta composte da n byte consecutivi (in genere 4 o 8), inoltre è richiesto che il primo byte di queste sia in una posizione multipla di 4 (allineamento), i trasferimenti di byte non allineati non sono possibili. Un esempio di un eventuale problema è il caso di un record in cui i campi sono dei caratteri, in questo caso la soluzione più immediata è utilizzare una parola per memorizzare ciascun campo (normalmente sarebbe un byte), soluzione non ottimale però dato che porta ad uno spreco di spazio. Un'altra soluzione è usare un "packed record", il quale consiste nel rappresentare con un'unica parola tutti e quattro i byte, evitando lo spreco di spazio ma rendendo l'accesso più complesso in assembly. In alcuni casi è possibile ordinare i campi sia per mantenere le parole allineate sia per evitare lo spreco di spazio.

Tipi Union

I tipi unione sono la controparte in C dei record varianti. Un'unione è analoga ad un record (struct) nella definizione e nella selezione dei campi, ma con la fondamentale differenza che solo uno dei campi di un'unione può essere attivo in un qualsiasi momento, visto che i campi (che possono essere di tipo diverso) condividono la rappresentazione in memoria. Esempio:

```
1   union Data{
2       int i;
3       float f;
4       char str[20];
5   } data;
6   float y;
7   ...
8   data.str = "abcd";
```

Particolare necessario solo in C, a riga 5 viene definito il campo `data`, di tipo unione, che serve per poter accedere ai veri e propri varianti. L'oggetto `data` contiene o un intero o un floating point o una stringa.

Record varianti

I record varianti sono una forma particolare di record in cui alcuni campi sono fissi, mentre altri sono mutuamente esclusivi. Esempio:

```
1   type studente = record
2       nome : packed array [1..6] of char;
3       matricola : integer;
4       case fuoricorso : Boolean of
5           true: (ultimoanno: 2000..maxint); // ultimo anno in
                cui era in corso
6           false: (anno : (primo, secondo, terzo); // anno
                frequentato
7                   inpari : Boolean; // in pari con gli esami
8                   )
9       end;
10
11  var s : studente;
12  s.fuoricorso := true;
13  s.ultimoanno := 2001;
```

Uno studente, in base a se è fuori corso o meno, ha dei campi diversi che lo descrivono. I primi due campi sono fissi, mentre il terzo campo, `fuoricorso`, preceduto dalla parola riservata `case` a riga 4, è il **tag**, o discriminante del record variante. Un tag `true` indica che il record ha un ulteriore campo: `ultimoanno` di tipo intervallo. Un tag `false` indica invece che il record ha due ulteriori campi: `anno` di tipo enumerazione e `inpari` di tipo booleano. Le due possibilità che compaiono tra parentesi tonde sono le varianti del record. In generale, il tag può essere di un qualsiasi tipo ordinale e può dunque essere seguito da un numero di varianti pari alla cardinalità di quel tipo. Da un punto di vista semantico, in un record variante solo una delle due varianti è significativa, mai entrambe. Quale delle due varianti sia significativa è indicato dal valore del tag; al tag e alle varianti si può accedere come ad un qualsiasi altro campo, come è stato fatto in riga 12.

Le due varianti `ultimoanno` e `anno` possono condividere la stessa locazione di memoria. Per memorizzare i record varianti in memoria si avranno tanti campi quanti sono i tipi fissi, e poi si avrà lo spazio maggiore che viene occupato da una delle varianti.

È possibile simulare i record varianti in C usando una combinazione di union e struct:

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

Nel codice esempio vediamo come ci sono i primi due campi che sono fissi, poi c'è il tag `fuoricorso` che però è completamente slegato dall'unione, il che significa che è compito del programmatore metterlo in relazione al significato dei campi dell'unione che lo segue, e infine il quarto campo che è una union che contiene la parte variabile: possiamo avere un singolo campo `ultimoanno` oppure due campi `anno` e `inpari` che però vanno messi insieme all'interno di una struttura. Questo rende più complicato accedere ai campi, per esempio per accedere a `inpari`, bisogna scrivere

```
s.campivarianti.studInCorso.inpari
```

invece del semplice `s.inpari` dei record varianti classici.

Union, Variant e type safety

I tipi unione permettono facilmente di aggirare i controlli di tipo.

```
1  union Data{
2      int i;
3      float f;
4      char str[20];
5  } data;
6  float y;
7  ...
8  data.str = "abcd";
9  y = data.f;
```

A riga 9 vediamo come la codifica ASCII di `"abcd"` viene trattata come un numero reale senza alcun avviso da parte del sistema. Lo stesso problema è presente anche nei tipi variant.

I record variant/union come abbiamo visto agevolano la scrittura del codice di controllo, ma peccano dal punto di vista della type safety e quindi

sono molto poco usati, per esempio Java non li usa. Le uniche motivazioni che possono portare all'uso di questi costrutti sono:

- risparmio di spazio, che però attualmente è meno importante
- alcuni tipi sono naturalmente descritti da tipi unione, per esempio:
  - lista: può essere vuota oppure un elemento seguito da una lista
  - albero binario: può essere una foglia o un nodo con sottoalbero sinistro e sottoalbero destro

In alcuni linguaggi come Algol68, Haskell e ML, esistono delle versioni sicure dei tipi union, ovvero si possono creare dei costrutti che hanno lo stesso comportamento. Nell'esempio del record variant `data`, non si può accedere direttamente ai singoli campi, ma bisogna definire un codice per ogni variante tramite un particolare costrutto `case`.

```
1  case data in
2  (int i) : a = i + 1
3  (float j) : x = sqrt(j)
```

Struct (e quasi Union) in Java con le sottoclassi

In Java non esiste il tipo struct, ma cerca di imitare C per gran parte della sintassi. Usando oggetti con campi pubblici e senza metodi si ottiene una struttura molto simile a una struct. Per quanto riguarda invece i record variant si può usare il meccanismo delle sottoclassi e dell'ereditarietà, per cui abbiamo una classe genitore che contiene i campi fissi comuni a tutte le varianti, e poi creiamo una sottoclasse per ogni variante con gli appositi campi che eredita dalla classe genitore.

```
1  enum AnnoCorso {PRIMO, SECONDO, TERZO};
2
3  class Studente{
4      public String nome;
5      public int matricola;
6  }
7
8  class StudenteInCorso extends Student{
9      public AnnoCorso anno;
10 }
11
12 class StudenteFuoriCorso extends Student{
13     public int ultimoAnno
14 }
```

```
15    ...
16    StudenteInCorso mario = new StudenteInCorso();
17    mario.nome = "mario";
18    mario.matricola = 456;
19    mario.anno = AnnoCorso.TERZO;
```

Rimane però una differenza con i record variant che non è possibile implementare in questo modo, ovvero che non si può cambiare la sottoclasse di un oggetto, per esempio trasformare `mario` in un `StudenteFuoriCorso`, ma bisogna creare una nuova istanza copiando i campi necessari.

Array

Gli **array** sono una collezione di dati omogenei, identificati da un indice di tipo ordinale (generalmente un intero). Gli array possono essere visti come una funzione da tipo indice a tipo degli elementi, rappresentata in modo estensionale, il tipo elemento può essere qualsiasi tipo memorizzabile e raramente un tipo funzionale. Alcuni esempi di dichiarazione:

```
1    int vet[30];    // tipo indice: tra 0 e 29, C
2    int[] vet;      // Java, notazione preferita
3    int vet[];      // Java, notazione alternativa
4    var vet : array [0..29] of integer; // Pascal
```

Array multidimensionali

Gli array multidimensionali, detti anche matrici, possono essere definiti in due diversi modi:

- array con più indici (definizione matematica)
- array con elementi altri array (definizione usata in C)

In Pascal le due definizioni sono equivalenti:

```
1    var mat : array [0..29,'a'..'z'] of real;
2    var mat : array [0..29] of array ['a'..'z'] of real;
```

Nella prima definizione `mat` viene definita come un array con più indici, il primo di tipo intero, il secondo di tipo carattere, ciascuna coppia di indici identifica una posizione contenente un tipo reale. La seconda definizione è diversa ma equivalente, inizialmente si dichiara un array con un unico indice (intero) e nel quale ogni posizione contiene un tipo array di indice carattere. In C e in java non è possibile dichiarare una matrice come un array con più indici, è possibile però usando array di array.

```
int mat[30][26] // C
int[][] mat     // Java
```

La principale operazione permessa è la **selezione**, sia `vet` una array ed `i` un indice è possibile ottenere l'elemento di `vet` in posizione `i` con la scrittura `vett[i]`, tale elemento può essere letto o modificato.

Alcuni linguaggi permettono lo **slicing**, che consiste nel selezionare una parte contigua di un array, come nel seguente esempio in Ada:

```
mat : array(1..10) of array (1..10) of real;

mat(3) // èla terza riga della matrice quadrata mat
```

In Python è possibile definire un sottoinsieme di un array già definito.

```
y=(1,2,3,4)
x = y[:-1]
```

In alcuni linguaggi è possibile il calcolo vettoriale, di solito presente in linguaggi adatti a calcolo scientifico, crittografia e grafica, sono presenti per esempio in Fortran90: `A + B` somma gli elementi di `A` e `B` dello stesso tipo. Sono presenti anche in Assembly, utile in alcuni ambiti dove la frequenza delle operazioni giustifica il supporto hardware.

Gli array vengono memorizzati attraverso locazioni contigue di memoria, dato un indice e la posizione iniziale è possibile attraverso opportuni conti (a seconda che le parole siano allineate o no) risalire alla posizione dell'elemento cercato. La memorizzazione degli array multi dimensionali avviene ripetendo più volte il passaggio eseguito per i vettori monodimensionali, si memorizza ciascuno degli array che compongono la matrice in maniera sequenziale. Sono possibili diversi metodi con cui eseguire questa memorizzazione, se prendiamo il caso di array bidimensionali abbiamo:

- ordine di riga: le righe, sub-array di un array, memorizzate in locazioni contigue, è il maggiormente usato.

  Es: `V[0,0] V[0,1]...V[0,9] V[1,0]...`

- ordine di colonna: le colonne, sub-array di un array, memorizzate in locazioni contigue.

  Es: `V[0,0] V[1,0] V[2,0]...V[13,0] V[0,1]...`

L'ordine utilizzato è importante per l'efficienza dei sistemi con cache e per gli algoritmi di scansione del vettore, se eseguiamo una scansione per colonne di un vettore memorizzato per righe dobbiamo esaminare punti distanti in memoria e verranno generati più cache miss[1].

[1] Il valore non è presente in cache e bisogna prelevarlo dalla memoria principale

La locazione di memoria in cui viene salvato un elemento va implementata manualmente in assembly, nei linguaggi di alto livello questo non è necessario.

Facciamo un esempio di calcolo per una matrice a tre dimensioni A, calcolando la locazione di $A[i, j, k]$ dove i è il piano, j è la riga, k la colonna e ogni elemento è di tipo `elemType` e $0 \leq i < l_1$, $0 \leq j < l_2$ e $0 \leq k < l_3$, con l_i indice massimo di tale dimensione:

- $S_3 = dim(\texttt{elemType})$ dimensione di un elemento `elemType`
- $S_2 = l_3 * S_3$ dimensione di una riga
- $S_1 = l_2 * S_2$ dimensione di un piano

Possiamo calcolare la locazione di $A[i, j, k]$ nel seguente modo:

$$A[i, j, k] = a + i * S1 + j * S2 + k * S3$$

dove:

- $a =$ indirizzo di inizio di A
- $a + i * S_1 =$ indirizzo di inizio del piano di $A[i, j, k]$, sotto-matrice $A[i]$
- $a + i * S_1 + j * S_2 =$ indirizzo di inizio della riga di $A[i, j, k]$, sotto-vettore $A[i, j]$
- $a + i * S_1 + j * S_2 + k * S_3 =$ indirizzo di $A[i, j, k]$

Quando si memorizza un array bisogna tenere in considerazione la **shape**, definita come il numero di dimensioni e il range di ciascuna di esse. La shape può essere determinata in vari istanti:

- **shape statica**: si tratta della forma più semplice, nella quale la shape è già definita al momento della compilazione, in questo caso è anche possibile precalcolare le posizioni.
- **shape fissata al momento dell'esecuzione della dichiarazione**: chiamata della procedura e definizione delle variabili locali (per vettori locali a una procedura)
- **shape dinamica**: le dimensioni variano durante l'esecuzione (Python, JavaScript)

In base a come viene determinata la shape cambia anche il metodo di accesso al vettore stesso, mentre con shape statica l'informazione sulla forma dell'array è mantenuta dal compilatore, nel caso che non lo sia questa viene contenuta in un descrittore dell'array detto **dope vector**. Il dope vector viene memorizzato nella parte fissa del RdA e contiene:

- puntatore all'inizio dell'array (nella parte variabile)

- numero dimensioni
- limite inferiore (se a run-time vogliamo controllare anche l'out-of-bound viene salvato anche limite superiore)
- occupazione per ogni dimensione (valore S_i)

Per accedere al vettore si calcola la posizione run-time basandosi sul dope-vector. Esempio: `A : array[l1..u1, l2..u2, l3..u3] of elemType`

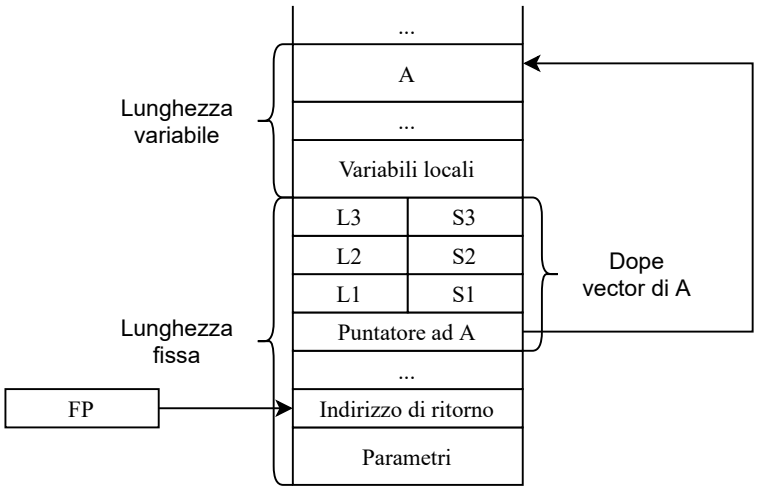


Figura 7.2: RdA con dope vector

In generale, se la shape è statica, l'array viene memorizzato nel RdA, in questo caso il dope vector perde anche la sua utilità siccome le informazioni sul vettore restano costanti per tutta la durata del programma. Nel caso la shape sia fissata al momento della dichiarazione si memorizza il dope vector nella parte iniziale del RdA e il vettore vero e proprio nella parte finale, se la forma è dinamica invece si memorizza tutto nella heap. In Java:

- gli array sono oggetti memorizzati nella heap
- la definizione di una variabile vettore non alloca spazio
- spazio allocato con una chiamata `new`
- array multidimensionali: memorizzati come vettori di puntatori.

```
int[] vettore = new int[4];
```

Insiemi

Gli insiemi sono disponibili in alcuni linguaggi di programmazione, dove non sono presenti di solito vengono implementati tramite librerie o classi. Gli elementi di un insieme sono un sottoinsieme di un tipo base, per esempio:

```
set of char S
set of [0..99]
```

dove il primo insieme è un sottoinsieme di caratteri e il secondo contiene i numeri da 0 a 99. Le operazioni possibili sui valori di tipo insieme sono il test di appartenenza di un elemento ad un insieme e, se gli elementi sono omogenei, le usuali operazioni insiemistiche di unione, intersezione e differenza, mentre non è sempre disponibile il complemento.

L'implementazione standard degli insiemi è mediante un vettore di booleani di lunghezza pari alla cardinalità del tipo base (universo) che definisce la funzione caratteristica la quale, per ogni elemento, decide se esso appartiene o meno all'insieme. Per esempio, un sottoinsieme di char, su un'implementazione che usa il codice ASCII a 7 bit, sarebbe rappresentato su 128 bit. Questa implementazione permette un'esecuzione molto efficiente delle operazioni insiemistiche come operazioni bit a bit sulla macchina fisica, ma è anche evidente che non è adatta per insiemi con un universo di grosse dimensioni. Per ovviare a questo problema, alcuni linguaggi limitano i tipi che possono essere usati come tipi base di un insieme oppure li implementano tramite tabelle hash.

Puntatori

Presenti in più linguaggi rispetto agli insiemi, i puntatori identificano una locazione di memoria in cui è contenuto un dato. Tutti i puntatori dal punto di vista strutturale sono uguali perché sono un indirizzo di memoria, ma vengono tipizzati in base all'oggetto puntato specificando il tipo puntato al momento della dichiarazione.

```
int *i \\ int* i
char *a \\ char* a
```

Le operazioni permesse sui valori di tipo puntatore sono di norma la loro creazione (funzioni di libreria che alloca e restituisce l'indirizzo, come `malloc`), il test di uguaglianza (in particolare l'uguaglianza con `null`) e la loro dereferenziazione (cioè l'accesso al dato puntato). Nel test di uguaglianza ci si può riferire a due nozioni, ovvero uguaglianza della locazione di memoria oppure uguaglianza dell'oggetto puntato, nella maggior parte dei casi si fa riferimento alla prima, che è più stretta e se è vera quella è vera anche la seconda.

L'uso dei puntatori non ha molto senso in linguaggi con modello a riferimento, per esempio in C vengono usati mentre in Java non sono permessi,

questo perché nel RdA ogni variabile contiene già un riferimento in memoria al dato associato (puntatore). In alcuni linguaggi, i puntatori possono far riferimento solo a locazioni di memoria nella heap, questo non è vero per esempio in C perché , data una variabile, si può ottenere un puntatore ad essa:

```
1   int i = 5;
2   p = &i;
```

e se la variabile è stata definita all'interno di un RdA abbiamo allora un puntatore al di fuori della heap.

Puntatori e array in C

Una caratteristica di C è che puntatori e array sono trattati in maniera uniforme, intercambiabili. Esempio:

```
1   int n;
2   int *a;    // puntatore a interi
3   int b[10]; // array di 10 interi
4   ...
5   a = b;     // a punta all'elemento iniziale di b
6   n = a[3];  // n ha il valore del terzo elemento di b
7   n = *(a + 3);  // idem
8   n = b[3];      // idem
9   n = *(b + 3);  // idem
```

le due dichiarazioni di riga 2 e 3 non sono però equivalenti, perché la seconda alloca anche lo spazio in memoria. Dopo la dichiarazione però è possibile fare tutte le seguenti operazioni proprio perché la sintassi sui vettori la si può usare sui puntatori e viceversa. Notare la presenza di aliasing, per esempio: `a[3] = a[3] + 1` modificherà anche `b[3]` perché si riferiscono alla stessa locazione di memoria.

Aritmetica dei puntatori

In C, sui puntatori sono possibili anche alcune operazioni aritmetiche: è possibile incrementare un puntatore, sottrarre tra loro due puntatori, sommare una quantità arbitraria ad un puntatore. Per esempio, se `a` è un puntatore ad un array di interi:

```
a + 3
```

incrementa il valore del puntatore `a` di `12` (`3 *` dimensione intero). In generale gli incrementi sono moltiplicati per la dimensione del tipo dell'elemento del vettore. Questo funziona bene se `a` è effettivamente un puntatore ad un array con almeno tre elementi, infatti non c'è alcuna garanzia di correttezza e si

potrebbe accedere facilmente a zone arbitrarie della memoria il che è un problema di sicurezza.

Nel caso di array multidimensionali le seguenti espressioni sono equivalenti:

```
1  b[i][j]
2  (*(b + i))[j]
3  *(b[i] + j)
4  *(*(b + i) + j)
```

Puntatori e tipi di dato ricorsivi

L'uso principale dei puntatori è nell'implementazione di tipi di dato ricorsivi come liste, code, alberi, alberi binari, che sono tipi di dato che possono crescere durante l'esecuzione del programma. Esempio di implementazione di una lista:

```
1  typedef struct listElem element;
2  struct listElem{
3      int info;
4      element *next;
5  }
```

Una definizione alternativa che fa uso del tipo `union`:

```
1  type int_list = union{
2                        struct val{
3                                   int info;
4                                   int_list next;
5                                 }
6                        void null
7                      }
```

con la quale si vede una lista o come un elemento vuoto, oppure come un record che contiene l'informazione e un puntatore al prossimo elemento.

Dangling reference (riferimenti pendenti)

Un problema con i puntatori è quello chiamato dangling reference, che occorre quando si fa riferimento a zone di memoria che non contengono un tipo di dato compatibile con quello specificato nella dichiarazione del puntatore. Le possibili cause possono essere:

- aritmetica sui puntatori, per esempio incrementando un puntatore in una locazione arbitraria
- accesso ad una locazione precedentemente deallocata sulla heap con `free(p)`

- accesso ad una locazione precedentemente deallocata nei RdA

Esempio di dangling reference in C:

```
1  int *ref;
2  int vett1[2];
3  void proc(){
4      int i = 5;
5      int vett2[2];
6      ref = &i;
7      vett1 = vett2;
8  }
```

Dopo la fine della procedura `proc`, lo spazio del RdA viene deallocato, ma i puntatori globali `ref` e `vett1` fanno ancora riferimento ad oggetti in quello spazio ora deallocato.

Le possibili soluzioni sono:

- restringere l'uso dei puntatori (come Java)
- nessuna deallocazione esplicita e meccanismi di garbage collection, tenendo traccia delle zone occupate e libere
- oggetti puntati solo nella heap (non nei RdA)
- introdurre dei meccanismi di controllo nella macchina astratta per controllare ogni accesso ai dati puntati

7.4 Inferenza di tipo

Inferenza di tipo significa che il compilatore deve capire il tipo di espressioni e sottoespressioni, blocchi di comandi, funzioni e procedure a partire dai tipi dichiarati per ogni identificatore, variabile o costante.

7.5 Equivalenza tra tipi

Due espressioni di tipo `T` e `S` sono equivalenti se denotano lo stesso tipo (ogni oggetto di tipo `T` è anche un oggetto di tipo `S` e viceversa).

Esistono due tipi di equivalenza tra tipi:

- equivalenza per nome
- equivalenza strutturale

7.5.1 Equivalenza per nome

Definizione *Due tipi sono equivalenti per nome solo se hanno lo stesso nome (cioè un tipo è equivalente solo a sé stesso).*

Essere la stessa espressione non è sufficiente, infatti due tipi `x` e `y` definiti nel seguente modo:

```
1   x : array [0..4] of integer
2   y : array [0..4] of integer
```

non sono equivalenti per nome. L'equivalenza per nome è usata in Pascal, Ada e Java.

Nel passaggio dei parametri si può richiedere l'equivalenza di tipo tra parametro formale e attuale: il codice seguente

```
1   int[4] sequenza = {0, 1, 2, 3};
2   void ordina (int[4] vett);
3   ...
4   ordina(sequenza);
```

è lecito in C, ma non in Pascal e Ada perché le due espressioni `vett` e `sequenza` di tipo "anonimo" corrispondono a due tipi distinti (perché non avendo nomi, non hanno neppure lo stesso nome). Affinché questo funzioni anche con l'equivalenza per nome bisogna quindi assegnare un nome al tipo:

```
1   typedef int[4] vettore;
2   vettore sequenza = {0, 1, 2, 3};
3   void ordina(vettore vett);
4   ...
5   ordina(sequenza);
```

e poi usare lo stesso tipo sia per `sequenza` sia per il parametro formale `vett`.

L'equivalenza per nome si distingue in **lasca** e **stretta**: stretta è quella vista fin'ora, mentre in quella lasca, usata in alcuni linguaggi (Pascal e Modula-2 per esempio), una semplice ridenominazione di un tipo non genera un nuovo tipo, ma solo un alias per lo stesso tipo. Per esempio nelle seguenti definizioni:

```
1   type T1 = 1..10;
2   type T2 = 1..10;
3   type T3 = int;
4   type T4 = int;
```

i tipi `T3` e `T4` sono equivalenti, mentre rimangono distinti i tipi `T1` e `T2` (perché `1..10` non è un tipo, ma una semplice espressione). Altro esempio:

```
1   type A = ...;
2   type B = A;    // B è un alias di A
```

sono equivalenti per la lasca, ma non per la stretta.

Da un punto di vista pragmatico l'equivalenza per nome è la scelta che più rispetta l'intenzione del progettista. Se il programmatore ha introdotto due nomi distinti per lo stesso tipo, avrà avuto le sue motivazioni, che il linguaggio rispetta mantenendo i tipi distinti.

7.5.2 Equivalenza strutturale

Una definizione di tipo è **trasparente** quando il nome del tipo non è che un'abbreviazione del tipo che viene definito. In un linguaggio con dichiarazioni trasparenti, due tipi sono equivalenti se hanno la **stessa struttura**, cioè se, sostituendo tutti i nomi con le relative definizioni, si ottengono tipi identici. Se un linguaggio adotta definizioni di tipo trasparenti, l'equivalenza tra tipi che si ottiene si dice equivalenza strutturale.

In breve, due tipi sono equivalenti strutturalmente se quando viene fatto l'unfolding[2] delle loro definizioni si ottiene esattamente la stessa espressione.

Definizione formale *L'equivalenza strutturale fra tipi è la minima relazione d'equivalenza che soddisfa le seguenti tre proprietà:*

- *un nome di tipo è equivalente a sé 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*[3] *a tipi equivalenti, allora essi sono equivalenti.*

Esempio:

```
1  typedef struct{int i; float f} coppia;
2  typedef struct{int j; coppia c} A;
3  typedef struct{int j; struct{int i; float f} c} B;
```

nella seconda definizione il secondo campo è di tipo `coppia`, mentre nella terza definizione il secondo campo è l'unfolding del tipo `coppia`, quindi per l'equivalenza strutturale, i tipi `A` e `B` sono equivalenti.

Anche per l'equivalenza strutturale ci sono due interpretazioni. Le definizioni:

```
1  typedef struct{int i; float f} coppia1;
2  typedef struct{float f; int i} coppia2;
```

[2]Unfolding significa arrivare alla definizione radice, perché un tipo potrebbe essere definito a partire da un altro tipo e così via.

[3]Avere lo stesso costruttore significa per esempio essere due vettori, due matrici o due record e avere lo stesso numero di indici o campi.

generalmente non vengono considerate equivalenti, mentre in ML o Haskell sì. Altro esempio:

```
1   typedef int[0..9] vettore1;
2   typedef int[1..10] vettore2;
```

le due definizioni generalmente non vengono considerate equivalenti, mentre in Ada e Fortran sì.

7.5.3 Equivalenza strutturale vs equivalenza per nome

I linguaggi più recenti tendono a essere definiti usando l'equivalenza per nome. Il problema dell'equivalenza strutturale è che si possono considerare oggetti dello stesso tipo accidentalmente. Esempio:

```
1   type student = {
2       name: string,
3       address: string
4   }
5   type school = {
6       name: string,
7       address: string
8   }
9   type age = float;
10  type weight = float;
```

i tipi `student` e `school` verrebbero considerati lo stesso tipo, anche se concettualmente sono diversi, stessa cosa per `age` e `weight`.

Spesso comunque si usano entrambi i tipi di equivalenza a seconda del costruttore di tipo usato. Per esempio in C viene usata l'equivalenza per nome sui tipi `struct` e `union` e l'equivalenza strutturale per gli array e i tipi definiti con `typedef`, tranne quando coinvolgono appunto `struct` e `union`. In Ada l'equivalenza per nome è molto forte:

```
x1, x2: array (1..10) of boolean;
```

`x1` e `x2` hanno tipi diversi, l'unico modo affinché vengano considerati lo stesso tipo è definire prima il tipo uguale all'espressione e poi definire `x1` e `x2` di quel tipo.

7.6 Compatibilità tra tipi

Definizione *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`*.*

Esempio: compatibilità tra interi e floating point

```
1   int n = 5;
2   float r = 5.2;
3   r = r + n;
```

l'espressione in riga 3 è lecita perché i due tipi sono compatibili.

Anche la compatibilità tra tipi dipende dal linguaggio di programmazione. Esempi di compatibilità che si possono trovare, via via più laschi:
`T` è compatibile con `S` se:

1. `T` e `S` sono equivalenti: è la nozione più restrittiva di compatibilità
2. i valori di `T` sono un sottoinsieme dei valori di `S`: è il caso di un tipo intervallo, contenuto nel suo tipo base (e quindi compatibile con esso)
3. tutte le operazioni sui valori di `S` sono possibili anche sui valori di `T`. L'esempio più semplice (che però non corrisponde alla compatibilità di nessuno dei principali linguaggi) è quello di due tipi record: si supponga di aver dichiarato

   ```
   1   type S = struct{
   2       int a;
   3   }
   4   type T = struct{
   5       int a;
   6       char b;
   7   }
   ```

   dove `T` è un'estensione del record `S` che quindi contiene gli stessi campi di `S` ed eventualmente altri. L'unica operazione possibile su `S` è la selezione del campo `a`, che ha senso anche sui valori di tipo `T`. In questi casi il record più grande è compatibile con quello più piccolo
4. i valori di `T` corrispondono in modo canonico ad alcuni valori di `S`: è possibile convertire un valore di tipo `T` in un valore di tipo `S`. È il caso già citato di `int` compatibile con `float` in cui viene per esempio aggiunta la parte decimale `.0` dopo un numero intero; si osservi che questo caso non rientra nel secondo punto, perché nell'implementazione i valori e le operazioni dei due tipi sono distinti. Stessa cosa per `int` e `long`
5. i valori di `T` possono essere fatti corrispondere ad alcuni valori di `S`: fatta cadere la richiesta di canonicità della corrispondenza, ogni tipo può essere reso compatibile con un altro definendo un modo (convenzionale) per trasformare un valore di `T` in uno di `S`. Un esempio è quello della compatibilità tra `float` o `long` e `int`, in cui viene definita un'operazione di troncamento con cui forzare la conversione

7.6.1 Conversione di tipo

Se `T` è compatibile con `S`, avviene una conversione di tipo, in cui un oggetto di tipo `T` diventa di tipo `S`. Questa conversione di tipo può essere declinata in due modi:

- conversione implicita, o **coercizione** (coercion): quando il compilatore inserisce la conversione, senza lasciarne traccia nel testo del programma
- conversione esplicita, o **cast**: quando la conversione è implementata da una funzione, inserita esplicitamente nel testo del programma

Coercizione

Da un punto di vista sintattico, la coercizione non ha altro significato che quello di annotare la presenza di una situazione di compatibilità. Da un punto di vista implementativo, invece, una coercizione può corrispondere a cose distinte, a seconda della nozione di compatibilità adottata. Ci sono tre possibilità, i tipi sono diversi ma:

1. hanno gli stessi valori e la stessa rappresentazione in memoria. Esempio: tipi strutturalmente uguali, ma nomi diversi: il compilatore controlla solo la compatibilità tra i tipi, senza il bisogno di generare codice di conversione
2. hanno valori diversi, ma la stessa rappresentazione in memoria sui valori comuni. Esempio: intervalli e interi: viene definito un array con indice che può variare in un intervallo, il compilatore deve introdurre un controllo di tipo dinamico che a tempo di esecuzione controlli che il valore intero dell'indice stia all'interno dell'intervallo. Il controllo non sempre viene inserito per motivi di efficienza
3. hanno sia valori che rappresentazione diversi. Esempio: interi e floating point oppure intero e long, il compilatore deve inserire il codice per la conversione

Cast

Le conversioni esplicite, o cast, sono una sintassi ad hoc nel linguaggio che specificano che un valore di un tipo deve essere convertito in un altro tipo. Esempio:

```
1   s = (S) t;
2   r = (float) n;
3   n = (int) r;
```

A seconda dei casi, potrebbe essere necessario del codice macchina di conversione. Si può sempre inserire esplicitamente un cast laddove esista una compatibilità, utile per documentare il codice. I linguaggi moderni tendono a favorire i cast rispetto alle coercizioni perché hanno un comportamento più prevedibile. Non tutte le conversioni esplicite sono consentite, dipende se il linguaggio dispone di una funzione di conversione per il caso specifico.

7.7 Polimorfismo

L'idea base del **polimorfismo** consiste nella possibilità per espressioni (e oggetti) di avere diverso tipo in base al contesto. Possiamo distinguere in:

- **polimorfismo ad hoc**, detto anche **overloading**
- **polimorfismo universale**, diviso a sua volta in **parametrico** e **di sottotipo**

7.7.1 Overloading

Come il nome suggerisce, per Overloading intendiamo un polimorfismo in cui ad un nome corrispondono più oggetti, il nome sovraccaricato appunto, e per capire quale oggetto è denotato da una specifica istanza di tale nome si usa il contesto in cui è utilizzato. L'esempio più semplice in cui si nota tale fenomeno è il caso della somma:

```
1  3 + 5      somma di interi
2  4.5 + 5.3  somma di float
3  4.5 + 5    somma tra float ed intero
4  "abc" + "def"  concatenazione di stringhe
```

Il compilatore traduce il simbolo `+` in modi diversi in base al contesto in cui è stato utilizzato, eseguendo un'apposita analisi. In Java non sempre è possibile capire quale significato usare a tempo di compilazione, alcuni metodi di una sottoclasse infatti possono essere ridefinizioni di metodi già presenti nella classe originale, in questo caso, se mi venisse passato un oggetto della sovraclasse inizialmente istanziato come sottoclasse, e chiamassi uno di quei metodi, sapere quale metodo effettivamente applicare non sarebbe possibile fino alla chiamata vera e propria.

In molti linguaggi è possibile definire più volte la stessa funzione con più argomenti o con argomenti di tipo diverso, quale definizione usare varia in base al linguaggio:

- numero di argomenti (Erlang)
- tipo e numero degli argomenti (C++, Java)
- tipo del risultato (Ada)

7.7.2 Polimorfismo universale parametrico

Diciamo che un tipo funzione ha polimorfismo universale se:

- ha un'infinità di tipi diversi
- è ottenuta per istanziazione da un unico schema generale

Una funzione polimorfa universale quindi è definita da un unico codice che lavora uniformemente su tutte le istanze del suo tipo generale.

Per capire meglio il polimorfismo universale prendiamo come esempio la funzione identità, tale funzione non dipende da nessun tipo: dato un tipo qualsiasi `T`, è sempre possibile ritornare `T`, la funzione allora può essere definita come:

```
identity(x) = x; : <T> -> <T> ∀ T. T -> T
```

In questo caso T è una variabile di tipo.
Un altro esempio può essere la funzione `reverse` che inverte l'ordine di un vettore indipendentemente dal tipo degli elementi dello stesso:

```
reverse(v) = ... ; : <T>[] -> void ∀ T. T[] -> void
```

Il vantaggio del polimorfismo universale parametrico è che non è necessario riscrivere il codice per tipi diversi di dato, ma basta una volta sola.

Polimorfismo in Scheme

In Scheme è quasi implicito perché se che il sistema di tipi è dinamico, è naturale definire funzioni come:

- `(map f list)` che applica la funzione `f` a tutti gli elementi di una lista

  ```
  (T -> S) -> (list T) -> (list S)
  ```

- `(select test list)` che seleziona gli elementi di `list` su cui `test` ritorna `true`

  ```
  (T -> Bool) -> (list T) -> (list T)
  ```

Tali funzioni sono definibili ed eseguibili per tutti gli argomenti, almeno fino a quando a run time non viene identificata l'impossibilità di eseguire una determinata operazione.

Nei linguaggi in cui il sistema di tipi è statico diventa più complicato gestire il polimorfismo.

Polimorfismo parametrico esplicito

Si tratta di una soluzione per il polimorfismo e consiste nel dichiarare esplicitamente il polimorfismo attraverso delle specifiche notazioni, nel caso di C++ per esempio si usa il function template. Prendiamo per esempio la funzione `swap` che scambia due interi:

```
1  void swap (int& x, int& y){
2      int tmp = x; x=y; y=tmp;
3  }
```

Si può definire un template che verrà poi istanziato scegliendo il tipo effettivo di `T`, in questo modo `swap` funziona su qualunque tipo:

```
1  template <typename T>     // T è una sorta di parametro
2  void swap (T& x, T& y){
3      T tmp = x; x=y; y=tmp;
4  }
```

Le istanziazioni sono automatiche:

```
1  int i,j; swap(i,j);   // T diventa int a link-time
2  float r,s; swap(r,s); // T diventa float a link time
3  String v,w; swap(v,w); // T diventa String a link time
```

Meccanismo simile sono i generics in Java:

```
1   public static < E > void printArray( E[] inputArray ) {
2      // Display array elements
3      for(E element : inputArray) {
4          System.out.printf("%s ", element);
5          }
6  }
7
8  public static void main(String args[]) {
9      Integer[] intArray = { 1, 2, 3, 4, 5 };
10     Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
11     printArray(intArray); // pass an Integer array
12     printArray(charArray); // pass a Character array
13 }
```

Bisogna tenere in considerazione come in questo caso il tipo parametrico deve essere un tipo i cui argomenti vengono passati per riferimento, nell'esempio infatti vengono utilizzati Character e Integer invece che char e int, la causa di questo è dovuta all'implementazione del polimorfismo.

In C++ funzioni polimorfe vengono istanziate più volte a seconda dei tipi nella quale è stata chiamata, questo causa una variazione di spazio necessario a memorizzare la procedura a seconda del tipo delle variabili interne generiche.

In Java non viene istanziato più volte il codice polimorfo, bensì si sfrutta il fatto che il modello sia a riferimento e quindi le dimensioni della procedura non sono variabili, questo è anche il motivo per cui nell'esempio dei generics in Java era richiesto l'utilizzo di Character e Integer.

In ML le funzioni non necessitano di una definizione del tipo, l'inferenza viene fatta automaticamente a tempo di compilazione e si identifica per ciascuna funzione il tipo più generale che la descrive. Come in Java non viene replicato il codice della funzione e si accede ai valori tramite riferimento.

```
1  swap(x,y) = let val tmp = !x in
2  x = !y; y = tmp end;
3  val swap = fn : 'a ref * 'a ref -> unit
```

Generalizzazione del polimorfismo parametrico

La formulazione iniziale del polimorfismo parametrico ha alcune limitazioni: prendiamo come esempio una funzione `quickSort(E[] inputArray)` che ordina un determinato array di tipo `E`, se `E` non fosse un tipo confrontabile tale la funzione non sarebbe applicabile dato che deve esistere un operatore di confronto.

A questo problema esistono diverse soluzioni, prima tra tutte è passare anche la funzione di confronto:

```
void quickSort(E[] inputArray, (E*E)->Bool compare)
```

Si tratta comunque di una soluzione non ottimale, dato che passare tutte le funzioni diventa piuttosto prolisso. Nei linguaggi ad oggetti è una buona soluzione richiedere che la classe `E` contenga un metodo di confronto, possibile in Java attraverso le interfacce, nei linguaggi funzionali è possibile richiedere che gli elementi di tipo `E` permettano l'applicazione di una funzione di confronto.

Polimorfismo di sottotipo

Viene usato nei linguaggi ad oggetti, si basa su una relazione di sottotipo `T < S` (`T` sottotipo di `S`) dove:

- un oggetto di tipo `T` ha tutte le proprietà di un oggetto di tipo `S`
- `T` è compatibile con `S`
- in ogni contesto o funzione che accetta un oggetto di tipo `T` posso inserire o passare un oggetto di tipo `S`

In genere conviene combinare le due forme del polimorfismo, di sottotipo e parametrico, per avere una maggiore espressività. Prendiamo come esempio la funzione `select` che dato un vettore di oggetti restituisce quello massimo:

- con solo il polimorfismo di tipo: `D[]`→`D`

  posso applicare `select` ad un vettore di sottotipo di `D`, ma all'elemento viene assegnato tipo `D`

- con entrambi i polimorfismi: ∀ `T < D. <T>[]`→`<T>`

  descrivo meglio il comportamento di `select` ed è possibile dare maggiori informazioni sul tipo di risultato.

Java estende il polimorfismo utilizzando anche la nozione di interface ed implements.

```
1  public <T extends D> T select (T[] vector) {
2  }
3
4  public <T implements D> T select (T[] vector) {
5  }
```

Capitolo 8

Tipi di dato astratti

In questo capitolo presenteremo alcuni dei principali modi con cui un linguaggio può mettere a disposizione dei meccanismi più sofisticati per definire astrazioni sui dati. Le astrazioni sui dati consistono nel nascondere le decisioni sulla rappresentazione delle strutture dati e sull'implementazione delle operazioni. Esempio: una coda FIFO realizzata mediante una lista o un vettore.

Un tipo di dato astratto (o ADT), è caratterizzato dai seguenti aspetti principali:

1. componente: nome dell'oggetto che vogliamo astrarre; esempio: struttura dati, funzione, modulo
2. interfaccia: modi per interagire dall'esterno con il componente e tipo (lista di operazioni)
3. specifica (formale o informale): descrizione del comportamento atteso, proprietà osservabili attraverso l'interfaccia
4. implementazione: organizzazione dei dati, codice delle funzioni, definiti all'interno del componente. Dovrebbero essere invisibili all'esterno, infatti dall'interfaccia non si può accedere all'implementazione, che viene protetta da una capsula che la isola

Esempio di astrazione sui dati: coda di priorità

- Componente
  - coda di priorità: struttura dati che restituisce elementi in ordine decrescente di priorità
- Interfaccia
  - tipo `PrioQueue`

  - operazioni

```
empty : PrioQueue
insert : ElemType * PrioQueue -> PrioQueue
getMax : PrioQueue -> ElemType * PrioQueue
```

- Specifica
  - `insert` aggiunge all'insieme di elementi memorizzati
  - `getMax` restituisce l'elemento a massima priorità e la coda degli elementi rimanenti
- Implementazione
  - un albero binario di ricerca
  - un vettore parzialmente ordinato
  - molte altre possibili implementazioni

Esempio di astrazione sui dati: gli interi

- Componente
  - tipo di dato integer
- Interfaccia
  - costanti, operazioni aritmetiche
- Specifica
  - le leggi dell'aritmetica
  - range di numeri rappresentabili, per evitare errori di overflow
- Implementazioni
  - hardware: complemento a 2, complemento a 1, lunghezza fissa
  - software: interi di dimensione arbitraria
- Tipo di dato astratto?
  - in C: no, posso accedere all'implementazione (attraverso le operazioni booleane)
  - Scheme: sì sugli interi, solo operazioni aritmetiche

8.1 Introduzione

Le idee precedenti si possono applicare in molti ambiti, per esempio in un algoritmo di ordinamento si specifica l'interfaccia con gli oggetti su cui si opera nascondendo l'implementazione. Queste idee, nate negli anni '70, se applicate alle strutture dati definiscono gli ADT. Riassumendo:

- separa l'interfaccia dall'implementazione, in questo modo si può modificare l'implementazione senza modificare il codice
- un ulteriore esempio: un tipo di dato `Set` con funzioni `empty`, `insert`, `union`, `is_member?`, ..., può essere implementato come: vettore, lista concatenata, ...

Usa meccanismi di visibilità e controlli di tipo per garantire la separazione.

ADT per una pila di interi

```
1   abstype Int_Stack{ // componente
2       type Int_Stack = struct{
3                           int P[100];
4                           int n;
5                           int top;
6       }
7   signature      // interfaccia
8       Int_Stack crea_pila()
9       Int_Stack push(Int_Stack s, int k);
10      int toanfirt Stack ci;
11      Int_Stack pop(Int Stack s),
12      bool empty(Int_Stack s);
13
14  operations
15      Int_Stack crea_pila(){
16          Int_Stack s = new Int_Stack();
17          s.n > 0;
18          s.top = 0;
19          return s;
20      }
21
22      Int_Stack push(Int_Stack s, int k){
23          if (s.n == 100) errore;
24          s.n = s.n + i;
25          s.P[s.top] = k;
26          s.top = s.top + l;
27          return s;
```

```
28      }
29
30      int top(Int_Stack s){
31          return s.P[s.top];
32      }
33
34      Int_Stack pop(Int_Stack s){
35          if (s.n ==> 0) errore;
36          s.n = s.n - 1;
37          s.top = s.top - l;
38          return s;
39      }
40
41      bool empty(Int_Stack s){
42          return (s.n == 0);
43      }
44  }
```

Un'altra implementazione per una pila di interi

```
1   abstype Int_Stack{
2       type Int_Stack = struct{
3                           int info;
4                           Int_stack next;
5       }
6
7   signature
8       Int_Stack crea_pila();
9       Int_Stack push(Int_Stack s, int k);
10      int top(Int_Stack s);
11      Int_Stack pop(Int_Stack s);
12      bool empty (Int_Stack s);
13
14  operations
15      Int_Stack crea_pila() {
16          return null;
17      }
18      Int_Stack push(Int_Stack s, int k){
19          Int_Stack tmp = new Int_Stack(); // nuovo elemento
20          tmp.info = k;
21          tmp.next = s;                // concatenalo
22          return tmp;
23      }
```

```
24        int top(Int_Stack s){
25            return s.info;
26        }
27        Int_Stack pop (Int_Stack s){
28            return s.next;
29        }
30        bool empty (Int_stack s){
31            return (s == null);
32        }
33    }
```

8.2 Principio di incapsulamento

Possiamo enunciare la proprietà di indipendenza dalla rappresentazione come segue:

Due implementazioni corrette di una stessa specifica di un ADT sono osservabilmente indistinguibili da parte dei clienti di quel tipo.

Se un tipo gode dell'indipendenza dalla rappresentazione, è possibile modificare la sua implementazione con una equivalente (e forse più efficiente) senza che ciò interferisca col contesto. Il contesto non ha alcun modo per accedere all'implementazione.

8.3 Oggetti e classi

Nei linguaggi ad oggetti i tipi di dati astratti "sono naturali", ovvero sono una componente fondamentale del paradigma orientato agli oggetti, attraverso le classi è immediato implementare degli ADT:

- variabili di istanza (rappresentazione interna) vengono tutte dichiarate private
- solo i metodi che implementano l'interfaccia sono pubblici,
- rispetto ad altre implementazioni degli ADT c'è una diversa sintassi per chiamare i metodi
  - `push(pila, elemento)`, il metodo push è generale, data una pila ed un elemento ritorna una nuova pila composta da quella data in input e l'elemento dato, è implementabile anche nei linguaggi funzionali
  - `pila.push(elemento)`, è un metodo specifico che viene chiamato su un oggetto di tipo pila e riceve come in input l'elemento da aggiungere, è classico dei linguaggi orientati agli oggetti

Inoltre è in generale abbastanza facile ed immediato trasformare il codice di un Abstract Data Type in codice per un linguaggio orientato agli oggetti.

8.4 Moduli

Si tratta di un meccanismo che permette di fare dell'**information hiding**, che consiste nel nascondere alcune informazioni per strutturare meglio il programma e semplificarne la comprensione. I moduli in generale permettono di definire quali informazioni vanno mantenute nascoste e quali invece no, per esempio quali variabili sono locali al modulo e non bisogna accedervi dall'esterno e quali invece sì oppure quali funzioni presenti all'interno del modulo possono essere chiamate dall'esterno e quali no, etc...

I moduli inoltre permettono spesso di implementare la compilazione separata, ovvero dividere il programma in moduli e compilare solo quelli che effettivamente è necessario compilare (perché sono nuovi e non ancora compilati, perché sono stati modificati, etc...) che in programmi di grandi dimensioni può essere necessario.

Una buona prassi è dividere moduli diversi in diversi file ma non è strettamente necessario.

All'interno di un modulo:

- definisco i nomi visibili all'esterno, alcuni in maniera ristretta
  - variabili in solo lettura
  - tipi esportati in maniera opaca, usabili ma non esaminabili
- definisco cosa voglio importare dagli altri moduli, ogni modulo, nel preambolo, specifica:
  - gli altri moduli che vuole importare, e nel dettaglio per ciascun modulo importato, seleziona le parti dei moduli che vuole importare

Il nome con cui vengono chiamati i moduli dipende dal linguaggio utilizzato:

- module: in Haskell, Modula, uno dei primi linguaggi ad usarli, ...
- package: in Ada, Java, Perl, ...
- structure: in ML, ...

I moduli sono presenti anche in Haskell.

```
1  module BinarySearchTree (Bst(Null, Bst), insert, empty, in,
       ...)
2      where
```

```
3   data Bst = ...
4   insert = ...
5   empty = ...
6   ...
```

Dove:

- module è l'intestazione, con una serie di operazioni
- BinarySearchTree è il nome del modulo
- where è una parola chiave
- la lista (Bst (Null, Bst), insert, empty) definisce i nomi esportati, le definizioni che non appaiono nella lista non visibili all'esterno

Va tenuto in considerazione il fatto che le definizioni non presenti nella lista che segue il nome non vengono viste all'esterno, questo rende il modulo la soluzione migliore, in Haskell, per implementare degli ADT.

Per importare un modulo in Haskell bisogna usare la parola chiave `import` all'inizio del codice della funzione che importa, selezionando eventualmente i nomi che si vogliono importare.

```
1   module Main (main)
2       where
3   import BinarySearchTree (Bst (Null), insert, in)
```

Capitolo 9

Paradigma ad oggetti

In questo capitolo vedremo le caratteristiche comuni e le differenze tra linguaggi di programmazione orientati agli oggetti.

9.1 Introduzione

Il paradigma ad oggetti è presente soprattutto in linguaggi imperativi, ma non solo. Si basa sul concetto di oggetto che contiene campi e metodi. I linguaggi orientati agli oggetti possono essere:

- puri: tutti i dati sono oggetti (Smalltalk, Ruby, mentre Java per esempio no perché ci sono gli interi)
- altro: gli oggetti sono un meccanismo di supporto (Ada)

Il paradigma ad oggetti estende il meccanismo dei tipi di dato astratti che permette l'information hiding e l'incapsulamento:

- nascondono la rappresentazione interna di un dato
- accesso al dato attraverso delle funzioni base definite all'interno del tipo di dato (astratto)

con meccanismi che permettono:

- estensione dei dati con riuso del codice (ereditarietà)
- compatibilità tra tipi (polimorfismo di sottotipo)
- selezione dinamica dei metodi

9.1.1 Debolezze del tipo di dato astratto

Per capire perché i tipi di dato astratto potrebbero non essere sufficienti facciamo un esempio: definiamo un tipo di dato contatore

```
1  abstype Counter{
2      type Counter = int;
3      signature
4          void reset(Counter C);
5          int get(Counter C);
6          void inc(Counter C);
7      operations
8          void reset(Counter C){
9              c = 0
10         }
11         void get(Counter C){
12             return c
13         }
14         ...
15 }
```

La rappresentazione concreta del tipo `Counter` è il tipo degli interi; le sole operazioni possibili sono l'azzeramento del contatore, la lettura del suo valore e il suo incremento. Supponiamo quindi di voler estendere il tipo di dato con nuove operazioni, per esempio, vogliamo tener conto di quante volte abbiamo chiamato l'operazione `reset` su un dato contatore. Per farlo dobbiamo aggiungere una variabile che conta il numero di reset e un'operazione che ne ritorna il valore. Per ottenere questo risultato abbiamo due possibilità: riscrivere un dato nuovo da zero, oppure sfruttare il tipo `Counter` per creare un nuovo tipo che lo contiene. Entrambe queste possibilità hanno in comune il fatto che non c'è nessuna relazione con il tipo `Counter` e bisogna ridefinire tutte le operazioni. In questo semplice caso può non essere un problema perché sono poche linee di codice, ma l'aggravio in una situazione reale può divenire assai significativo. In più, se si volessero fare modifiche ad uno dei due tipi, non si ripercuoterebbero sull'altro, ma bisognerebbe riscriverle manualmente. Un altro problema è che i due tipi non sono compatibili.

I tipi di dato astratti garantiscono incapsulamento ed occultamento dell'informazione, ma sono rigidi da impiegare in un progetto di una certa complessità. Da quanto abbiamo appena detto non è irragionevole prevedere costrutti che:

- permettano l'incapsulamento e l'occultamento dell'informazione
- permettano di estendere un tipo di dato astratto potendo ereditare parte del codice

- permettano la compatibilità tra tipi, ovvero poter utilizzare il tipo di dato esteso in contesti che accettano il tipo di dato originale
- permettano la selezione dinamica dei metodi: procedure che usano in maniera uniforme un tipo e un'estensione dello stesso, devono decidere a tempo di esecuzione quale implementazione dei metodi utilizzare

9.2 Concetti fondamentali

9.2.1 Oggetti

Un oggetto è una capsula (record, struct) che contiene:

- campi: dati
- metodi: procedure associate ai dati, formalmente memorizzate insieme ad essi

Si possono definire delle variabili di tipo oggetto e poi invocare un metodo su quell'oggetto che risponderà eseguendo la procedura associata.

9.2.2 Classi

Una classe è un modello di un insieme di oggetti, la sua descrizione contiene:

- tipo dei campi (dati)
- codice per l'implementazione dei metodi
- nome, visibilità dei campi e metodi
- costruttori degli oggetti

In un linguaggio con classi, ogni oggetto "appartiene" ad (almeno) una classe, nel senso che la struttura dell'oggetto corrisponde alla struttura fissata dalla classe. Nel seguente esempio:

```
1  class Counter{
2      private int x;
3      public void reset(){
4          x = 0;
5      }
6      public int get(){
7          return x;
8      }
9      public void inc(){
10         x = x + 1;
```

```
11        }
12    }
```

si osservi come la classe riporta l'implementazione dei tre metodi, che sono dichiarati `public`, cioè visibili da chiunque, mentre il dato `x` è `private`, cioè inaccessibile dall'esterno dell'oggetto stesso.

I seguenti aspetti sono presenti con la definizione di una classe:

- information hiding e incapsulamento, si può definire cosa è visibile all'esterno e cosa no (`private, public, protected`)
- astrazione sui dati e sul controllo, assegniamo un nome alla classe, ai campi, ai metodi
- estensibilità e riuso del codice nel caso definissimo delle sottoclassi

9.2.3 Linguaggi prototype based

I linguaggi OO si distinguono in:

- class based: più comuni
- prototype based (object based): non bisogna definire una classe per creare un oggetto
  - oggetti come record con metodi
  - non sono necessari pattern predefiniti

L'esempio di JavaScript

JavaScript è il principale esponente dei linguaggi prototype based. Non ha nessuna parentela con Java ed è il linguaggio di default per introdurre codice in pagine html.

```
1    var person = {firstName:"John", lastName:"Doe", age:50,
2        name = function () {
3        return this.firstName + " " + this.lastName;
4        }
5    };
```

Gli oggetti sono come dei record con l'aggiunta dei metodi. Gli oggetti sono estensibili, possiamo aggiungere nuovi campi o metodi:

```
    person.eyeColor = blue;
```

È un linguaggio con tipi dinamici, il controllo dei tipi viene fatto a run-time. Non c'è nessuna distinzione tra dati e metodi:

- formalmente ogni oggetto contiene i propri metodi, che possono essere replicati o ridefiniti
- implementazione: si memorizza un puntatore a metodi condivisi

In JavaScript c'è la possibilità di definire dei costruttori:

```
function Person(first, last, age) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.name = function () {
        return this.firstName + " " + this.lastName;};
}

var myFather = new Person("John", "Doe", 50);
```

e di estendere i costruttori:

```
Person.prototype.altName = function() {
    return this.lastName + " " + this.firstName;
```

Anche i tipi base sono degli oggetti e hanno i rispettivi costruttori standard:

```
var x1 = new Object();    // A new Object object
var x2 = new String();    // A new String object
var x3 = new Number();    // A new Number object
var x4 = new Boolean();   // A new Boolean object
var x5 = new Array();     // A new Array object
var x6 = new RegExp();    // A new RegExp object
var x7 = new Function();  // A new Function object
var x8 = new Date();      // A new Date object
```

Meccanismo delle closure

In JavaScript non esistono `public private protected`, ma è tutto pubblico. Con il seguente codice però si vede come si potrebbe implementare una sorta di information hiding:

```
function counter() {
    var count = 0;
    return function() {
        return ++count;
    };
}
var closure = counter();
```

```
8   closure(); // returns 1
9   closure(); // returns 2
```

Quando viene eseguita la funzione `counter` si crea il suo RdA, si assegna la funzione `function` alla variabile `closure` e si "elimina" il RdA, nel senso che non è più accessibile direttamente. Siccome `closure` fa riferimento all'ambiente in cui è stata definita, fa ancora riferimento alla variabile `count` presente nel RdA di `counter`, quindi il RdA viene mantenuto insieme a `count`, ma non è più accessibile dall'esterno se non attraverso `closure`.

9.2.4 Identificatori this e self

All'interno dei metodi di un oggetto spesso si fa riferimento a campi dello stesso oggetto oppure a metodi di altri oggetti. Questo può essere fatto:

- in maniera esplicita con le parole chiave `this` o `self` (a seconda del linguaggio):

  ```
  this.field, self.field
  ```

  per affermare che si fa riferimento ad un campo interno

- in maniera implicita: nel metodo appare solo `field`. Implicitamente si intende il campo `field` dell'oggetto corrente

Per esempio in Java il riferimento è implicito ed è utile usare la parola chiave `this` nei casi di mascheramento del campo o del metodo.

9.2.5 Modello a riferimento – variabile

Normalmente i linguaggi di programmazione ad oggetti usano il modello a riferimento:

- le variabili contengono un puntatore all'oggetto
- gli oggetti vengono memorizzati nella heap
- la creazione degli oggetti è esplicita mediante funzioni di costruzione che allocano lo spazio
- sono previsti meccanismi di garbage collection per poter riutilizzare la memoria

Non tutti i linguaggi orientati agli oggetti funzionano in questo modo, per esempio C++ usa il modello a valore:

- le variabili contengono i campi dell'oggetto
- gli oggetti vengono memorizzati direttamente nello stack di attivazione

- la creazione degli oggetti è implicita
- accesso diretto ai dati (non c'è bisogno di seguire il puntatore)

solitamente questo modello non si usa perché è troppo a basso livello.

9.2.6 Incapsulamento

Come già detto in precedenza, l'incapsulamento e l'information hiding sono due concetti molto importanti nella programmazione ad oggetti. Ogni linguaggio permette di definire un oggetto nascondendone una parte attraverso opportuni modificatori che determinano se i campi o i metodi sono:

- pubblici: `public`
- privati: `private`
- protetti: `protected`

Se si etichetta un campo/metodo come `public`, questo può essere acceduto/chiamato anche dall'esterno. La parte pubblica definisce l'interfaccia della classe con l'esterno. Se si etichetta un campo/metodo come `private`, questo può essere acceduto/chiamato solo dall'interno del codice dell'oggetto stesso. Infine, se si etichetta un campo/metodo come `protected`, questo può essere acceduto/chiamato da altre sottoclassi che estendono la classe principale oppure all'interno del package, in base al linguaggio di programmazione.

9.2.7 Metodi o campi statici

Oltre alle etichette di visibilità, i campi e i metodi possono essere etichettati come `static`. Se un campo è `static` significa che è una singola copia di una variabile condivisa tra tutti gli oggetti di una classe, quindi le modifiche su quel campo sono visibili anche agli altri oggetti. Se un metodo è `static` significa che è un metodo generale che non fa riferimento ai campi dei singoli oggetti, ma può accedere solo alle variabili `static`, inoltre non può usare l'identificatore `this`, nemmeno in maniera implicita. Per quanto riguarda la memoria, una variabile `static` locale a una procedura viene memorizzata nella parte statica (globale) della memoria, dunque preserva il suo valore tra una chiamata e l'altra; anche nel caso di procedure ricorsive viene usata sempre la stessa copia.

9.2.8 Costruttori

Gli oggetti vengono creati mediante la chiamata a un costruttore, di solito la sintassi è:

```
Counter c = new Counter();
```

Un costruttore alloca lo spazio necessario a contenere l'oggetto nella heap ed eventualmente inizializza i vari campi. È possibile definire più costruttori, utile nei casi in cui si voglia inizializzare i campi in modo diverso. Normalmente il costruttore ha sempre il nome della classe, quindi in questi casi non si possono differenziare dal nome, bensì la scelta viene fatta in base al numero e al tipo degli argomenti. Se invece il nome del costruttore può essere arbitrario, allora ognuno potrà avere un nome diverso. Nel caso dell'inizializzazione di un oggetto di una sottoclasse, si chiama prima il costruttore della super-classe (solitamente implicita), eventualmente con chiamata esplicita in base al linguaggio, e poi quello della sottoclasse.

9.2.9 Distruttori

È possibile definire anche dei distruttori, che sono dei metodi che vengono eseguiti prima della deallocazione di un oggetto. Il recupero dello spazio di memoria nella heap può essere gestito in modo esplicito (pericoloso perché potrebbe esserci dell'aliasing, portando al problema della dangling reference) oppure tramite il garbage collector, che è meno efficiente (perché ovviamente è un altro programma che deve andare in esecuzione), ma più sicuro.

9.2.10 Sottoclassi

Si possono estendere le classi con nuovi metodi o campi attraverso il meccanismo delle sottoclassi. Esempio di sottoclasse:

```
1  class NamedCounter extends Counter{
2      private String name;
3      public void set_name(String n){
4          name = n;
5      }
6      public String get_name(){
7          return name;
8      }
9  }
```

dove `NamedCounter` è una sottoclasse (o classe derivata) di `Counter`. Ogni istanza di `NamedCounter` ha tutti i campi e i metodi (pubblici e protetti; anche quelli privati, ma non sono accessibili) di `Counter` più quelli che eventualmente vengono dichiarati. Il tipo `NamedCounter` è compatibile con `Counter`.

9.2.11 Ridefinizione di metodi (Overriding)

Una fondamentale caratteristica del meccanismo di sottoclasse è costituita dalla possibilità che una sottoclasse modifichi la definizione (l'implementazione) di un metodo presente nella superclasse.

```
1  class NewCounter extends Counter{
2      private int num_reset = 0;
3      public void reset (){
4          count = 0;
5          num_reset++;
6      }
7      public int get_num_reset() {
8          return num_reset;
9      }
10 }
```

La classe `NewCounter` contemporaneamente estende la classe `Counter` con nuovi campi e ridefinisce il metodo `reset` che avrà quindi due diverse definizioni per `Counter` e `NewCounter`.

9.2.12 Ridefinizione di campi (Shadowing)

Oltre a modificare l'implementazione di un metodo, una sottoclasse può anche ridefinire una variabile d'istanza (o campo) definita in una superclasse. Questo meccanismo si chiama shadowing (mascheramento). Potremmo ad esempio modificare i nostri contatori estesi usando la seguente sottoclasse di `NewCounter` dove, per qualche motivo, il valore iniziale di `num_reset` è inizializzato a 2 e viene incrementato di 2 ogni volta:

```
1  class NewCounterPari extends NewCounter{
2      private int num_reset = 2;
3      public void reset (){
4          count = 0;
5          num_reset = num_reset + 2;
6      }
7      public int get_num_reset() {
8          return num_reset;
9      }
10 }
```

Quello che si ottiene è che ogni riferimento a `num_reset` all'interno di `NewCounterPari` si riferisce alla variabile locale (inizializzata a 2) e non a quella dichiarata in `NewCounter`.

9.2.13 Getter e Setter - Esempio in Ruby

Ruby è un linguaggio ad oggetti puro, tutti i dati sono oggetti e ha la caratteristica che tutti i campi sono di default `private` e quindi ci si può accedere solo tramite metodi specifici.

```
1  class Temperatura
2      def celsius
3          return @celsius
4      end
5      def celsius=(temp)
6          @celsius = temp
7      end
8      def farenheit
9          return @celsius * 9/5 + 32
10     end
11     def farenheit=(temp)
12         @celsius = (temp - 32) * 5/9
13     end
```

A riga 3 viene implicitamente definita la variabile `celsius` tramite `@celsius`. Per accedere a questo campo definiamo in riga 2 un metodo ad hoc che solitamente viene chiamato getter. A riga 5 definiamo invece un metodo setter, che permette di assegnare un valore a `celsius` dall'esterno della classe passando al metodo il valore `temp`.

In Ruby viene tutto ridotto a chiamate di metodi, si usa la sintassi standard attraverso *zucchero sintattico* che la semplifica:

```
1  Temperatura.celsius = 0 # sta per Temperatura.celsius=(0)
2  y = Temperatura.farenheit
3  y + 5 # sta per y.+(5)
```

9.2.14 Sottoclassi e meccanismi di hiding

Una sottoclasse può modificare la visibilità della superclasse. Per esempio in C++ si possono nascondere i metodi pubblici nella superclasse:

```
1  class derived : protected base {...}
2  class derived2 : private base {...}
```

In C++, la scrittura `:` è l'equivalente di `extends` in Java. Possiamo quindi usare i modificatori di visibilità mentre si estende la superclasse per cambiare la visibilità dei metodi. In Java e C# questa possibilità non c'è.

9.2.15 Ereditarietà

Attraverso la definizione di sottoclassi introduciamo un meccanismo di ereditarietà per il riutilizzo e la condivisione del codice. Le modifiche effettuate al codice della sovraclasse si ripercuotono su tutte le sottoclassi.

L'ereditarietà può essere di due tipi in base al linguaggio di programmazione:

- **singola** (Java): ogni classe può ereditare (estendere) da una sola superclasse
- **multipla** (C++): una classe può ereditare da più di una superclasse. Rende più semplice il riuso del codice, ma porta al problema del name clash

Name clash

Si ha conflitto di nomi quando una classe `C` eredita contemporaneamente da `A` e `B`, le quali forniscono entrambe l'implementazione di un metodo con lo stesso nome. Un semplice esempio:

```
1   class A{
2       int x;
3       int f(){
4           return x;
5       }
6   }
7   class B{
8       int y;
9       int f(){
10          return y;
11      }
12  }
13  class C extending A,B{
14      int h(){
15          return f();
16      }
17  }
```

I problemi più interessanti dell'ereditarietà multipla si presentano tuttavia nel cosiddetto **problema del diamante**, che si verifica quando le diverse superclassi da cui si eredita, ereditano da una stessa superclasse.

```
1   class Top{
2       int w;
3       int f(){
4           return w;
5       }
6   }
7   class A extending Top{
8       int x;
9       int g(){
10          return w + x;
11      }
```

```
12  }
13  class B extending Top{
14      int y;
15      int f(){
16          return w + y;
17      }
18      int k(){
19          return y;
20          }
21      }
22  class Bottom extending A,B{
23      int z;
24      int h(){
25          return z;
26      }
27  }
```

In questo esempio `B` esegue un overriding della funzione `f` definita nella sovraclasse `Top`. A questo punto `Bottom` ha a disposizione due versioni del metodo `f`. Anche in questo caso abbiamo il solito problema di conflitto di nomi, ma è soprattutto il problema implementativo quello più rilevante, cioè come far sì che il metodo `f` invocato su un'istanza di `Bottom` selezioni correttamente ed *efficientemente* il codice opportuno.

Le possibili soluzioni sono:

- impedire i conflitti tra i metodi, il compilatore da errore se si cerca di ereditare da due superclassi che definiscono un metodo con lo stesso nome
- in caso di conflitto, specificare a quale versione si vuole fare riferimento. Questa è la soluzione adottata da C++, implementata con la seguente sintassi: `A::f` oppure `B::f`
- in caso di conflitto, la scelta viene fatta in automatico seguendo un criterio predefinito, per esempio: si sceglie il metodo della classe che appare per ultima nel codice

Nessuna di queste soluzioni è ottimale, per ciascuna delle precedenti si possono fare esempi in cui risulta innaturale.

Mixin

Mixin è una soluzione sofisticata alternativa all'ereditarietà multipla, l'idea è che si può ereditare un metodo da una classe anche se non si è una sottoclasse, sfruttando le classi astratte. A seconda del linguaggio di programmazione il mixin viene usato in modo diverso.

9.2.16 Polimorfismo di sottotipo

Un'altra caratteristica dei linguaggi orientati agli oggetti è che c'è in maniera naturale il meccanismo del polimorfismo di sottotipo. Un tipo `T` è sottotipo di `S (T <: S)` quando `T` è compatibile con `S` e quindi nei contesti di tipo `S` si può inserire un oggetto `T`.

Definendo una relazione di sottotipo si possono definire funzioni polimorfe: se `S (T <: S)` e definiamo `f : S -> R`, allora `f : T -> R`. La funzione `f` accetta argomenti sia di tipo `S` che di tipo `T`.

Spesso se la classe `T` è sottoclasse di `S`, allora la classe `T` è anche sottotipo di `S`, infatti se `T` possiede tutti i metodi di `S` possiamo usare gli oggetti `T` in tutti i contesti `S`. Questo non è sempre vero in linguaggi come C++ ed Eiffel perché nelle sottoclassi si possono rendere privati metodi pubblici della superclasse, se questo non accade però sono comunque dei sottotipi.

```
1  class B : public A {...} // B sottotipo di A
2  class C : A {...} // C non sottotipo di A, ma eredita
```

La relazione di sottoclasse non coincide con la relazione di sottotipo anche quando ci sono i binary methods, che sono dei metodi di una classe `A` che hanno come parametro un oggetto dello stesso tipo `A`. Un esempio di binary method è il test di uguaglianza: un oggetto controlla se un altro oggetto è identico a lui controllando campo per campo. I binary method sono incompatibili con la relazione di sottotipo. Non approfondiamo perché diventerebbe complicato.

La relazione di sottotipo è utile per implementare il polimorfismo di sottotipo.

```
1  interface Printable{void print(){...};};
2  void f(Printable x){...};
3
4  // f : (T <: Printable) -> void
```

La funzione `f` può essere chiamata su qualsiasi oggetto di una classe che implementa l'interfaccia `Printable`.

Polimorfismo parametrico

Usiamo parametri di tipo `<T>` per definire classi e funzioni parametriche. Esempio: dato un albero, calcola la sua altezza indipendentemente dal tipo dei nodi

```
1  class Tree<T>{ ...};
2  public static <T> int height (Tree<T> t) {...};
3  Tree<int> tint = new Tree<int>();
4  ...
```

```
5   n = height<int> tint;
6   m = height tint;
7
8   // height: forall T . Tree(T) -> int
```

Combinazione dei due polimorfismi

Definiamo funzioni parametriche con vincoli sul parametro:

```
1   interface Printable{ void print() }
2   public static <T extends Printable> void printAll (Tree<T>
        tTree) { ...}
3
4   // printAll: forall T <: Printable . Tree(T) -> void
```

La funzione stampa tutti i nodi dell'albero indipendentemente dal tipo, purché soddisfino il vincolo di implementare l'interfaccia `Printable`.

9.2.17 Sottoclassi e relazione d'ordine

La relazione d'ordine tra sottoclassi è transitiva: `A <: B, B <: C` implica `A <: C`. In alcuni linguaggi esiste una classe più generale di tutte che solitamente prende il nome di `Object`, tutte le classi sono sue sottoclassi. La classe `Object` contiene metodi comuni a tutte le sottoclassi che per esempio sono la creazione e il test di uguaglianza.

9.2.18 Metodi astratti, classi astratte, interfacce

Questi costrutti sono anche utilizzati per arricchire la relazione di sottotipo.

- **Metodi astratti**: contengono solo la dichiarazione, senza il corpo (codice), etichettata come `abstract`
- **Classe astratta**: se contiene almeno un metodo astratto. Non si possono creare oggetti di una classe astratta, servono come modello per classi concrete (in cui tutti i metodi sono implementati)
- **Interfacce**: classi con solo metodi astratti e nessun campo

I vantaggi delle interfacce sono:

- definire la struttura di una classe senza specificarne il codice; si definisce il tipo dei metodi, ma non la loro implementazione
- è possibile definire il tipo delle classi

- scollega la relazione di sottotipo dalla relazione di ereditarietà: non dovendo ereditare il codice, non c'è alcuna controindicazione ad implementare (essere sottotipo) di più di un'interfaccia

Esempio:

```
1  abstract class A{
2      public int f();
3  }
4  abstract class B{
5      public int g();
6  }
7  class C extending A,B{
8      private x = 0;
9      public int f(){
10         return x;
11     }
12     public int g(){
13         return x + 1;
14     }
15 }
```

Il tipo `C` è sottotipo sia di `A` che di `B`. Anche se le classi `A` e `B` avessero avuto un clash di nomi, non sarebbe stato un problema perché non si eredita il codice, il metodo in conflitto verrà implementato una sola volta nella classe `C`. La gerarchia dei sottotipi non è un albero.

9.2.19 Ereditarietà e sottotipo

È utile distinguere le due nozioni:

- **ereditarietà**: meccanismo per riusare il codice dei metodi, è una relazione tra le implementazioni di due classi
- **sottotipo**: meccanismo che permette di usare un oggetto in un altro contesto, è una relazione tra le interfacce di due classi (interfaccia intesa come insieme di campi e metodi pubblici di una classe)

Sono due concetti formalmente indipendenti, ma sono spesso collegati nei linguaggi OO, infatti con le sottoclassi si ha che l'ereditarietà dei metodi implica la relazione di sottotipo.

Sia C++ che Java hanno costrutti che introducono contemporaneamente le relazioni tra due classi: in Java `extends` introduce sia ereditarietà che sottotipo, mentre `implements` introduce solo il sottotipo.

Esempio in Java:

```
1  interface A{
2      int f();
3  }
4  interface B{
5      int g();
6  }
7  class C{
8      int x;
9      int h(){
10         return x+2
11     }
12 }
13 class D extends C implements A, B{
14     int f(){return x };
15     int g(){ return h() };
16 }
```

9.2.20 Selezione dinamica dei metodi

Sappiamo che nel definire sottoclassi possiamo fare l'overriding di metodi, quindi c'è il problema di scegliere l'implementazione corretta in base al tipo dell'oggetto. Esempio:

```
1  Counter V[100];
2  ...
3  for(Counter count : V){count.reset()}
```

`V` può contenere sia elementi di tipo `Counter` che di tipo `NewCounter`. Può capitare quindi che, nell'istruzione nel corpo del for, ci siano oggetti di entrambi i tipi. La selezione dinamica, a tempo di esecuzione, in funzione del tipo dell'oggetto che riceve sceglie il metodo da chiamare.

9.2.21 Selezione statica

Con la selezione statica, implementata per esempio in C++, si chiama sempre il metodo della classe originaria. Questo permette una maggiore efficienza perché non bisogna eseguire dei controlli di tipo a tempo di esecuzione. Ci sono comunque dei meccanismi per permettere la selezione dinamica (metodi `virtual`).

9.2.22 Duck Typing

Alcuni linguaggi (Python, JavaScript) non si basano sulla relazione di sottotipo, ma sul duck test: "*If it walks like a duck and it quacks like a duck, then it must be a duck*".

L'uso di un oggetto in un contesto è lecito se l'oggetto possiede tutti i metodi necessari, indipendentemente dal suo tipo. Viene usato nei linguaggi con un sistema di tipo dinamico dove viene eseguito un controllo a tempo di esecuzione che se un oggetto riceve un messaggio `m`, `m` appartenga ai suoi metodi. I tipi di dati dinamici hanno maggiore espressività rispetto ai tipi statici, ma minore efficienza, minore controllo degli errori, sono infatti possibili errori di tipo nascosti.

Capitolo 10

Paradigma funzionale

In questo capitolo andremo ad analizzare le caratteristiche dei linguaggi funzionali.

10.1 Imperativi vs funzionali

Nel paradigma imperativo i programmi scritti sono composti da una serie di istruzioni eseguite sequenzialmente, l'architettura dei calcolatori è strutturata per un linguaggio macchina basato su questo paradigma, dato che l'architettura in generale risulta essere basata sulla macchina di Von Neumann. I linguaggi ad alto livello continuano ad essere vincolati al paradigma imperativo, sia per quanto riguarda i linguaggi di Von Neumann sia per quanto riguarda quelli orientati agli oggetti, anche se per questi ultimi è comunque possibile crearli con il paradigma funzionale.

Il paradigma funzionale ha invece una struttura dichiarativa, ovvero il programma corrisponde ad una serie di definizioni di funzioni, la computazione consiste in andare a valutare un'espressione composta dalle funzioni dichiarate in precedenza.

Il fatto che si basi sulla valutazione di funzioni porta ad un'assenza di stato, quindi non esistono neanche variabili modificabili (al limite esistono delle "variabili" costanti che vengono dichiarate, ma non possono essere modificate), l'assenza di stato porta anche ad una naturale assenza di cicli, essi infatti necessitano di considerare più volte un valore di una variabile e modificarlo (sia il while che il for devono variare il valore di una variabile per avere senso).

10.2 Meccanismo di valutazione

Per specificare come le espressioni vengono valutate esistono due approcci.

10.2.1 Meccanismo di valutazione teorico

La valutazione avviene mediante riscritture, ovvero si parte dall'espressione che si vuole valutare e si riscrive fino ad ottenere un risultato finale. Sia data la seguente definizione:

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

La valutazione di `fact 3` consiste nelle seguenti riscritture:

```
1  if 3 == 0 then 1 else 3 * (fatt (3 - 1))
2  3 * (fatt 2)
3  3 * (if 2 == 0 then 1 else 2 * (fatt (2 - 1))
4  3 * 2 * (fatt 1)
5  ...
```

Non è difficile capire come la computazione consista dei seguenti passi:

- sostituire una chiamata di funzione con il suo corpo istanziato:

  `fatt 2` diventa `if 2 == 0 then 1 else 2 * (fatt (2 - 1)`

- applicare le funzioni base:

  `3 - 1` diventa `2`

10.2.2 Meccanismo di valutazione implementato

A livello implementativo non si usa il meccanismo teorico, ma si ricorre all'utilizzo di record di attivazione via via più complessi, la valutazione di `fatt 3` consiste nella valutazione dell'espressione:

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

nell'ambiente:

```
n ==> 3
fatt ==> (lambda n) if n == 0 then 1 else n * (fatt (n - 1)
```

che a sua volta porta alla valutazione di:

```
n == 0
```

nell'ambiente:

```
...
```

Questo meccanismo di valutazione crea una pila di espressioni da valutare con i relativi ambienti, attualmente per l'implementazione si utilizza come base la macchina SECD di Landin (1964), basata proprio su una pila.

10.3 Caratteristiche dei linguaggi funzionali

Come già detto precedentemente i linguaggi funzionali non permettono i cicli, ripetere una valutazione non avrebbe senso perché porterebbe sempre allo stesso risultato, è possibile però utilizzare la ricorsione valutando la stessa espressione con valori diversi (banalmente è possibile simulare i cicli usando la ricorsione di coda).

Le funzioni consistono in oggetti di primo ordine, quindi:

- possiamo avere funzioni di ordine superiore
- le funzioni sono argomento o risultato di altre funzioni
- ammesse funzioni senza nome, per esempio `(lambda (x) (+ x 1))`, espressioni che rappresentano una funzione

Un'ultima caratteristica dei linguaggi funzionali, non strettamente necessaria ma molto comune per questi ultimi è il polimorfismo, implementato di solito in due diversi modi:

- type checking dinamico: Scheme
- type checking statico: Haskell, schemi di tipo, type checking sofisticato

10.4 Vantaggi dei linguaggi funzionali

Uno dei vantaggi principali dei linguaggi funzionali è proprio l'assenza di stato, che semplifica il ragionamento[1] e riduce le possibilità di errori, per esempio non è possibile l'aliasing.

Prendiamo come esempio un caso in cui il paradigma imperativo ha portato ad un errore a causa dell'aliasing e del modello a riferimento.

L'idea era scrivere delle funzioni di libreria per l'accesso protetto a delle risorse, dove abbiamo una risorsa protetta ed una lista di utenti che possono accedervi.

```
1  class ProtectedResource {
2      private Resource theResource = ...; //risorsa privata
3      private String[] allowedUsers = ...; //lista di utenti
4      public String[] getAllowedUsers() {
5      return allowedUsers;
6  }
7
8  public String currentUser() { ...}
9  public void useTheResource() {
10     for(int i=0; i < allowedUsers.length; i++) {
```

[1] Ci dissociamo da questa affermazione

```
11          if(currentUser().equals(allowedUsers[i])) {
12              ...// access allowed: use it
13          return; }}
14      throw new IllegalAccessException();}}
```

L'errore è banale, sebbene l'attributo `allowedUsers` sia privato, il metodo `getAllowedUsers()` ritorna un riferimento a tale oggetto, rendendo possibile a chiunque chiami tale metodo di modificare la lista degli utenti permessi. Tale errore è facilmente risolvibile facendo in modo che non venga passato il riferimento alla lista ma piuttosto ad una copia della stessa. Nei linguaggi funzionali questo non può avvenire dato che si lavora sempre su una copia ed in ogni caso non è modificabile.

10.5 Haskell e Scheme

Nelle sezioni successive andremo a presentare il paradigma funzionale attraverso un esempio concreto: Haskell. Verrà confrontato anche con un altro linguaggio funzionale, Scheme, rispetto al quale ha fatto scelte diametralmente opposte.

Origini

Scheme è abbastanza simile al Lisp (1958, Johh McCarthy), essendo uno dei suoi tanti dialetti, Haskell invece è un po' più moderno dato che il linguaggio vero e proprio nasce negli anni '90, ma il progenitore, ML, nasce nel '73 (David Milner, Edimburgo).

10.5.1 Sintassi

La sintassi tra i due differisce molto, in Scheme si è voluto ricorrere a meno costrutti possibili portando ad una maggiore facilità di apprendimento ma ad una conseguente maggiore prolissità del codice, Haskell presenta anch'esso un non molto elevato numero di costrutti base, ma a differenza di Scheme rende disponibile molto zucchero sintattico rendendo i programmi molto sintetici. Per esempio il seguente codice implementa i crivello di Erastotene in Haskell:

```
1  primes = filterPrime [2..]
2      where filterPrime (p:xs) =
3          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

10.5.2 Sistemi di tipi

Il sistema di tipi differisce tra i due linguaggi, sebbene entrambi i linguaggi siano fortemente tipati, Scheme utilizza un sistema di tipi dinamico mentre Haskell ne usa uno statico.

In entrambi non è necessario esplicitare il tipo all'interno dell'espressione. Nonostante i due linguaggi siano entrambi fortemente tipati, differiscono per il fatto che in Haskell il controllo di tipo viene fatto staticamente, assegnando a tempo di compilazione un tipo per ciascuna variabile cercando di mantenere la consistenza, in Scheme invece l'assegnazione di tipo viene eseguita a run-time eseguendo dei controlli man mano che il codice viene eseguito.
In Haskell inoltre è possibile forzare il tipo.

10.5.3 Meccanismo di valutazione

I due linguaggi hanno un diverso meccanismo di valutazione:

- Scheme: linguaggio eager, call-by-value
  - gli argomenti di una funzione vengono valutati prima di essere passati nel corpo
  - si valuta tutto appena possibile
- Haskell: lazy, call-by-need (passaggio per nome)
  - gli argomenti di una funzione vengono passati così come sono
  - valuto un argomento se proprio non posso farne a meno
  - posso gestire strutture dati infinite (stream)

Esistono delle espressioni che convergono utilizzando una valutazione lazy, ma divergono con una valutazione eager, ma non il viceversa. La valutazione eager è più facile da implementare della lazy, la quale, se implementata in modo semplice (e naiv), porta a valutare lo stesso argomento molte volte con perdite di efficienza.

10.5.4 Vantaggi e svantaggi della valutazione lazy

La valutazione lazy porta con sé alcuni vantaggi, tra cui come già accennato precedentemente, il fatto che porta alla terminazione un maggior numero di espressioni. Un ulteriore vantaggio è che si evita lavoro inutile, non dovendo valutare tutto se non strettamente necessario, inoltre si evitano un maggior numero di errori. Tra gli svantaggi troviamo:

- potenziale ripetizione dell'esecuzione
  - `(\x -> x + x) (fact 1000)`
  - lo si evita con il call-by-need, valuto argomenti replicati una volta sola
- espressioni con side-effect complicate da gestire
  - i side-effect modificano lo stato: memoria, I/0

- se la valutazione di un argomento provoca un side effect, diventa difficile capire quando questo avviene
- soluzione Haskell funzionale puro

10.5.5 Valori

Quando si definisce un linguaggio di programmazione funzionale, soprattutto se eager, è importante descrivere il suo insieme di valori. Nei linguaggi funzionali tutto è un'espressione, quindi conoscere l'insieme dei valori è importante per sapere se un'espressione è stata completamente valutata.

Quindi, per ogni tipo di dato specifichiamo quali sono i suoi valori, per esempio per le costanti numeriche, `1`, `2`, `3`, ... sono i valori. Tutte le altre espressioni di tipo intero no.

Una coppia `(e1, e2)` è un valore se e solo se le sue due componenti `e1` e `e2` sono state valutate e quindi sono a loro volta dei valori.

Le lambda astrazioni sono considerate valori:

```
1  (lambda (x) (+ 3 1)) // funzione costante
2  (lambda () (+ 3 1)) // funzione senza argomento, thunk
```

10.5.6 Purezza

Scheme non è un linguaggio funzionale puro, infatti esistono effetti collaterali che possono modificare lo stato, per esempio in Scheme se abbiamo una matrice e vogliamo modificarne un valore possiamo farlo, se fosse invece un linguaggio puro, dovremmo crearne una nuova con il valore modificato. Un altro esempio è il seguente:

```
1  (define b 1)
2  (define succ (lambda (x) (+ x b)))
3  (set! b 5)
```

A riga 3 si modifica la variabile `b`, il che comporta anche una modifica del comportamento della funzione `succ`. Se avessimo semplicemente ridefinito `b` con il valore `5` la modifica non avrebbe influenzato la funzione `succ`.

Haskell è invece un linguaggio puramente funzionale, come (quasi) tutti i linguaggi lazy perché la gestione dei side-effect nei linguaggi lazy è più complessa in quanto non è chiaro in quale istante un'espressione è valutata e quante volte.

10.6 Haskell - introduzione

In generale, in Haskell sono presenti molti dei concetti di Scheme, ma sotto una veste diversa. Haskell è più sofisticato, ci sono nuovi costrutti e meccanismi per definire programmi.

Il sito haskell.org dispone di tutte le informazioni.

- Tutorial: A gentle introduction to Haskell 98, buono, ma si riferisce ad una versione vecchia di Haskell.
- Compilatore: Glasgow Haskell Compiler (linguaggio compilato).
- Interprete: ghci per l'esecuzione interattiva.

10.7 Valori e tipi

Numerici: varie classi di numeri

```
5 - 5 :: 5 :: Num t => t --- un generico tipo numerico
5.0 :: Fractional t => t --- un generico tipo frazionario
pi :: Floating a => a --- un generico tipo floating point
```

Overloading, ma nessuna coercizione (conversione implicita)

```
(1 + 2) + pi --- accettato, a (1+2) associato un tipo
   numerico generico
(rem 7 4) + pi --- genera errore, Integral incompatibile
```

`(rem 7 4)` e `(1 + 2)` ritornano lo stesso risultato, ma nel valutare le espressioni, nel primo caso valuta la somma come se `1` e `2` fossero numeri floating point per rendere il risultato compatibile con la somma con `pi`, mentre nel secondo caso non può perché non ha senso che un resto sia un valore floating point e quindi da errore di tipo.

10.7.1 Sistema di inferenza di tipi

C'è un sistema di inferenza di tipi, quindi non serve (quasi mai) dichiarare il tipo di un identificatore, è compito del compilatore capire il tipo dal contesto. L'algoritmo cerca lo schema di tipo più generale associabile ad un'espressione. ghci fornisce il tipo di un'espressione con `:type espressione`

```
1 Prelude> :type (rem 7 4)
2 (rem 7 4) :: Integral a => a
```

(`Prelude>` fa parte dell'interfaccia)

10.7.2 Tipi scalari

- Caratteri (Unicode set)

  `'a' :: Char`

- Boolean

  `True False :: Bool`

10.7.3 Tipi composti

Enuple

Le enuple possono contenere elementi di tipi diversi, possono essere di lunghezza arbitraria e si possono innestare

```
('b', 5, pi) :: (Num t, Floating t1) => (Char, t, t1)
('b', ((5, pi), (1,2,3))) --- naturalmente posso iterare
```

Esistono distruttori, ma solo per le coppie: selezione del primo o del secondo valore di una coppia:

```
fst (2,4)
snd (2,4)
fst (2,3,5) --- genera errore di tipo
```

`fst` e `snd` vedremo che non sono molto utili perché se si vogliono esaminare gli elementi di un tipo esiste un costrutto di pattern matching che permette di prelevare i vari elementi dell'enupla.

Liste

La differenza tra liste ed enuple è che le liste possono contenere solo elementi dello stesso tipo.

```
1  [1,2,4] :: Num t => [t]
2  [1,2,pi] :: Floating t => [t]
3  [1,2,'a'] --- type error
4  ['a','b','d'] == "abd" --- :: [Char] stringhe identificate
       con liste di caratteri
```

La riga 2 è accettata perché `1` e `2` vengono visti come floating point. Le due scritture a riga 4 sono equivalenti.

Più formalmente, le liste sono formate da due costruttori: uno è la lista vuota e l'altro è una funzione che permette di inserire elementi dentro la lista

```
1  [] :: [a] --- lista vuota, nil di Scheme
2  (:) :: a -> [a] -> [a] --- concatenazione, infisso, cons
3  1:2:4:[] --- senza parentesi, (:) associa a destra
4  (1:(2:(4:[]))) ---- posso aggiungere parentesi, a
       differenza di Scheme
5  1:2:4:[] == [1,2,4] --- zucchero sintattico
6  'a':'b':'c':[] == "abc" --- zucchero sintattico
```

Si possono specificare delle sequenze aritmetiche similmente ad altri linguaggi come Python:

```
[1..9] --- definizioni compatte
[1,3..20] --- schema più sofisticato
[1,2,4..64] --- non così sofisticato, definibile con list
   comprehension
```

Ci sono i distruttori `head, tail` equivalenti di `car, cdr` di Scheme, anche questi poco usati per la stessa motivazione dei distruttori di enuple.

```
head [2,4,6]
tail [2,4,6]
head (tail [2,4,6])
```

Si possono costruire liste di liste, ma sempre di tipi omogenei

```
[[1,2],[1,2,3],[1]] :: [[Integer]]
[[1,2],[1,2,3], 1 ] --- genera errore di tipo
```

10.7.4 Tipi definiti dall'utente

Esiste in Haskell un meccanismo piuttosto potente e sofisticato per definire un ricco insieme di tipi.

Tipi enumerazione

```
data Colore = Rosso | Verde | Marrone
y = Rosso
```

Haskell è **case sensitive**, ed è importante seguire le regole specifiche di ogni tipo di identificatore, pena un errore: i nomi delle variabili devono essere tutti minuscoli, mentre tipi, costruttori e costanti devono avere la prima lettera maiuscola.

```
data Colore = rosso | Verde | Marrone --- genera errore
data colore = Rosso | Verde | Marrone --- genera errore
```

Il tipo di libreria `Bool` è definito come un'enumerazione ed è ridefinibile:

```
data Bool = True | False
data NBool = True | False
```

Non si può ridefinire più volte un tipo con la stessa struttura perché poi il sistema di inferenza di tipi non funzionerebbe più, non saprebbe di quale tipo è un'espressione.

Tipi record

Permettono di definire dei tipi con più componenti.

```
1  data IntPoint = IPt Integer Integer //dichiarazione
2  IPt 3 6                             //Utilizzo
```

La sintassi è composta da un costruttore, nell'esempio `IPt`, e degli argomenti, sempre nell'esempio i due `Integer`. Se si scrive `IPt` e di seguito due interi viene restituito un tipo `IntPoint` composto dai due interi dati ai quali è possibile accedere in base alla posizione. Mancano le etichette sulle componenti, ma si possono introdurre con una sintassi alternativa. `IntPoint` etichetta la struttura.

Tipi union

Permettono di definire uno stesso tipo con diverso numero di componenti, ovvero più strutture alternative, ciascuna con un proprio costruttore.

```
1  data Point23Dim = Pt2 Integer Integer // dichiarazione
2                  | Pt3 Integer Integer Integer
3
4  Pt2 1 2                                 // utilizzo
5  Pt3 1 2 3
```

Tipi parametrici

Si tratta dei tipi generici già visti nei capitoli precedenti.

```
data Point a = Pt a a
```

In questo esempio abbiamo che:

- il primo `a` è un parametro di tipo, viene interpretato come una variabile di tipo
- `Pt` costruttore generico di tanti tipi diversi che prende due elementi di tipo `a` e costruisce il `Point` di quel tipo

```
1  // tipo ritornato dalla funzione in Haskell
2  Pt 2 3 :: Num a => Point a
3  Pt 'a' 'b' :: Point Char
```

I tipi parametrici sono analoghi ai generics in Java, la definizione precedente è ottenibile (non esattamente, dato che in Haskell non c'è un'istanziazione esplicita dei tipi parametrici) in Java con il seguente codice:

```
1  class Point<T>{
2      private T x,y;
3
4      public void pt(T x, T y){
5          this.x = x;
6          this.y = y;
7      }
8      public T getX(){
9          return x;
10     }
11     public T getY(){
12         return y;
13     }
14 }
```

Nameset

In Haskell è possibile usare lo stesso nome per definire oggetti appartenenti a categorie sintattiche differenti, il significato dipende dal contesto. Per esempio qui viene usato due volte `Point`, ma si può fare perché usano due nameset diversi: il primo definisce un tipo, il secondo definisce una funzione (un costruttore è una funzione). Volendo però, come a riga 2, si può esplicitare il tipo associato, i `::` separano l'espressione dal tipo e quindi i due significati non si sovrappongono.

```
1  data Point a = Point a a
2  Point 2 3 :: Point Integer
```

Enuple

Le enuple possono essere viste come un caso particolare di tipo record parametrico. Esempio di enupla:

```
(3, 'a', ['b']) :: Num t => (t, Char, [Char])
```

La virgola serve per dividere i campi, le parentesi sono obbligatorie.

La definizione precedente è un'istanza del seguente meccanismo più generale:

```
data (_ ,_ ,_ ) a b c = (_, _, _ ) a b c
```

A sinistra dell'uguale abbiamo la definizione del tipo terna con tre parametri `a b c` e a destra il costruttore `(_, _, _ )` che prende tre valori e costruisce la terna.

Si possono definire enuple di lunghezza arbitraria e tutte vengono considerate tipi diversi. Concettualmente abbiamo tanti costruttori ognuno con un numero di argomenti differenti, ma tutti con lo stesso nome.

Tipi ricorsivi (parametrici)

I tipi di dato che si possono definire possono anche essere ricorsivi.

```
1  data Tree a = Leaf a | Branch (Tree a) (Tree a)
2  data List a = Nil | Cons a (List a)
3
4  Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)) :: Tree Integer
5  Cons 'a' (Cons 'z' (Cons '8' Nil)) :: List In
```

Definizione di un nuovo tipo `T`:

- vengono definiti un insieme di costruttori, che sono funzioni che restituiscono elementi di `T`
- di ogni costruttore viene definito il dominio, ovvero la lista di argomenti
- le definizioni sono parametriche rispetto ad uno o più tipi
- le definizioni ricorsive sono lecite

Tipi di dato ricorsivi in Scheme

Possiamo definire tipi di dato ricorsivi anche in Scheme usando le liste. L'approccio canonico è il seguente:

```
1  (define (Leaf a) (list 'LTree a))
2  (define (Branch tl tr) (list 'BTree tl tr))
3  (define (leftSon t) (car (cdr t)))
4  (define (BTree-? t) (eq? (car t) 'BTree))
```

Definiamo i costruttori `Leaf` e `Branch`, i distruttori `leftSon` e `rightSon` e la funzione di controllo `BTree` (dato un oggetto, controlla se è un albero). È conveniente usare il primo elemento della lista come un letterale che definisce l'oggetto che rappresenta quella lista. Non c'è nessuna definizione o controllo di tipo, il programmatore usa uno schema canonico per definire gli oggetti, i controlli vengono fatti a tempo di esecuzione.

Tipi predefiniti come caso particolare

In Haskell, i tipi di dato predefiniti sono quasi tutti un caso particolare dei tipi ricorsivi:

- `Boolean`: tipo di dato predefinito importato da libreria standard

- Enuple, Liste: predefiniti, ma usano una sintassi ad hoc riconoscibile e i tipi che contengono sono standard
- `Integral`: seguono il pattern base, ma contengono troppi elementi per essere definibili in maniera standard
- `Floating`: non sono tipi di dato ricorsivi, non sono ordinali

Equazioni di tipo, equivalenze tra tipi

Possiamo dare un nome alle espressioni di tipo:

```
1  type String = [Char] --- predefinito
2  type Person = (Name,Address)
3  type Name = String
4  data Address = None | Addr String
5
6  type IntTree = Tree Integer
7  type BinaryFunction a = a -> a -> a
8  type BinaryIntFunction = BinaryFunction Integer
```

Con la parola chiave `data` si indica il tipo di dato e il suo costruttore, mentre con la parola chiave `type` associamo ad un'espressione di tipo un certo identificatore (nome).

Haskell implementa l'equivalenza strutturale (sottoparagrafo 7.5.2 a pagina 169), quindi due espressioni di tipo sono uguali se il loro unfolding restituisce la stessa espressione, esempio:

- `[String]` e `[[Char]]` sono espressioni di tipo equivalenti perché anche le stringhe sono liste di caratteri
- anche le seguenti sono tra lo equivalenti
  - `Person -> Name`
  - `(Name,Address) -> Name`
  - `(Name,Address) -> [Char]`

Nota: tipi ricorsivi con nomi diversi, con stessa struttura, non sono equivalenti in Haskell anche se usa l'equivalenza strutturale:

```
1  data Point1 = Point1 Integer Integer
2  data Point2 = Point2 Integer Integer
```

10.8 Variabili e binding

In Haskell esistono delle "variabili", ovvero dei nomi associabili ad un valore, come per esempio:

```
x = 5 + 3
```

Il legame tra valore e variabile è immutabile, gli identificatori non possono essere ridefiniti, se non in ambienti locali, questo significa che se in una parte del codice si descrive una variabile e nello stesso ambiente si riscrive la stessa si ottiene un errore. Diversa è la situazione con l'interprete, dove questo è possibile, non si tratta però di una contraddizione dato che automaticamente l'interprete genera un nuovo ambiente dove la valutazione è valida.

Il legame è lazy, per nome, se associo quindi una variabile ad una espressione quest'ultima non viene immediatamente valutata se non nel momento in cui questa viene forzata. Grazie a questo l'ordine delle dichiarazioni non ha importanza:

```
1  x = y + 3
2  y = 0
3
4  // è equivalente a
5
6  y = 0
7  x = y + 3
```

Il legame lazy rende possibili dichiarazioni mutuamente ricorsive come la seguente:

```
1  x = y + 3
2  y = x - 3
3  z = z + 1
```

Tali dichiarazioni sono lecite e portano a divergere (loop infinito) il programma solo quando viene forzata la valutazione di `x`, `y` o `z`.

Ambienti locali

Gli ambienti locali permettono di ridefinire qualcosa che era stato già definito all'esterno. Il costrutto utilizzato è `let ... in ...`, composto da una parola chiave `let` seguita da delle dichiarazioni, queste ultime seguite dalla parola chiave `in` e, per terminare, un'espressione.

```
1  y = 5          // dichiarazione esterna
2  let y = a + 3  // let seguito da dichiarazioni
3      a = 5
4  in y + a       // in seguito dall'espressione
```

Le dichiarazioni all'interno del `let` possono essere mutuamente ricorsive.

Esiste anche un altro costrutto simile, il `where`, tale costrutto si usa solo all'interno di definizioni per specificare un ambiente dove fare la valutazione:

```
1  y = 2 + z
2  where z = x*x
```

10.9 Funzioni

Essendo Haskell un linguaggio funzionale, le funzioni sono oggetti del primo ordine, quindi è possibile scrivere delle espressioni, che prendono il nome di lambda astrazioni, per descriverle.
La notazione è leggermente diversa da quella di Scheme, consiste in:

```
\ "argomenti" -> "corpo della funzione"
```

Un esempio di dichiarazione è il seguente:

```
\ x -> x + 1
```

dove, al posto di scrivere la lettera greca λ (lambda), si usa il simbolo \ perché è il simbolo che più la ricorda.

Possiamo associare un identificatore alle lambda espressioni ottenendo quindi delle funzioni vere e proprie, sono possibili due alternative equivalenti:

```
1  "identificatore" "argomenti" = "corpo"
2  "identificatore" = \ "argomenti" -> "corpo"
```

Un esempio di dichiarazione è il seguente:

```
1  inc x = x + 1
2  inc = \ x -> x + 1
```

10.9.1 Pattern matching

Haskell permette di definire delle funzioni per casi, dei quali verrà applicato il primo valido.

```
1  fact 0 = 1
2  fact n = n * fact (n-1)
3
4  fib 1 = 1
5  fib 2 = 1
6  fib n = fib (n-1) + fib (n-2)
7
8  fibAux 1 = (0,1)
```

```
9    fibAux n = (snd pair, fst pair + snd pair)
10              where pair = fibAux(n-1)
```

Con il pattern matching possiamo evitare di definire e utilizzare le funzioni `head` e `tail` per le liste nel seguente modo:

```
1    // definizioni per pattern matching
2    head (x:xs) = x
3    tail (_:xs) = xs
4
5    // generano errore in caso di lista vuota
```

Il simbolo _ è una **wildcard** che sostituisce le variabili nei pattern e rende esplicito il non uso della variabile, qualsiasi valore viene accettato. Tutte le **wildcard** sono considerate variabili distinte.

```
1    empty :: [a] -> Bool
2    empty (_:_) = False
3    empty _ = True
```

Alcuni altri esempi di definizioni per pattern matching:

```
1    // lunghezza lista (predefinita)
2    length :: [a] -> Integer
3    length [] = 0
4    length (_:xs) = 1 + length(xs)
5
6    // funzione per applicare una funzione ad ogni elemento di
         una lista (predefinita)
7    map :: (a->b) -> [a] -> [b]
8    map _ [] = []
9    map f (x:xs) = f x : map f xs
10
11   squares = map (\x -> x * x) [1..10] // utilizzo della
         funzione appena definita
```

10.9.2 Currying

Le funzioni a più argomenti possono essere viste come funzioni ad un argomento che ritornano una funzione. Il concetto principale del currying è l'isomorfismo tra `(A * B) -> C` e `A -> (B -> C)`. Le tre seguenti definizioni per la funzione somma sono di conseguenza equivalenti:

```
1    add = \ x -> ( \ y -> (x + y )) // con currying
2    add = \ x y -> x + y
3    add x y = x + y
```

Il currying permette di eseguire delle valutazioni parziali che possono essere utilizzate per definire altre funzioni, utilizzando la prima definizione dell'esempio possiamo definire la funzione successore come `succ = add 1` oppure equivalentemente come `succ = \ x -> add 1 x`.
La versione uncurryed è comunque possibile.

```
1  uncurryedAdd (x, y) = x + y
2  uncurryedAdd :: Num a => (a, a) -> a
3  uncurryedAdd (5,7)
4
5  // notare che le parentesi sono quelle dei tipi enupla
6  // non quelle che racchiudono i parametri di una procedura
```

10.9.3 Regole sintattiche per evitare le parentesi

Per applicare le funzioni in Haskell è sufficiente la giustapposizione, scrivere `f x` è equivalente alla notazione matematica $f(x)$. In Haskell è inoltre possibile usare la stessa notazione di Scheme, `(f x)`. In Scheme le parentesi sono necessarie e forzano la valutazione di un'espressione, mentre in Haskell non sono obbligatorie e servono solo per disambiguare.

Esistono una serie di regole che in Haskell permettono di omettere le parentesi. Una di queste convenzioni è l'associazione a sinistra dell'applicazione, scrivere `add x y` è equivalente a scrivere `(add x) y`, potrei anche scrivere `(add x y)` e `((add x) y)`. Un'altra convenzione è per le espressioni di tipo, la freccia `"->"` associa a destra, quindi:

```
add :: Integer -> Integer -> Integer
```

è un'abbreviazione di

```
add :: Integer -> (Integer -> Integer)
```

10.9.4 Alcune operazioni funzionali standard

La funzione `curry` trasforma una funzione binaria nella sua versione curryficata:

```
1  curry :: ((a, b) -> c) -> a -> b -> c
2  curry f x y = f (x, y)
3  curry f = \ x y -> f (x, y)
```

Esiste anche la funzione inversa:

```
1  uncurry :: (a -> b -> c) -> (a, b) -> c
2  uncurry f (x, y) = f x y
3  uncurry f = \ (x, y) f x y
```

È possibile anche comporre due funzioni:

```
1  compose :: (b -> c) -> (a -> b) -> a -> c
2  compose f g x = f ( g x )
3  compose f g = \ x -> f ( g x )
```

Per quest'ultima esiste una notazione predefinita infissa utilizzando il simbolo "." : `f . g`

10.10 Meccanismo di valutazione, passaggio dei parametri

La valutazione è **lazy**, quindi quando i parametri vengono passati ad una funzione quello che viene effettivamente passato è l'espressione e l'ambiente in cui va valutata (la loro **closure**). La valutazione comunque non avviene finché non viene forzata nel corpo della funzione.

Per esempio se definiamo `v=1/0` non abbiamo un errore fino a che non forziamo la valutazione, come è stato detto più volte. Anche il codice seguente, dove la valutazione dell'argomento darebbe errore, non crea errori:

```
v=1/0
show(if v<0 then 0 else 1)
```

Questo avviene perché la valutazione dell'argomento viene effettuata solo se necessario, la funzione `show`, che trasforma un'espressione in una stringa, non forza la valutazione e quindi non viene generato un errore.

10.11 Dati lazy

La valutazione differita (lazy) permette di definire tipi di dati potenzialmente infiniti, dove di volta in volta viene valutata solo la parte richiesta dal resto del codice.

Prendiamo come esempio il seguente codice:

```
ones = 1 : ones :: Num a => [a]
```

Nell'esempio definiamo ricorsivamente la lista `ones`, la definizione non da problemi perchè verrà espansa solo quando viene forzata la valutazione.

```
head(tail(tail ones)) // i due tail forzano un'espansione
```

Possiamo definire liste più complesse come per esempio la lista di tutti i naturali.

```
1  numsFrom n = n : numsFrom (n + 1)
2  nums = numsFrom 0
```

Per ottenere la lista di tutti i naturali definiamo la funzione `numFrom` che dato un numero intero ritorna una lista con primo elemento l'argomento `n` e come coda la lista ritornata da `numFrom n+1` (che ha come primo elemento `n+1`...). Ovviamente se valutiamo tale funzione con argomento `0` otteniamo la lista di tutti i naturali, nell'esempio `nums` è proprio la lista di tutti i naturali. Altri esempi di dati lazy sono la lista di tutti i quadrati:

```
squares = map (\x -> x^2) (numsFrom 0)
```

Un altro esempio è la sequenza di Fibonacci:

```
1 addList (x:xs)(y:ys) = (x + y) : addList xs ys
2 fib = 1:1:(addList fib (tail fib))
```

Per quest'ultima la sequenza è calcolata in maniera efficiente grazie al call-by-need, `fib` viene replicata, ma valutata una volta sola. Notare come la correttezza sia dovuta al fatto che la somma della sequenza di Fibonacci con la sua traslata di una posizione a destra restituisce la sequenza di Fibonacci meno il primo elemento.

10.11.1 Estrarre parti finite dei dati lazy

Esiste una funzione predefinita (definita nel prelude) `take` che, dato un intero n ed una lista, ritorna i primi n elementi della lista.

```
1 take 0 _ = []
2 take _ [] = []
3 take n (x:xs) = x : take (n-1) xs
```

Esiste inoltre una funzione per selezionare l'i-esimo elemento di uno stream.

```
1 takeEl 0 (x:_) = x
2 takeEl n (_:xs) = takeEl (n-1) xs
```

10.11.2 Strutture dati infinite nei linguaggi eager

Nei linguaggi funzionali eager è più macchinosa la creazione di strutture dati infinite, dato che se si scrive un'espressione questa viene valutata fino a che non diventa un valore, definire la struttura come nel caso dei linguaggi lazy allora non è possibile perché essendo infinita la valutazione non terminerebbe mai.

Per risolvere si sfrutta il fatto che il lambda è un valore.

```
1 (define (numsFrom n) (lambda () (cons n (numsFrom (+ n
     1)))))
2 (define nums (numsFrom 0))
```

Il `lambda ()`, che usato in questo contesto viene chiamato **thunk**, permette di bloccare la valutazione dell'espressione, impedendo una valutazione infinita dovuta alla chiamata `numsFrom (+ n 1)`.

Una volta costruita la "sequenza" è necessario costruire una funzione apposita per stamparla, definiamo quindi la funzione `take` similmente a come è stato fatto in Haskell, a differenza di quest'ultimo usiamo i case.

```
1  (define (take n xs)
2      (cond
3          [(equal? n 0) null]
4          [(equal? (xs) null) null]
5          [true (cons (car (xs)) (take (- n 1) (cdr (xs))))]))
6
7  (take 20 nums)
```

Possiamo anche definire la funzione map per lo stream.

```
1  (define (lazymap f xs)
2      (if (equal? (xs) null)
3          xs
4          (lambda () (cons (f (car (xs)))
5                           (lazymap f (cdr (xs)))))))
6  (take 20 (lazymap (lambda (x) (* x x)) nums))
```

10.12 Sintassi infissa

Diverse funzioni predefinite (come quelle aritmetiche) usano la sintassi infissa:

```
+ * : ^ -
++      // concatenazione di stringhe
.       // composizione di funzioni, si può scrivere come:
        // f . g = \ x -> f ( g x) applica prima g e poi f
```

Si possono anche definire nuove funzioni e usare la forma infissa; si può fare usando identificatori non standard (non testo e numeri):

```
1  (#) a b = rem a b
2  (\\) a b = a + b
```

Nell'esempio mettiamo tra parentesi # e \\ per far sì che diventino gli identificatori delle funzioni e che si possano usare con notazione infissa.

Dobbiamo anche specificare l'associatività e la priorità di ogni identificatore:

```
1  infixl 9 # --- infisso, assoc. a sinistra, priorità 9
```

```
2   infixr 5 \\ --- infisso, assoc. a destra, priorità 5
3
4   7 # 4 # 3 \\ 2 \\ 5 --- diventa ((7 # 4) # 3) \\ (2 \\ 5)
```

Le parentesi permettono di usare gli identificatori in notazione infissa come se fossero in prefissa:

```
(+) :: Num a => a -> a -> a

// forme equivalenti
5 + 4 = 9      // uso normale
(+) 5 4 = 9
```

Inoltre,

- `(+ 4)` è zucchero sintattico per `\ x -> x + 4`
- `(5 +)` è zucchero sintattico per `\ y -> y + 5`

Gli apici permettono la trasformazione opposta, usare un identificatore in notazione prefissa come se fosse in infissa:

```
add :: Num a => a -> a -> a

// forme equivalenti
add 5 4 = 9   // uso normale
5 `add` 4 = 9
```

Questo cambiamento di forma si può applicare anche a funzioni n-arie:

```
add3Values x y z = x + y + z :: Num a => a -> a -> a -> a
sei = (1 `add3Values` 2) 3
```

10.13 List comprehension

Le liste sono un tipo di uso molto comune in Haskell, per questo è utile avere dello zucchero sintattico. Per esempio:

```
[ f x | x <- xs ]
```

applica la funzione `f` a tutti gli elementi della lista `xs`. Questa sintassi richiama una notazione insiemistica (applica la funzione `f` ad `x`, dove `x` è un elemento della lista `xs`). La parte `x <- xs` viene detta *generatore*. Esempio pratico:

```
[ x + 1 | x <- [1..6] ]
```

Ogni elemento della lista che contiene gli elementi da `1` a `6` viene incrementato di `1`.

Sono possibili più generatori:

```
1  [ (x * y) | x <- [1..10], y <- [1..10] ]
2  [[ (x * y) | x <- [1..10]] | y <- [1..10] ]
```

A riga 1 vediamo che `x` prende tutti i valori da `1` a `10`, e per ognuno di questi valori `y` prende anch'esso tutti i valori da `1` a `10`. Otteniamo quindi tutte le combinazioni.

A riga 2 invece diventa una lista di liste in modo da far variare prima tutto `x` e poi tutto `y`.

Possiamo introdurre il meccanismo delle *guardie*, espressioni booleane per filtrare:

```
[ (x * y) | x <- [1..10], y <- [1..10], (mod (x*y) 2) == 1 ]
```

Dopo i generatori mettiamo un'espressione booleana che deve essere verificata per generare l'elemento. Nell'esempio vengono generati solo numeri dispari.

10.13.1 Esempio di uso delle liste: QuickSort

Questa versione di QuickSort non è efficiente come la versione imperativa.

```
1  quickSort [] = []
2  quickSort (x:xs) =
3      quickSort[ y | y <- xs, y < x] ++
4      (x : quickSort[ y | y <- xs, x <= y])
5
6  quickSort ([3,7..40]++[2,5..30])
7  quickSort "Hello World"
```

Le stringhe sono liste di caratteri. Sui caratteri esiste una relazione d'ordine (riga 3) esprimibile con la type class di Haskell `Ord`:

```
quickSort :: Ord a => [a] -> [a]
```

Questo meccanismo somiglia a quello delle interfacce, si richiede che un tipo abbia un insieme di funzioni.

10.14 Funzioni di base e Prelude

In Haskell, ogni volta che si lancia l'interprete o il compilatore, viene caricato il modulo Prelude[2] che contiene delle funzioni predefinite:

[2]Per approfondimenti: A tour of Prelude, Hackage (in generale Hackage offre una panoramica dei package e delle librerie disponibili per Haskell)

- le classiche operazioni aritmetiche e logiche, con usuale notazione infissa, implementate in linguaggio macchina:

```
+ - * / || && < > <= >= == /= ...
```

- una serie di funzioni analitiche con implementazione interna:

```
sin cos tan exp ...
```

- funzioni di I/O
- funzioni Haskell predefinite su liste e numeri

```
head tail take sum mcd lcd max
```

10.14.1 Esercizio sulle liste

Definire una funzione `reverse` che inverte gli elementi di una lista.

```
1 reverseA []=[] // cambiamo il nome perché ègià definita nel
      prelude
2 reverseA(x:xs) = reverseA xs ++ [x]
```

Questa versione è poco efficiente (quadratica) perché ogni volta si concatena un elemento alla fine della lista, che è molto costoso se la lista è lunga.

Definiamo una versione più efficiente che fa uso di un accumulatore e della ricorsione di coda:

```
1 // reverseAux xs acc = reverse xs ++ acc
2 // idea: prende due argomenti, inverte la prima lista e la
      concatena all'accumulatore
3 reverseAux [] acc = acc
4 reverseAux (x:xs) acc = reverseAux xs (x : acc)
5
6 reverseB xs = reverseAux xs []
```

Definiamo ora la versione di `reverse` con le funzioni predefinite `foldl` o `foldr`, ma prima definiamo queste ultime:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

Il primo argomento `(b -> a -> b)` è la funzione che bisogna reiterare, il secondo `b` è il valore restituito in caso di lista vuota, il terzo `[a]` è la lista su cui vogliamo iterare la funzione `f`, il quarto `b` è il risultato finale. Per esempio, `foldl f z l`, usando come valore iniziale `z`, applica `f` a tutti gli elementi della lista da sinistra a destra, ossia:

```
foldl f z [a1, a2, ... , an] = f ( (f ( f z a1 ) a2) ... )
```

Definizione di `foldl`:

```
1  foldl f z [] = z
2  foldl f z (x:xs) = foldl f (f z x) xs
```

Definizione di `reverse` usando `foldl`:

```
reverseC xs = foldl (\ ys x -> (x:ys)) [] xs
```

Definiamo ora la versione di `reverse` con la funzione predefinita `foldr`, ma prima definiamo quest'ultima: tipo

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

come `foldl`, ma in ordine opposto, si parte dall'ultimo elemento della lista, ossia:

```
foldr f z [a1, a2, ... , an] = f a1 (f a2 ... ( f an z ) ... )
```

Definizione di `foldr`:

```
1  foldr f z [] = z
2  foldr f z (x:xs) = f x (foldr f z xs)
```

Definizione di `reverse` usando `foldr`:

```
reverse xs = foldr (\ y ys -> (ys ++ [y])) [] xs
```

10.15 Pattern matching e definizione per casi

Il pattern matching è zucchero sintattico rispetto al costrutto case sull'n-upla degli argomenti. La seguente definizione

```
1  take 0 _ = []
2  take _ [] = []
3  take n (x:xs) = x : take (n-1) xs
```

è equivalente dal punto di vista del compilatore alla seguente scrittura:

```
1  take n xs = case (n, xs) of
2                (0, _) -> []
3                (_, []) -> []
4                (n, (y:ys)) -> y : take (n-1) ys
```

In Haskell sono permesse definizioni per casi con i case che considerano più valori contemporaneamente. Anche questo tipo di case può essere visto come zucchero sintattico per una serie di case sulle singole componenti:

```
1  take n xs = case n of
2                0 -> []
3                _ -> case xs of
4                       [] -> []
5                       (y:ys) -> y : take (n-1) ys
```

Le definizioni per pattern matching rendono il codice più leggibile e permettono di scrivere in maniera sintetica e chiara un insieme di costrutti case annidati tra loro.

10.15.1 Case singolo

Un case viene eseguito nel seguente modo: viene forzata la valutazione dell'argomento `x`, se `x` è uno scalare, viene valutato completamente e il suo valore viene comparato con quello dei vari casi. Se `x` è di tipo `data` (lista, albero, ...), la valutazione per casi avviene in modo più lazy possibile, ovvero viene valutato il minimo indispensabile dell'argomento per capire quale dei casi deve essere applicato. Per esempio se `x` è una lista, viene valutata solo al punto di capire se è una lista vuota oppure un valore e poi un'altra lista (esempio a riga 3 del codice precedente). Esempio:

```
1  data Tree a = Leaf a | Branch (Tree a) (Tree a)
2  height = \ t -> case t of
3      Leaf n -> 0
4      Branch t1 t2 -> max (height t1) (height t2)
```

se è una foglia non valuta il suo valore e se è un branch non valuta i due sottoalberi. Le definizioni potrebbero non essere esaustive, ovvero non tutti i casi vengono considerati. I casi non previsti generano errori a run-time.

10.15.2 Pattern

I pattern possono essere piuttosto complessi. Dato un pattern (per esempio `x : 0 : xs` che controlla che la lista abbia almeno due elementi con il secondo elemento uguale a `0`) ed un'espressione `e` la valutazione del pattern su `e` può:

- fallire: l'espressione valutata non rispetta il pattern (per esempio se il secondo elemento della lista è uguale a `1`
- avere successo: l'espressione rientra nella struttura del pattern
- divergere (o generare errore): la computazione non produce sufficienti risultati, per esempio una lista che non genera mai il primo elemento

10.15.3 Ordine di valutazione

La valutazione degli argomenti viene forzata seguendo l'ordine di scrittura delle regole (case), dalla prima all'ultima. Sulle singole regole, i pattern vengono testati da sinistra a destra:

- se un pattern fallisce, si passa alla regola successiva
- se un pattern diverge, la computazione diverge
- se un (la sequenza di) pattern ha successo, si valuta il corpo con i legami creati dal pattern

Esempio:

```
1  take 0 _ = []
2  take _ [] = []
3  take n (x:xs) = x : take (n-1) xs
```

in riga 1 controlla che l'argomento sia uguale a zero, in riga 2 controlla che il secondo argomento sia generato dal costruttore `[]`.

```
1  take _ [] = []
2  take 0 _ = []
3  take n (x:xs) = x : take (n-1) xs
```

La prima definizione potrebbe divergere mentre la seconda no nel caso in cui dessimo come primo argomento un'espressione numerica che quando viene valutata diverge e come secondo un vettore vuoto, questo perché nella seconda definizione il primo argomento non viene valutato, mentre nella prima deve essere valutato per confrontarlo con `0`.

10.15.4 Pattern con guardie

Ai pattern possiamo far seguire una lista di guardie, che sono delle espressioni booleane esaminate dalla prima all'ultima se il pattern ha successo. Se una guardia viene valutata `True`, allora viene valutato il corpo corrispondente, altrimenti si passa alla guardia successiva. Se tutte le guardie vengono valutate `False` il pattern fallisce e si passa al successivo. Esempio:

```
1  sign x | x > 0 = 1
2         | x == 0 = 0
3         | x < 0 = -1
```

Le guardie si possono anche usare con il costrutto case:

```
1  sign x = case x of
2               y | y > 0 -> 1
3                 | y == 0 -> 0
4                 | y < 0 -> -1
```

10.15.5 If then else

Anche in Haskell si può usare il costrutto if then else:

```
if b then e1 else e2
```

che è equivalente a

```
case b of
   True -> e1
   False -> e2
```

10.15.6 Forma generale del pattern con guardia

```
"funName" "pattern1" | "guardia11" = exp11
                     | ...
                     | "guardia1n" = exp1n
"pattern2" | "guardia21" = exp21
           | ..
           | "guardia2m" = exp2m
"pattern3" ...
...
```

Le guardie con il costrutto case:

```
case "pattern1" | "guardia11" -> exp11
                | ...
                | "guardia1n" -> exp1n
"pattern2" | "guardia21" -> exp21
           | ..
           | "guardia2m" -> exp2m
"pattern3" ...
...
```

10.15.7 Esempio

Calcolare la somma dei numeri pari di una lista, definizione per pattern matching.

```
sommaPari [] = 0
sommaPari (x:xs) | (mod x 2) == 0 = x + sommaPari xs
                 | True = sommaPari xs
```

Se la lista è vuota, la somma e ovviamente zero. Se la lista ha almeno un elemento ed esso è pari viene sommato.

Calcolare la somme di numeri dispari di una lista, definizione via case.

```
1    sommaDispari = \ ys ->
2    case ys of [] -> 0
3                (x:xs) | (mod x 2) == 0 ->
                    sommaDispari xs
4                       | True -> x + sommaDispari xs
```

10.16 Ambiente locale

Come detto in precedenza, le dichiarazioni definiscono un legame immutabile, per cui gli identificatori non possono essere ridefiniti e sono validi su tutto il programma (è accettata la mutua ricorsione). È utile avere ambienti locali, variabili locali e funzioni ausiliarie per definire altre funzioni.

Esistono due costruttori di ambienti locali:

- `let`: definisce un'espressione, si inserisce nel codice prima dell'utilizzo effettivo dell'ambiente

```
1    let x = 5 + y
2        y = 6
3    in x + y
```

- `where`: viene usato nelle definizioni, si inserisce nel codice dopo l'utilizzo effettivo dell'ambiente

```
1    doubleAbs x | x < 0 = - x * y
2                | x == 0 = x
3                | x > 0 = x * y
4    where y = x + 1
```

  in questo esempio, il `where` permetterebbe la mutua ricorsione con l'uso di `doubleAbs` dentro il `where`

10.17 Layout

Negli esempi di codice proposti sono assenti sia separatori ";" tra dichiarazioni, pattern ed espressioni, sia blocchi "{ }" per raccogliere gruppi di dichiarazioni, questo perché vengono dedotti dal compilatore in base al layout del programma:

- i separatori sono impliciti con gli a capo
- i blocchi sono definiti implicitamente attraverso l'indentazione

Separatori e blocchi possono essere anche esplicitati:

```
z = let x = 5 + y
        y = 6
    in x + y

// è equivalente a scrivere

z = let { x = 5 + y; y = 6 } in x + y
```

Una tabulazione imprecisa porta ad errori di compilazione, spesso gli errori ritornati sono poco chiari e non viene messo in evidenza l'errore di tabulazione quindi nel caso ci si scontri con qualche errore e non si riesca a capirne il perché, l'indentazione può essere una causa.

```
z = let x = 5 + y
          y = 6    // indentazione sbagliata!
    in x + y
```

10.18 Record come campi etichettati

Come è stato già descritto precedentemente, in Haskell si identificano le componenti dei data type attraverso la loro posizione, è possibile però accedere alle componenti attraverso pattern matching, nel seguente esempio ne mostriamo un esempio:

```
data Point = Pt Float Float // definizione di Point
p = Pt 3 4

xCoord (Pt x _) = x         // pattern matching per accedere
                            // al primo elemento di point
```

Alternativamente, come nei record, posso accedere alle componenti attraverso etichette **field labels**.

```
data Point = Pt {xCoord, yCoord :: Float}
```

dove la struttura è `xCoord` e `yCoord` sono i nomi dei campi e `Float` è il tipo. Con questa definizione diventano possibili le seguenti espressioni:

```
p = Pt{ xCoord = 3, yCoord =5} // dichiarazione
a = xCoord p       // accedere al campo, xCord
                   // diventa una funzione Point->Float
q = p{ xCoord = 5} // definizione di un nuovo Point
                   // da un altro, cambiando un valore
a2 = xCoord p
b = xCoord q
```

Si può usare la stessa etichetta per diversi costruttori:

```
data Point23 = Pt2 {xCoord, yCoord :: Float}
             | Pt3 {xCoord, yCoord, zCoord :: Float}
```

L'uso delle etichette con tipi union possono causare degli errori verificabili solamente a runtime, per esempio:

```
p = Pt2{xCoord = 3, yCoord = 5}
a = zCoord p
```

La dichiarazione è accettata ma a run-time genera un errore quando si forza la valutazione di `a`.

10.19 Type classes e polimorfismo

Le type classes sono un meccanismo per migliorare il polimorfismo parametrico in Haskell. In generale, Haskell permette di definire funzioni senza specificarne il tipo, esempio classico:

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

`map` è una funzione che prende in input una lista e una funzione e applica quella funzione su ogni elemento della lista. Si può dunque scrivere questo codice senza specificare nessun tipo. A run-time è possibile verificare il tipo assegnato alla funzione `map` perché i controlli di tipo vengono fatti a run-time dove ad ogni espressione viene assegnato un tipo. Il tipo restituito da Haskell è:

```
map :: (a->b) -> [a] -> [b]
```

ed è stato derivato mediante un algoritmo di inferenza di tipo che associa a `map` il tipo più generale che descrive il suo comportamento. L'algoritmo funziona nel seguente modo: inizia guardando la prima definizione, vede che il secondo argomento è una lista e che anche il risultato è una lista, scrive quindi la parte `...[a] -> [b]` dicendo che sono liste, ma senza specificarne il tipo. Dalla seconda definizione vede che il primo argomento viene usato con la forma `f x`, dunque deve essere una funzione, il cui dominio è del tipo di `x`, ovvero `a`, e vede che il risultato viene inserito nella lista risultato, allora il suo codominio sarà di tipo `b`. La soluzione più generale è quindi quella presentata sopra, ma si possono comunque associare anche altri tipi, come per esempio:

```
(a->a) -> [a] -> [a]
oppure
(Char->b) -> [Char] -> [b]
```

che comunque sono ottenibili per istanziazione del tipo più generale. Una possibilità per forzare il tipo di `map` è quella di scrivere una dichiarazione di tipo come quelle precedenti prima della definizione della funzione `map`.

Esistono funzioni polimorfe non descrivibili in questo modo. Esempio:

```
1  quickSort [] = []
2  quickSort (x:xs) = quickSort[ y | y <- xs, y < x]
3                        ++ x : quickSort[ y | y <- xs, x <= y]
```

ha tipo `Ord a => [a] -> [a]`. `quickSort` è applicabile ad un lista `[a]` solo a condizione che ai suoi elementi possano essere applicate le funzioni `<` e `<=`. Questa condizione viene espressa mediante la condizione `Ord a` con il tipo `a` appartenente alla type class `Ord`.

10.19.1 Overloading

Affinché `quickSort` sia polimorfa bisogna permettere che le funzioni `< <=` appartengano a più tipi, eventualmente con implementazioni diverse (si possono definire più algoritmi per confrontare oggetti, per esempio confrontare due interi o due caratteri è diverso), questo si ottiene con l'*overloading*.

10.19.2 Type classes – dichiarazioni e istanziazioni

Definire una classe significa definire una serie di condizioni da porre su un certo tipo.

```
1  class Eq a where
2      (==) :: a -> a -> Bool
```

Questa definizione dice che un tipo `T`, per appartenere alla classe `Eq`, deve avere una funzione `==` (usata in notazione infissa).

Bisogna dire esplicitamente se un tipo appartiene ad una classe, fornendo poi un'implementazione per i metodi di quella classe.

```
1  instance Eq Integer where
2      x == y = integerEq x y
```

10.19.3 Esempio di uso delle type classes

Vogliamo definire una funzione `elem` che dato un elemento e una lista controlla se l'elemento appartiene alla lista.

```
1  elem x [] = False
2  elem x (y:ys) = x == y || (elem x ys)
```

A questa funzione polimorfa viene associato il tipo

```
elem :: (Eq a) => a -> [a] -> Bool
```

perché l'algoritmo si accorge che viene usata la funzione `==`, dunque il tipo deve appartenere alla classe `Eq`. Il simbolo `=>` è da intendersi come implicazione logica: se la condizione (*constraint*) `Eq a` è soddisfatto (il tipo `a` appartiene alla classe `Eq`), allora `elem` ha tipo `a -> [a] -> Bool`.

10.19.4 Implementazione di default dei metodi

La "vera" definizione di `Eq`, classe predefinita del Prelude, ma che può essere ridefinita, è la seguente:

```
1  class Eq a where
2      (==), (/=) :: a -> a -> Bool
3      x /= y = not (x == y)
4      x == y = not (x /= y)
```

dove oltre alla definizione di `==` c'è anche quella dell'opposto.

A livello di documentazione vengono specificate le uguaglianze attese, ma Haskell controlla solo il tipo dei metodi e che i metodi forniti siano sufficienti a determinare tutti gli altri. Per esempio le seguenti definizioni:

```
1  instance Eq Integer where
2      x == y = integerEq x y
3      x \= y = False
4  oppure
5  instance Eq Integer where
6      x == y = integerEq x y
7  oppure
8  instance Eq Integer where
9      x \= y = False
```

sono semanticamente scorrette, ma accettate lo stesso da Haskell (es.: nella definizione a riga 1 sappiamo che la relazione uguale e diverso devono dare risultato opposto, ma così non è).

10.19.5 Definizione di instance polimorfe

```
1  instance (Eq a) => Eq (Tree a) where
2      Leaf l1       == Leaf l2        = l1 == l2
3      Branch tl1 tr1 == Branch tl2 tr2 = tl1 == tl2
4                                          && tr1 == tr2
5      _             == _              = False
```

Quello che vogliamo ottenere è che, a partire dal fatto che su un tipo `a` ci sia un'uguaglianza, vogliamo definire la relazione di uguaglianza su tutti gli

alberi costruiti sul tipo `a`. Nell'esempio abbiamo il primo caso a riga 2 dove confrontiamo due foglie, due foglie sono uguali se i due valori sono uguali. A riga 3 abbiamo il caso in cui si confrontano due sottoalberi, che sono uguali se i loro sottoalberi sinistri e destri sono uguali. A riga 4 abbiamo un caso che viene usato quando si confrontano una leaf o un branch, e dunque saranno sempre diversi.

Queste definizioni strutturali possono essere create automaticamente da Haskell usando `deriving` nella definizione di tipo:

```
1  data Tree a = Leaf a | Branch (Tree a) (Tree a)
2     deriving (Eq)
```

10.19.6 Relazione di sottoclasse

Si può dichiarare una classe come sottoclasse di un'altra

```
1  class (Eq a) => Ord a where
2      (<), (<=), (>), (>=) :: a -> a -> Bool
3      min, max             :: a -> a -> a
4      x <= y = x < y || x == y
5      ...
```

- `Ord` è sottoclasse (subclass) di `Eq`
- `Eq` è superclasse (superclass) di `Ord`

Dichiara che la classe `ord` possiede tutti i metodi della classe `Eq`, quindi `==` e `\=` (le classi di Haskell sono più simili alle interfacce di Java e simili). Nel definire un tipo instance di `Ord` bisogna fornire anche i metodi di `Eq`. Un tipo `T` instance di `Ord` è automaticamente instance di `Eq`.

10.19.7 Sottoclassi multiple

Si può definire una classe come sottoclasse di più classi.

```
1  class (Eq a, Show a) => EqAndShow a where
2  ...
```

Le instance del tipo `EqAndShow` devono possedere i metodi delle due sottoclassi `Eq` e `Show` più eventuali metodi aggiuntivi definiti dopo.

10.20 Visibilità dei nomi

I metodi e classi sono definiti a livello globale (visibili in tutto il programma) quindi devono seguire una serie di regole di seguito elencate:

- uno stesso nome di metodo non può essere dichiarato su più classi
- metodi comuni possibili solo attraverso la definizione di sottoclassi
- metodi, variabili, funzioni sono nello stesso name space quindi non si può usare lo stesso nome per un metodo ed una funzione

10.21 Relazione con i linguaggi Object Oriented

Pur con le dovute distinzioni è possibile notare l'esistenza di una relazione tra le type class e le classi nei linguaggi orientati agli oggetti, il miglior paragone può essere fatto con Java.

Tabella 10.1: Comparazione Haskell e Java

Haskell	Java
Types	Classes
Type Classes	Interfaces
Elements	Objects
Instance Schema Type	Class Type implements Schema{...}
Polimorfismo parametrico con classi	Tipi generici con relazione di sotto tipo

Haskell però non è un linguaggio orientato agli oggetti quindi presenta una serie di differenze con Java, tra cui:

- i metodi non sono collegati agli oggetti, non si usa la sintassi `object.method`
- nessun mascheramento, stato nascosto (nomi privati, pubblici)
- non c'è ereditarietà, ma vengono definite implementazioni di default

10.22 Categorizzazione delle espressioni in Haskell

Le varie categorie di oggetti che si possono definire dentro Haskell sono abbastanza complicate, possono essere categorizzate inizialmente in:

- espressioni semplici, che possono essere valori, come `5 + 2`, o funzioni, `\ x -> x + 1`
- espressioni di tipo, per esempio `integer` oppure `integer->integer`
- costruttori di tipo, per esempio quelli degli alberi:

```
1  data Tree a = Leaf a | Branch (Tree a) (Tree a)
2  data BTree a b =
3      LLeaf a | BBranch b (BTree a b) (BTree a b)
```

- classi, proprietà di tipi, come `Eq` e `Ord`

10.22.1 Kind

Kind è una generalizzazione della nozione di tipo, è un'etichetta che associamo ai tipi più complessi. Se, in generale, vogliamo descrivere le espressioni semplici, ad ognuna associamo un tipo, mentre usiamo i Kind per descrivere gli oggetti delle altre categorie. Esempio: le espressioni semplici hanno un tipo

```
λ> :type (+1)
(+1) :: Num a => a -> a
```

mentre i tipi hanno il kind `*`, che semplicemente significa che è un tipo

```
λ> :kind Integer -> Integer
Integer -> Integer :: *
```

ai costruttori di tipo vengono assegnati dei kind più complessi, esprimono il fatto di essere funzioni da tipo a tipo

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
data BTree a b = LLeaf a | BBranch b (BTree a b) (BTree a b)

λ> :kind Tree
BTree :: * -> *

λ> :kind BTree
BTree :: * -> * -> *

λ> :kind (->)
(->) :: TYPE q -> TYPE r -> *

λ> :kind (->) Integer
(->) Integer :: * -> *
```

Il kind di `Tree` indica che prende un tipo e restituisce un tipo. Il kind del costruttore `(->)` indica che prende due tipi e costruisce il tipo delle funzioni che vanno da uno all'altro (per esempio le funzioni `Integer -> Integer`).

Alle classi i kind esprimono i constraint sul tipo

```
λ> :kind Eq
Eq :: * -> Constraint

λ> :kind Functor
Functor :: (* -> *) -> Constraint
```

10.23 Costruttori di tipo e classi predefinite

Nel prelude viene definito un insieme di classi con molte sottoclassi, principalmente ordini e numeriche. Tipi e costruttori di tipo sono definiti anche come istanze delle opportune classi. Di seguito una rassegna dei tipi e classi predefiniti.

10.23.1 Esempi di classes – Functor

Oltre a `Eq` e `Ord` che abbiamo visto in precedenza, c'è anche `Functor`, che pone delle condizioni su un costruttore di tipo.

```
1  class Functor f where
2      fmap :: (a -> b) -> f a -> f b
```

Nello specifico, un costruttore di tipo deve avere la funzione `fmap`. Esempio: il costruttore di lista

```
1  instance Functor [] where
2      fmap = map
```

Le liste sono dei "funtori" perché assegnano la funzione `map` a `fmap`, questo è possibile perché anche `map` prende una generica funzione `(a -> b)` e, presa una lista di `a`, costruisce una lista di `b`.

Ci si aspetta che le seguenti regole siano soddisfatte:

- `fmap id == id`
- `fmap (f . g) == fmap f . fmap g`

10.23.2 Costruttori di tipo canonici – Maybe

Il costruttore `Maybe` serve a rappresentare eccezioni ed errori.

```
data Maybe a = Nothing | Just a
```

Gli elementi di `Maybe a` sono `Nothing` in caso di eccezione o `Just a` in caso di computazione corretta.

Esempio di uso:

```
myhead :: [a] -> Maybe a
myhead [] = Nothing
myhead (x:xs) = Just x
```

`Maybe` è un funtore

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

10.23.3 Costruttori di tipo canonici – Either

Either mette insieme due tipi diversi, è un po' come fare i tipi Union

```
data Either a b = Left a | Right b
```

10.23.4 Classi numeriche

Haskell ha un ricco insieme di tipi numerici ereditato da Scheme

- interi (dimensione fissa `Int` e arbitraria `Integer`)
- frazionari (coppie di interi)
- floating point (precisione singola e doppia)
- complessi (coppie di floating point)

Catalogato con un ricco insieme di classi.

Num

`Num` è la classe più generale ed è caratterizzata dall'avere quelle operazioni che tutti i numeri possono avere.

```
class (Eq a) => Num a where
    (+), (-), (*) :: a -> a -> a
    negate, abs :: a -> a
    fromInteger :: Integer -> a
```

Per esempio non c'è la divisione perché sarebbe diversa tra interi e floating point.

Real

`Real` è la classe più generale di numeri ordinati. Non ci sono i numeri complessi.

```
class (Num a, Ord a) => Real a where
    toRational :: a -> Rational
```

Le sue sottoclassi si dividono per il tipo di divisione

- `Integral`, numeri con la divisione intera (modulo e resto)

```
class (Real a, Enum a) => Integral a where
    quot, rem :: a -> a -> a
    -- arrotonda verso 0
    -- (x `quot` y)*y + (x `rem` y) == x
    div, mod :: a -> a -> a
    -- arrotonda al numero più piccolo
    -- (x `div` y)*y + (x `mod` y) == x
    toInteger :: a -> Integer
```

- `Fractional`, numeri con divisione esatta

```
class Num a => Fractional a where
    (/) :: a -> a -> a
    recip :: a -> a
```

Floating point

I floating point sono una sottoclasse dei frazionari.

```
class Fractional a => Floating a
-- RealFrac fornisce funzioni di arrotondamento (all'intero
    più vicino)
class (Real a, Fractional a) => RealFrac a
```

Con le relative istanze di tipo `Integer`, `Double`, `Float`, `Int32`, `Natural`. `Num` conta una quarantina di possibili istanze.

10.23.5 Casting esplicito

Per passare da un tipo numerico ad un altro bisogna esplicitare nel codice la conversione. Per farlo, sono a disposizione un insieme di funzioni ad hoc:

```
fromIntegral :: (Num b, Integral a) => a -> b
fromInteger :: Num a => Integer -> a
realToFrac:: (Real a, Fractional b) => a -> b
```

```
4   fromRational :: Fractional a => Rational -> a
5   ...
6   ceiling :: (RealFrac a, Integral b) => a -> b
7   floor :: (RealFrac a, Integral b) => a -> b
8   truncate :: (RealFrac a, Integral b) => a -> b
9   round :: (RealFrac a, Integral b) => a -> b
```

10.24 Input/output in Haskell

La gestione dell'I/O in Haskell è piuttosto complessa lato implementativo, questo è dovuto al fatto che Haskell deve gestire in maniera pulita delle funzioni con effetto collaterale.

Dal punto di vista pratico invece è simile ad altri linguaggi di programmazione. Per fare input/output bisogna definire una variabile `main` di tipo `IO()`. Dentro il `main` c'è una lista di comandi I/O che leggono dati da input, chiamano altre funzioni per valutare il risultato e generano in uscita il risultato finale.

Haskell è un linguaggio funzionale puro, non esiste memoria e stato, vengono definite solo funzioni da un tipo ad un altro, senza effetti collaterali. Questo vincolo rende complesso gestire l'input/output. Anche in una versione semplice in cui abbiamo una stringa di input e una di output, la coppia di stringhe costituiscono uno stato `S` che può essere modificato dalle operazioni di input/output, per esempio:

- input: consuma caratteri nel file di input per generare un valore
- output: da un valore genera un carattere e lo inserisce nel file di output

Per trattare funzioni con effetti collaterali ci sono due alternative:

- la prima soluzione, che è la più usata tra i linguaggi funzionali (non puri) (come Scheme e ML), consiste nel permettere che la valutazione di espressioni abbia effetti collaterali senza segnalarli nel tipo dell'espressione
- la seconda soluzione, usata da Haskell, consiste nel segnalare gli effetti collaterali nel tipo delle funzioni che li possono avere. Esempio: supponiamo di avere la funzione `f :: A -> B` che modifica lo stato, essa viene rappresentata nel seguente modo:

  ```
  fs :: A x S -> B x S
  ```

  la funzione prende in input, oltre al valore `A`, anche lo stato `S` e anche il risultato ha lo stato.

  Per usare funzioni con effetti collaterali in Haskell, si definisce nel modulo principale la funzione

```
main :: S -> (() x S)        main :: IO()
```

l'esecuzione del programma provoca la valutazione di `main` che modifica lo stato.

10.24.1 Il tipo IO

In Haskell esiste un costruttore di tipi predefiniti `IO`, che può essere interpretato intuitivamente come

```
data IO a = (S -> (a,S)) --- pseudo definizione
```

cioè un qualcosa che prende lo stato e restituisce un valore e lo stato modificato.

Le espressioni con tipo `IO a` rappresentano **action**:

- comandi, se hanno tipo `IO()`
- espressioni, di tipo `a` con effetti collaterali se hanno tipo `IO a`

La definizione di tipo `IO` è nascosta, non esistono costruttori espliciti, ma esistono funzioni predefinite che lavorano sui tipi `IO`

- `getChar :: IO Char`, ossia `S -> (Char, S)`. Dato uno stato, restituisce un carattere e modifica lo stato (modificato perché un carattere viene tolto) (prende un carattere dall'input e lo restituisce)
- `putChar :: Char -> IO ()`, ossia `Char -> S -> ( (), S)`. `()` è il tipo delle enuple con nessun elemento, contiene un unico valore, la enupla vuota `()` equivalente al `void` in C. Dato un carattere ed uno stato, `putChar` restituisce lo stato modificato.

10.24.2 Stato

Cosa sia lo stato non è definito esplicitamente, intuitivamente contiene i file di input e output, gli altri file manipolabili e eventuali condizioni di errore o eccezioni. La memoria non c'è, non ci sono variabili modificabili essendo un linguaggio funzionale.

10.24.3 Costrutti per comandi

Le espressioni di tipo `IO` hanno solo un numero limitato di costrutti per manipolarle; la parte imperativa non viene mescolata con quella funzionale. Le uniche funzioni `f` definibili di tipo `(IO a) -> Integer` sono funzioni costanti che preso in input un comando (action) `c` lo ignorano non eseguendolo.

Possiamo comporre i comandi base con `(>>)` e `(>>=)`:

- `(>>)` combina due action `a, b` in uno unico

```
(>>) :: I0 a -> I0 b -> I0 b
((S -> (a, S)) x (S -> (b, S))) -> (S, (b -> S))
```

  Prende lo stato e lo modifica secondo il primo comando, ottiene un valore, che viene ignorato, e uno stato modificato e applica il secondo comando allo stato modificato che a sua volta restituisce un valore e lo stato come risultato

- `(>>=)` compone due comandi (azioni) tenendo conto del risultato generato dal primo comando

```
(>>=) :: I0 a -> ( a -> I0 b) -> I0 b
((S -> (a, S)) x ((a, S) -> (b, S))) -> (S, (b -> S))
```

Possiamo creare un comando con:

- `return :: a -> I0 a  (a -> (S -> (a, S)))`, che prende un valore di tipo `a` e restituisce un'azione che lascia lo stato inalterato e ritorna il valore

10.24.4 Monadi

La sintassi di `(>>)` e `(>>=)` deriva da una nozione generale (monad), poco intuitiva nel caso di composizione di comandi.

```
c1 >> c2 >> c3 --- associa a destra
c1 >> (c2 >> c3)
--- lo posso scrivere come
do { c1; c2; c3 }
--- o come
do  c1
    c2
    c3
--- esempio
do  getChar
    putChar 'h'
    putChar 'e'

--- la notazione:
c1 >>= (\ x -> ( c2 >>= (\ y -> c3)))
--- la posso scrivere come
do  x <- c1
    y <- c2
    c3
```

```
--- esempio
do  c1 <- getChar
    c2 <- getChar
    putChar c2
    putChar c1
```

Le monadi sono una nozione presa dalla matematica (teoria delle categorie) e sono utili per descrivere diverse costruzioni algebriche della matematica, per esempio a partire da un insieme, prendere l'insieme delle parti di quell'insieme. Un altro esempio è quello di prendere un insieme di elementi base come simboli terminali e costruire una grammatica libera. In informatica è utile per descrivere diversi costruttori di tipo (liste, `IO`, `Maybe`). In Haskell esiste una `type class` per costruttori di tipo `Monad m : * -> *`. `IO`, `[]` e `Maybe` sono istanze della type class `Monad` e in quanto tali devono mettere a disposizione le seguenti operazioni:

- `(>>=) :: m a -> (a -> m b) -> m b`
- `return :: a -> m a`

dove al posto del costrutto generico `m` si può mettere `IO` oppure `Maybe` ottenendo definizioni diverse. Le funzioni caratteristiche delle monadi possono essere usate come base per la programmazione.

Esempio

```
--- vogliamo costruire la lista di tutte le coppie di
    valori tra la prima e la seconda lista tali che i due
    valori sono diversi
[(x,y) | x <- [1,2,3] , y <- [3,2,1], x /= y]

--- oppure in questo modo con gli operatori delle monadi
[1,2,3] >>= (\ x -> [3,2,1] >>= (\y -> return (x/=y) >>=
    (\r -> case r of True -> return (x,y)
                     _ -> fail "")))

--- oppure ancora così
do  x <- [1,2,3]
    y <- [3,2,1]
    True <- return (x /= y)
    return (x,y)
```

10.24.5 Separazione tra parte funzionale ed I/O

In Haskell si è costretti a separare la parte di I/O da tutto il resto del codice, le operazioni sui comandi infatti, >> e >>=, prendono in input comandi

e restituiscono comandi, quindi una funzione che prende in input un comando e lo usa in maniera significativa deve essere per forza di cose un comando.

Una funzione di tipo standard (Integer) non può forzare l'esecuzione di un comando ed estrarre informazioni da questo, l'effetto è che tutta quella parte del codice che restituisce qualcosa di diverso dal tipo I/O non ha nessun effetto collaterale, non può interagire con l'ambiente e non legge o stampa nulla, le uniche operazioni con cui tali azioni sono possibili risultano essere solo quelle con tipo finale I/O.

Riassumendo:

- dentro un comando posso chiamare una funzione
- nelle valutazione di una funzione non posso inserire un comando

10.24.6 Entry point

In un programma compilato, bisogna definire l'entry point, ovvero l'espressione che deve essere valutata come risultato dell'esecuzione del programma. In Haskell l'entry point del programma, che permette di usare il programma senza interagire con l'interprete, è denotato dall'espressione che per convenzione ha nome `main`, che deve essere di tipo `I/O` oppure, come spesso accade, tipo `IO ()`; se avesse tipo diverso, l'esecuzione del programma non avrebbe nessun effetto (e avrebbe poco senso). A volte è necessario introdurre dichiarazioni di tipo per il `main`.

```
1  main :: IO()
2  main = do c <- getChar
3            putChar c
4            putChar '\n'
```

L'entry point deve essere inserito nel modulo `main`.

10.24.7 Funzioni di IO

Si possono combinare più comandi di IO per definire funzioni IO più complesse, un esempio è la funzione `getline`, definita a partire da dei comandi base `getChar`. Questa funzione è già presente nel prelude.

```
1  getLine2 :: IO String
2  getLine2 = do c <- getChar
3                if c == '\n'
4                  then return "" --- alias per []
5                  else do l <- getLine2
6                          return (c:l)
```

Similmente posso stampare una stringa usando una serie di `putChar`

```
putString :: String -> IO ()
putString [] = return ()
putString (x:xs) = do putChar x
                      putString xs

--in Prelude 'putStr, putStrLn :: String -> IO ()
```

Un altro esempio è `if then else` in notazione infissa.

10.24.8 Esempio I/O

I comandi possono essere trattati come ogni altro dato, per esempio posso creare un array di comandi:

```
toDoList = [putChar 'H',
            do c <- getChar
            putChar c,
            putStrLn "ello" ]
toDoList :: [IO ()]
```

Non è possibile eseguire i vettori di comandi ma è necessario combinarli in un unico comando da valutare nel main.

```
sequence2 [] = return ()
sequence2 (c:cs) = do c
                      sequence2 cs
```

Una seconda definizione di putString

La funzione putString può essere facilmente ridefinita come:

```
putString2 = sequence . map putChar
```

La funzione `map` prende una serie di caratteri e applica il comando per stamparlo. Il punto è la combinazione di comandi e `sequence` serve per unire tutti i comandi in uno solo.

Tipi e definizioni alternative

```
sequence2 :: Monad m => [m a] -> m ()
sequence2 :: [IO a] -> IO ()

-- foldr :: (a -> b -> b) -> b -> [a] -> b
sequence3 :: [IO a] -> IO ()
sequence3 = foldr (>>) (return ())
```

```
7
8   -- foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
9   sequence4 :: [IO ()] -> IO ()
10  sequence4 = foldl (>>) (return ())
11
12  c2 = sequence2 toDoList
13  c3 = sequence3 toDoList
14  c4 = sequence4 toDoList
```

Si tratta di una composizione del comando `sequence`.

10.24.9 File

La gestione dei file di testo all'interno di Haskell avviene tramite le seguenti funzioni:

```
1   readFile :: FilePath -> IO String
2   writeFile :: FilePath -> String -> IO ()
3   appendFile :: FilePath -> String -> IO ()
4
5   --- dove
6   type FilePath = String
7   -- Defined in 'GHC.IO'
```

Possiamo costruire comandi per copiare un file in un altro:

```
1   copy fileS fileD = do s <- readFile fileS
2                         writeFile fileD s
3
4   c5 = copy "sommaPari.hs" "prova"
```

Oppure comandi per copiare il contenuto dopo averlo modificato:

```
1   modify fSource fDestin trasf = do s <- readFile fSource
2                                     writeFile fDestin (trasf s)
3
4   c6 = modify "sommaPari.hs" "prova" tail
```

10.24.10 Classe Show

Sono funzioni standard per trasformare i dati in una stringa per stampare e leggere valori I/O:

```
1   class Show a where
2     show :: a -> String
3     shows :: a -> String -> String
4     showsPrec :: Int -> a -> String -> String
```

```
5     showList :: [a] -> ShowS
6     {-# MINIMAL showsPrec | show #-}
```

Shows

La funzione `shows` è una versione più efficiente di `show`:

```
shows :: a -> String -> String
```

Prende in input il valore che si vuole stampare e in più una stringa accumulatore per migliorare l'efficienza, evitando di dover concatenare liste in quanto il costo della concatenazione è proporzionale alla lunghezza della stringa.

```
1   putStr ("i num. " ++ show x ++ ' ': show y ++ " sono uguali
        ")
2
3   --- diventa
4
5   putStr ("i num. " ++ shows x (' ': shows y " sono uguali "))
```

Esempio

```
1   data BTree a = Null | BTree a (BTree a) (BTree a)
2
3   showsBTree Null = ('_':)
4   showsBTree (BTree n tl tr) = ('(':) . showsBTree tl . shows
        n . showsBTree tr . (')':)
5
6   showsBTree :: Show a => BTree a -> String -> String
7
8   instance Show a => Show (BTree a) where
9       showsPrec _ = showsBTree
10
11  --- In alternativa:
12  data BTree a = Null | BTree a (BTree a) (BTree a)
13      deriving (Eq, Show)
```

10.24.11 Classe Read

Similmente esiste la classe `Read`:

```
1   class Read a where
2     readsPrec :: Int -> ReadS a
3     readList :: ReadS [a]
```

```
4       ...
5       ...
6       {-# MINIMAL readsPrec | readPrec #-}
7
8       ---notare che il tipo ReadS è:
9       type ReadS a = String -> [(a, String)]
```

La funzione `ReadS` legge una stringa dall'input e la trasforma in un dato. La stringa letta potrebbe contenere più di un dato e quindi, data la stringa, restituisce una lista di coppie (valore, stringa rimanennte) con le varie interpretazioni della stringa.

Implementa un parser usando back-tracking[3]: semplice, ma inefficiente.

Esempio

```
1   readsBTree ('_':s) = [(Null,s)]
2   readsBTree ('(':s) =
3       [((BTree n tl tr), s3) | (tl, s1) <- readsBTree s,
4                                (n, s2) <- reads s1,
5                                (tr, ')':s3) <- readsBTree s2 ]
6
7   readsBTree :: Read a => String -> [(BTree a, String)]
8
9   instance Read a => Read (BTree a) where
10      readsPrec _ = readsBTree
```

L'osservazione da fare è che con un codice semplice si può costruire un parser, tale semplicità ha un costo però in efficenza.

10.25 Moduli

Anche in Haskell esiste il concetto di modulo, trattato in precedenza nel paragrafo 8.4 a pagina 183, i quali permettono di nascondere/rendere disponibili definizioni al programma che lo importa. Un esempio di modulo in Haskell è il seguente:

```
1   module BinarySearchTree (Bst (Null, Bst), insert, empty)
2    where
3   data Bst = ...
4   insert = ...
5   empty = ...
```

[3]Interpretare la stringa di input con un automa a pila che cerca le possibili alternative e scarta quelle non valide fino a trovare quella valida

Dove:

- la sintassi è `module "nome" "lista"`
- la lista (Bst (Null, Bst), insert, empty) definisce in nomi esportati
- le definizioni che non appaiono nella lista non sono visibili all'esterno
- i costruttori di tipo di dato sono messi vicino al tipo per leggibilità
- se si omette la lista dei nomi, tutto viene esportato

I moduli possono essere utilizzati per implementare i tipi di dato astratti nascondendo l'effettiva implementazione del tipo di dato. Viene fornito solo un insieme di funzioni primitive per interagire con il tipo, in questo modo si può cambiare in modo trasparente l'implementazione.

```
module BinarySearchTree (Bst, insert, null, empty, search)

-- nell'esempio precedente abbiamo esportato anche i
    costruttori, non abbiamo nascosto l'implementazione
```

10.25.1 Importazione

```
1 module Main (main)
2     where
3 import BinarySearchTree (Bst (Null), insert)
```

Si possono selezionare i nomi da importare oppure importarli tutti omettendo la lista.

10.25.2 Qualified names

Se un nome viene definito su più moduli e vengono importati tutti si verifica un clash che solitamente viene risolto non importando più volte uno stesso nome usando la parola chiave `except`, per esempio:

```
import BinarySearchTree except (lista delle definizioni da
    non importare)
```

L'altra soluzione è specificare ad ogni utilizzo il modulo a cui fare riferimento usando i qualified names, nomi con prefissi:

```
1 BinarySearchTree.insert
2 Main.insert
```

10.26 Array

Per un'implementazione efficiente, Haskell fornisce il tipo `Array` definito nel modulo `Array` da caricare con il comando:

```
import Array
```

All'interno di questo modulo troviamo le seguenti definizioni:

10.26.1 Index type

Haskell permette una notevole libertà per il tipo degli indici, infatti può essere un qualsiasi tipo che appartiene alla classe `Ix`:

```
1  class (Ord a) => Ix a where
2      range :: (a,a) -> [a]
3      index :: (a,a) -> a -> Int
4      inRange :: (a,a) -> a -> Bool
```

La funzione `range` prende due valori di tipo `a` e restituisce una lista di valori di tipo `a`; l'idea è che la coppia di valori definiscono gli estremi dell'intervallo di valori e il valore restituito è la lista dei numeri compresi nell'intervallo. Nel caso in cui i tipi `a` fossero a loro volta delle coppie, per esempio `((0,1),(10,11))`, la lista restituita conterrà tutti i valori tra 0 e 10 e tra 1 e 11.

La funzione `index`, dato un range e una posizione, restituisce il numero presente in quella posizione nell'intervallo.

La funzione `inRange` restituisce `True` se un valore appartiene all'intervallo, `False` altrimenti.

Sono dichiarati istanze di `Ix` gli interi, i caratteri, i tipi enumerazione, le enuple di elementi di `Ix`, per esempio `(Int, Int)`.

10.26.2 Funzioni sugli array

Creazione

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

Al costruttore va dato il range degli indici e poi la lista degli elementi nella forma di coppie (indice, valore), dove l'indice indica la posizione nell'array dove inserire il valore; gli elementi possono quindi anche essere forniti fuori ordine. In certi casi non serve inizializzare elemento per elemento, ma si possono usare i metodi della list comprehension (paragrafo 10.13 a pagina 222) per esempio:

```
squares = array (1,100) [(i, i*i) | i <- [1..100]]
```

Gli elementi vengono effettivamente valutati e inseriti nell'array solo quando ci si accede (creazione lazy).

Sono possibili definizioni ricorsive.

Per accedere ai singoli elementi di un array si usa il seguente operatore infisso:

```
(!) :: Array a b -> a -> b

squares!7 => 49
```

Accesso agli estremi del campo indici

La funzione `bounds` dato un array restituisce la coppia dei suoi indici estremi.

```
bounds:: Array a b -> (a,a)

bounds squares => (1,100)
```

Modifica

Aggiornare i valori di un array tramite la funzione `//` infissa.

```
1  (//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b
2
3  -- Esempio:
4  swapRows :: (Ix a, Ix b, Enum b) =>
5                    a -> a -> Array (a,b) c -> Array (a,b) c
6
7  swapRows i i' a =
8         a // ([((i, j), a!(i',j)) | j <- [jLo..jHi]] ++
9                 [((i',j), a!(i ,j)) | j <- [jLo..jHi]])
10      where ((iLo,jLo),(iHi,jHi)) = bounds a
```

Alla funzione `//` si fornisce l'array da modificare e la lista di elementi da modificare nella forma (indice, valore nuovo). Formalmente si crea un nuovo array, ma evitando di duplicare tutti gli elementi.

Capitolo 11

Analizzatori in Haskell

In questo capitolo andremo a vedere gli strumenti messi a disposizione da Haskell per eseguire l'analisi lessicale e sintattica, già analizzate per C nel paragrafo 2.7 a pagina 49, in particolare andremo ad analizzare l'analizzatore lessicale Alex e l'analizzatore sintattico Happy che sono strumenti analoghi a Lex e Yacc, ma che producono codice Haskell. Sono concettualmente più semplici e puliti.

11.1 Alex – analizzatore lessicale

Alex permette di generare uno scanner, ovvero presa una stringa la divide in lessemi e per ciascuno produce un token. Il codice Alex è composto da due elementi principali:

- definizione del tipo dei token
- regole, composte come espressioni regolari, che permettono di associare ad una stringa un token

Sono presenti anche delle parti ausiliarie sotto forma di codice aggiuntivo e/o definizioni ausiliarie.

```
1  { -- codice Haskell da mettere in testa
2      module Main (main) where
3  }
4  %wrapper "basic"      -- definisce il tipo di scanner
5
6  $digit = 0-9      -- dichiarazioni ausiliarie
7  $alpha = [a-zA-Z] -- insiemi di caratteri (identificatore $)
8  @num = $digit+    -- espressioni regolari (identificatore @)
9
10 tokens :-             -- nome arbitrario e separatore
11                       -- regole
```

```
12      let        { \s -> TokenLet } -- forma: regexp azione
13      $white+    ;                  -- azione vuota
14      @num       { \s -> TokenInt (read s) }
15      [\=\+\-\*] { \s -> TokenSym (head s) }
16  ...            -- ogni azione stesso tipo :: String -> Token
17
18  { -- codice Haskell
19  data Token =      -- definisco il tipo dei token prodotti
20      TokenLet          |
21      TokenIn           |
22      TokenSym Char     |
23      TokenVar String   |
24      TokenInt Int
25      deriving (Eq,Show) -- token confrontabili e stampabili
26
27  main = do         -- alexScanToken :: String -> [Token]
28      s <- getContents -- s èla stringa del file di input
29      print (alexScanTokens s)
30  }
```

La funzione `getContents` elabora tutto il file, a differenza di `getLine` che prende solo una riga.

Quando un'espressione regolare viene letta non si può eseguire qualunque codice, ma bisogna specificare quale token si vuole generare.

Espressioni regolari di Alex

In Alex le espressioni regolari sono simili a Lex ma presentano alcune differenze, tra cui la separazione di:

- espressioni regolari generiche
- insiemi, espressi regolari che generano singoli caratteri

La sintassi per gli insiemi in Alex consiste in:

- `char` – singolo carattere, se speciale preceduto da `\`
- caratteri non stampabili `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`
- codifica Unicode `\x0A` equivalente `\n`
- `\$iden` – nome per un set di caratteri, definito in precedenza
- `char-char` – range di caratteri, tutti i caratteri nell'intervallo.
- `set0 # set1` – differenza tra insiemi

- `[set0set1set2]` – unioni di insiemi, senza spazi
- `~set o [^set0set1set2]` – complemento di insiemi

Sono presenti anche alcune macro:

- `.` – tutti i caratteri escluso `\n` newline
- `$printable` – tutti i caratteri stampabili
- `$white` – tutti gli spazi `[\ \t\n\f\v\r]`.

Per rappresentare le espressioni regolari è disponibile la seguente sintassi:

- `set` – un insieme
- `@foo` – nome per espressione regolare, definito in precedenza
- `"sdf"` – singola stringa
- `(reg)` – posso usare parentesi, per disambiguare
- `reg1 reg2` – concatenazione
- `reg1 | reg2` – unione
- `reg*` – zero o più ripetizioni
- `reg+` – una o più ripetizioni
- `reg?` – zero o un'istanza
- `reg{n}` – `n` ripetizioni
- `reg{n,}` – `n` o più ripetizioni
- `r{n,m}` – tra `n` e `m` ripetizioni

Opzioni

Alex mette a disposizione la possibilità di avere più opzioni per generare l'analizzatore lessicale, tra cui sono presenti:

- opzione `basic`: si tratta dell'opzione più semplice e fornisce la funzione `alexScanTokens :: String -> [token]`, con questa opzione tutte le azioni devono avere tipo `String->Token` per un qualche tipo token
- opzione `posn` : rispetto a `basic` fornisce delle informazioni sulla posizione, utili per il debugging.

  Utilizza un tipo particolare:

```
data AlexPosn = AlexPn !Int -- posizione nella stringa
                       !Int -- line number
                       !Int -- column number
```

Tutte le azioni devono avere tipo `AlexPosn -> String -> token`, l'argomento di `AlexPosn` è relativo al primo carattere della stringa riconosciuta.

La funzione fornita è: `alexScanTokens :: String -> [token]`.

Un esempio di uso di `posn` è il seguente:

```
...
%wrapper "posn"
...
    "--".*        ;
    let           { \p s -> Let p}
    in            { \p s -> In p}
    $digit+       { \p s -> Int p (read s) }
...
data Token =
    Let AlexPosn      |
    Int AlexPosn Int  |
    ...
...
    print (alexScanTokens s)
```

11.2 Happy – analizzatore sintattico

Happy è lo strumento che serve per costruire parser in Haskell; è simile a Yacc. Interagisce con lo scanner per ottenere la sequenza di token e restituisce l'albero di derivazione, altrimenti in alcuni casi può anche capire la struttura di una stringa che rappresenta un'espressione per poi valutarla, restituendo un intero.

Ci sono alcune similitudini con la funzione `read`: entrambe, data una stringa di simboli, restituiscono il dato. `read` è un algoritmo semplice, simula un automa a pila, bottom-up, non deterministico; Happy si basa sugli automi a pila, bottom-up, deterministici, LALR.

Happy come già detto è simile a Yacc, vede la sequenza di token come un'espressione formata da un certo insieme di sottoespressioni che a loro volta sono formate da sottoespressioni fino ai token base. Ad ogni espressione associa un valore.

Le definizioni principali contenute nel file Happy sono:

- la grammatica che guida il riconoscimento

- i tipi dei valori da associare alle varie sottoespressioni
- funzioni che, per ogni produzione, calcolano il valore associato all'espressione a partire dai valori associati alle componenti

11.2.1 Esempio

Parser per una grammatica delle espressioni aritmetiche `exp` scritte con una sintassi simile a quella di Haskell dove troviamo:

- interi
- variabili
- operatori aritmetici
- costrutto `let var = exp in exp`

Struttura del file Happy

L'**intestazione**, codice da inserire in testa al programma: contiene la dichiarazione del modulo

```
1  {
2      module Main where
3      import Data.Char
4  }
```

Una **sequenza di dichiarazioni**:

```
1  %name calc
2  %tokentype { Token }
3  %error { parseError }
```

dove a riga 1 definiamo il nome della funzione di parsing generata, a riga 2 il tipo dei token in input, in questo caso `calc :: [Token] -> T` e infine a riga 3 definiamo il nome della funzione da chiamare in caso di errore.

Lista dei simboli terminali e non terminali, con lo scopo di poter utilizzare nelle regole di Happy la stessa sintassi di Yacc:

```
1  %token
2      let { TokenLet }
3      in  { TokenIn }
4      int { TokenInt $$ }
5      var { TokenVar $$ }
6      '=' { TokenEq }
7      '+' { TokenPlus }
8      '-' { TokenMinus }
```

```
9       '*' { TokenTimes }
10      '/' { TokenDiv }
11      '(' { TokenOB }
12      ')' { TokenCB }
```

L'idea è di associare ai tipi dei token in Haskell un altro identificatore più simile a quelli di Yacc. Quindi a sinistra troviamo il nome usato nelle regole Happy simile a quelli di Yacc, mentre a destra troviamo il token Haskell usato nel codice. `$$` è un placeholder per il valore associato dal costruttore.

Un **separatore** `%%`.

Le **regole della grammatica**, definiscono come espandere (ridurre a) ogni simbolo non terminale con più alternative. L'azione definisce come costruire l'espressione associata al non terminale a partire dai termini associati alle componenti. Il parser cerca di ridurre l'input a `Exp`; il non terminale descritto nella prima regola, il simbolo iniziale, restituisce il valore corrispondente.

```
1   Exp : let var '=' Exp in Exp { Let $2 $4 $6 }
2       | Exp1                   { Exp1 $1 }
3
4   Exp1 : Exp1 '+' Term         { Plus $1 $3 }
5       | Exp1 '-' Term          { Minus $1 $3 }
6       | Term                   { Term $1 }
7
8   Term : Term '*' Factor       { Times $1 $3 }
9       | Term '/' Factor        { Div $1 $3 }
10      | Factor                 { Factor $1 }
11
12  Factor
13      : int                    { Int $1 }
14      | var                    { Var $1 }
15      | '(' Exp ')'            { Brack $2 }
```

In pratica è l'elenco delle produzioni di una grammatica, dove per ogni non terminale vengono definite le regole di produzione.

L'analisi sintattica viene fatta nel seguente modo: a partire dalla stringa finale, il programma cerca di applicare molte regole al contrario fino ad arrivare al simbolo iniziale. In più, ogni volta che viene applicata una regola al contrario, dice qual è il valore che deve essere associato all'espressione contratta a partire dai valori associati alle singole sottoespressioni (la parte a destra compresa tra `{ }`), i simboli `$1 $2 ...` servono per collegarsi ai valori generati dalle sottoespressioni.

Codice ausiliario: definiamo la funzione per gestire gli errori:

```
1   {
```

```
2       parseError :: [Token] -> a
3       parseError _ = error "Parse error"
4   }
```

e definiamo i tipi dei vari non terminali definiti prima:

```
1   data Exp
2           = Let String Exp Exp
3           | Exp1 Exp1
4           deriving Show
5
6   data Exp1
7           = Plus Exp1 Term
8           | Minus Exp1 Term
9           | Term Term
10          deriving Show
11
12  data Term
13          = Times Term Factor
14          | Div Term Factor
15          | Factor Factor
16          deriving Show
17
18  data Factor
19          = Int Int
20          | Var String
21          | Brack Exp
22          deriving Show
```

In ogni tipo viene definito il costruttore, il primo elemento è il costruttore, il secondo è il tipo, per esempio per `Exp`, il costruttore può essere `Let String Exp Exp` oppure `Exp1 Exp1`, dove il primo `Exp1` è il costruttore e il secondo il tipo del suo argomento.

Definiamo anche il tipo associato ai token. Il tipo è `Token` come lo avevamo definito nella riga `%tokentype { Token }`:

```
1   data Token
2       = TokenLet
3       | TokenIn
4       | TokenInt Int
5       | TokenVar String
6       | TokenEq
7       | TokenPlus
8       | TokenMinus
9       | TokenTimes
10      | TokenDiv
```

```
11      | TokenOB
12      | TokenCB
13  deriving Show
```

Funzione principale

Viene definito il main affinché il programma diventi standalone:

```
1   main = do inputString <- getContents
2       print ( calc (lexer inputString)
3   }
```

Il programma legge l'input e lo assegna alla variabile `inputString` che viene data in input a `lexer` (di tipo: `lexer :: String -> [Token]`) che è l'analizzatore lessicale generato tramite Alex (bisogna importare il modulo relativo e la funzione associata `alexScanTokens` e poi assegnarla con il codice `lexer = alexScanTokens`; nei casi più semplici può essere definita in Haskell internamente). L'output di `lexer` (una sequenza di token) viene dato in input alla funzione `calc` generata da Happy (di tipo: `calc :: [Token] -> Exp`) (era il nome che avevamo assegnato alla funzione di parsing con la definizione `%name calc`). La funzione `print`, a differenza di `putchar`, può prendere qualsiasi tipo di dato, trasformarlo in stringa e stamparlo in output.

Definizione di lexer senza l'uso di Alex

```
1   -- Prende una stringa e genera una lista di token
2   lexer :: String -> [Token]
3   lexer [] = []
4   lexer (c:cs)
5           | isSpace c = lexer cs
6           | isAlpha c = lexVar (c:cs)
7           | isDigit c = lexNum (c:cs)
8   lexer ('=':cs) = TokenEq : lexer cs
9   lexer ('+':cs) = TokenPlus : lexer cs
10  lexer ('-':cs) = TokenMinus : lexer cs
11  lexer ('*':cs) = TokenTimes : lexer cs
12  lexer ('/':cs) = TokenDiv : lexer cs
13  lexer ('(':cs) = TokenOB : lexer cs
14  lexer (')':cs) = TokenCB : lexer cs
15
16  -- dove
17  lexNum cs = TokenInt (read num) : lexer rest
18      where (num,rest) = span isDigit cs
19
```

```
20  lexVar cs =
21      case span isAlpha cs of
22          ("let",rest) -> TokenLet : lexer rest
23          ("in",rest) -> TokenIn : lexer rest
24          (var,rest) -> TokenVar var : lexer rest
```

11.2.2 Uso

I file Happy hanno lo stesso suffisso di Yacc, un esempio di file Happy è: `example.y`. Il comando `happy example.y` genera il file `example.hs`. Il comando `happy example.y -i` produce il file `example.info` contenente informazioni dettagliate sul parser.

11.2.3 Valutare la stringa in input

Come detto in precedenza, invece di generare l'albero di derivazione si può anche valutare la stringa. Quindi, al posto di restituire un'espressione, `calc` può valutare e restituire un intero, per esempio:

```
1  Term : Term '*' Factor   { $1 * $3 }
2         | Term '/' Factor { $1 / $3 }
3         | Factor          { $1 }
```

Le variabili e il costrutto `let` forzano però una valutazione più complessa:

- bisogna gestire l'ambiente `env` implementato come lista

  ```
  [(String, Int)] -- (nome variabile, intero associato)
  ```

- le regole associano ai non terminali una funzione `env -> Int`

Quindi le definizioni diventano:

```
1   -- bisogna aggiornare l'ambiente e restituirlo
2   Exp : let var '=' Exp in Exp { \p -> $6 (($2,$4 p):p) }
3         | Exp1 { $1 }
4
5   -- dato l'ambiente p, restituisce un valore
6   Exp1 : Exp1 '+' Term { \p -> $1 p + $3 p }
7         | Exp1 '-' Term { \p -> $1 p - $3 p }
8         | Term { $1 }
9
10  Term : Term '*' Factor { \p -> $1 p * $3 p }
11        | Term '/' Factor { \p -> $1 p `div` $3 p }
12        | Factor { $1 }
13
```

```
14  Factor
15      : int { \p -> $1 }
16      | var { \p -> case lookup $1 p of
17                        Nothing -> error "no var"
18                        Just i -> i }
19      | '(' Exp ')' { $2 }
```

La funzione `lookup` di riga 16 serve per vedere se all'interno dell'ambiente sia stata definita la variabile (quindi se si trova nella lista).

11.2.4 Gestione dell'ambiguità

Normalmente vogliamo usare una grammatica non ambigua, ma nel caso non lo sia possiamo usarla lo stesso definendo un ordine di precedenza tra gli operatori.

```
1   %right in
2   %nonassoc '>' '<'
3   %left '+' '-'
4   %left '*' '/'
5   %left NEG
6   %%
7   Exp : let var '=' Exp in Exp { Let $2 $4 $6 }
8         | Exp '+' Exp         { Plus $1 $3 }
9         | Exp '-' Exp         { Minus $1 $3 }
10        | Exp '*' Exp         { Times $1 $3 }
11        | Exp '/' Exp         { Div $1 $3 }
12        | '(' Exp ')'         { $2 }
13        | '-' Exp %prec NEG   { Negate $2 }
14        | int                 { Int $1 }
15        | var                 { Var $1 }
```

L'ordine con cui sono scritti gli operatori nelle prime 6 righe definisce la precedenza, l'operatore scritto più in basso ha la precedenza più alta.

11.2.5 Un tecnicismo

Supponiamo che il testo da analizzare sia su più righe, allora si può definire la grammatica come una linea oppure come un insieme di linee seguito da una singola linea, in questo caso si associa a sinistra:

```
1   lines : line          { [$1] }
2         | lines line    { $2 : $1 }
```

oppure può essere una linea o una linea seguita da un insieme di linee:

```
1   lines : line          { [$1] }
2         | line lines    { $1 : $2 }
```

Concettualmente sono molto simili, ma dal punto di vista dell'efficienza è meglio usare la prima soluzione, perché la seconda occupa più spazio.

Capitolo 12

Type Assignment System

12.1 Analisi semantica

Si tratta della terza fase del compilatore e consiste nell'eseguire una specifica analisi di coerenza sul codice, visto come albero sintattico. Tali controlli non sono verificabili attraverso le grammatiche e quindi necessitano di una fase ad hoc. L'analisi semantica è composta da:

- estrazione di informazioni per comporre la symbol table, nella quale associo gli identificatori di tipo
- esecuzione dei controlli non realizzabili con le grammatiche context free, come per esempio: numero dei parametri procedura, identificatore dichiarato prima di essere usato, cicli for non modificano la variabile di ciclo, etc...
- esecuzione del type checking

12.2 Type system

Un sistema di tipi viene definito informalmente nella documentazione, viene di solito implementato formalmente nel compilatore attraverso un algoritmo di controllo di tipo, tale algoritmo però può essere difficile da comprendere, la soluzione è usare una definizione formale.

12.2.1 Descrizione formale

L'idea è fornire un metodo per descrivere il sistema di tipi con maggiore formalità della documentazione ma più semplice da comprendere con l'algoritmo.

Si tratta di un'idea simile ai sistemi logici, con la differenza che in questo caso il giudizio è volto a dimostrare che il programma ha un certo tipo.

Si costruisce un sistema di derivazione che permette di derivare espressioni (chiamati *giudizi*) del tipo:

- $x_1 : A_1, x_2 : A_2, \ldots x_n : A_n \vdash M : A$

  Che dice che un pezzo di codice M ha tipo A se gli identificatori dello stesso hanno un certo tipo.

In questo modo a partire dal tipo degli identificatori presenti in un programma si riesce a stabilire il tipo delle sottoespressioni del programma.

Intuitivamente, a sinistra del $\vdash$ abbiamo una serie di ipotesi secondo le quali possiamo decidere che il tipo del programma M è A (parte a destra).

Nel caso di sistemi di tipi complicati si utilizzano dei *giudizi ausiliari*:

- $x_1 : A_1, x_2 : A_2, \ldots x_n : A_n \vdash A$

  Significa che il tipo dell'espressione A è di tipo corretto nell'ambiente:

  $x1 : A1, x2 : A2, \ldots xn : An$

- $x_1 : A_1, x_2 : A_2, \ldots x_n : A_n \vdash \cdot$

  Che significa che l'ambiente è ben formato.

Tali giudizi ausiliari non sono strettamente necessari dato che tali controlli possono essere eseguiti facilmente anche attraverso grammatiche libere.

12.2.2 Regole di derivazione

La definizione formale di tipi si riduce alla definizione di un sistema di regole per derivare dei giudizi, similmente a quello che succede nella deduzione in logica. Le regole per derivare sono della forma:

$$\frac{Gamma_1 \vdash M_1 : A_1 \ldots Gamma_n \vdash M_n : A_n}{Gamma \vdash M : A}$$

Dove $Gamma_1 \vdash M_1 : A_1 \ldots Gamma_n \vdash M_n : A_n$ sono dette premesse e $Gamma \vdash M : A$ è detta conclusione.
Se una regola non ha premesse allora è detta assioma.
Le regole poi possono essere definite per induzione sulla struttura di M, associando ad ogni costrutto del linguaggio una speciale regola di derivazione, inoltre con l'arricchirsi del linguaggio basta aggiungere nuove regole senza che quelle vecchie perdano di utilità.

L'approccio è quindi modulare nel caso della definizione delle regole, rimane abbastanza semplice definire le nuove regole all'aumentare della complessità ma un'analisi del sistema di derivazione diventa comunque più complesso.

12.2.3 Derivazioni

Le derivazioni hanno una struttura ad albero dove, da un insieme di assiomi (foglie), viene derivato un giudizio finale (radice). Permettono di derivare giudizi **validi**.

12.2.4 Type checking e type inference

Dato un sistema di tipi, consideriamo i due problemi di type checking e type inference:

- controllo di tipo, type checking: dato un termine M, un tipo A e un ambiente Γ, determinare se $\Gamma \vdash M : A$ sia valido
- inferenza di tipo, type inference: dato un termine M e un ambiente Γ, trovare un tipo A tale che $\Gamma \vdash M : A$ sia valido

L'analisi semantica risolve il problema della type inference che, specialmente in alcuni linguaggi come Python e Haskell dove è possibile non dichiarare i tipi delle variabili, può diventare molto complesso visto che bisogna capire prima anche il tipo degli identificatori da cui si parte.

12.2.5 Type soundness

La proprietà di type soundness serve per garantire che se si parte da un programma ben tipato certi tipi di errori non possano accadere. Non tutti gli errori sono evitabili con un controllo statico, alcuni controlli per forza di cose devono essere fatti a tempo di esecuzione. In generale un sistema di tipi forte dovrebbe garantire che la maggior parte degli errori non possano verificarsi se si parte da un programma ben tipato. Formalmente, se si parte da un programma ben tipato, si dimostra che la computazione preserva i tipi e si dimostra che una certa classe di errori non può essere generata da quel programma, allora si ha la garanzia formale dell'assenza di quegli errori nella computazione.

12.3 Esempi di sistemi di tipi: sistema F1

Possiamo vedere F1 come un frammento minimo di Haskell ma con tipi espliciti e privo di polimorfismo.

I tipi presenti sono un insieme di tipi base `K in Basic` e dei tipi funzione `A -> B`. Per la definizione del codice invece gli unici meccanismi presenti sono la definizione di funzioni anonime, tramite la lambda astrazione, e l'applicazione di una funzione ad un argomento. Gli elementi base del linguaggio possono essere dunque raggruppati in:

- variabili `x`

- costanti `c : K`
- funzioni `\ x:A . M`

  dove `x:A` impone un tipo all'argomento della funzione e `M` è il codice della funzione.
- applicazioni `M N`

  dove M è la funzione e N è l'argomento da sostituire al parametro formale

Data la definizione del linguaggio ora è possibile andare a definire le regole per la derivazione del tipo.

12.3.1 Regole per buona formazione e buon tipo

Le regole di "buona formazione" e "buon tipo" sono facilmente definibili attraverso una grammatica libera. Di seguito lo schema di regole:

Variabile (Var)

$$\Gamma,\ x : A,\ \Gamma' \vdash\ x : A$$

Che significa che se in un ambiente x ha tipo A allora si può concludere che x ha tipo A.

Il significato di $\Gamma,\ x : A,\ \Gamma'$ è semplicemente un modo di scrivere in forma abbreviata l'ipotesi, l'ambiente infatti conterrebbe una serie di variabili con un tipo associato e con questa scrittura si dice semplicemente che in questa serie di variabili è presente x con associato il tipo A.

Funzione (Fun)

$$\frac{\Gamma, x : A \ \vdash\ M : B}{\Gamma \ \vdash\ (\setminus x : A.M) : A \to B}$$

Il significato è che se nell'ambiente Γ con la variabile x di tipo A ($x : A$), riesco a stabilire che il corpo della funzione M ha tipo B ($M : B$), allora posso stabilire che la funzione che ha x come parametro e M come corpo ha tipo $A \to B$.

Applicazione (App)

$$\frac{\Gamma \ \vdash\ M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \ \vdash\ MN : B}$$

Tale regola afferma che se la funzione M ha tipo $A \to B$ e l'argomento N ha tipo A allora l'applicazione MN (funzione M chiamata con l'argomento

N) ha tipo B. Per ogni costante bisogna definire una regola che gli assegna il tipo.

12.3.2 Costruzione di una derivazione

La costruzione è top-down, a partire da un termine da tipare, selezioniamo la regola da applicare e riduciamo il problema a quello di tipare i sottotermini. Ad ogni passo abbiamo due problemi:

- selezionare la regola (dallo schema di regole) da applicare: per farlo è sufficiente osservare il costrutto principale del termine
- istanziare lo schema di regole: le regole generiche fanno riferimento a termini generici, bisogna istanziarli nel caso particolare, simile a meccanismi di pattern matching

12.3.3 Unit

Si tratta di un tipo base semplice con un unico elemento, analogo all'enupla vuota in Haskell:

$$\Gamma \vdash unit : Unit$$

12.3.4 Bool

Tipo con due soli valori costanti, true e false.

$$\Gamma \vdash true : Bool \qquad \Gamma \vdash false : Bool$$

Per il tipo Bool si può collegare la funzione IF THEN ELSE:

$$\Gamma \vdash (if_A \ \_ \ then \ \_ \ else \ \_) : Bool \to A \to A \to A$$

Oppure alternativamente:

$$\frac{\Gamma \vdash M : Bool \qquad \Gamma \vdash N_1 : A \qquad \Gamma \vdash N_2 : A}{\Gamma \vdash if_A \ M \ then \ N_1 \ else \ N_2 \quad : A}$$

Marcare il costrutto if con il tipo A (quindi if_A) significa esplicitare il tipo dei due argomenti N_1 ed N_2, semplificando l'Inferenza altrimenti più complessa.

12.3.5 Naturali

Sono possibili varie opzioni, se si vuole definire un linguaggio semplice si può fare utilizzando solo tre costanti:

- la costante 0:

$$\Gamma \vdash 0 : Nat$$

- la costante successore:

$$\Gamma \vdash succ : Nat \to Nat$$

- la costante predecessore:

$$\Gamma \vdash pred : Nat \to Nat$$

Da queste tre costanti è possibile ottenere ogni numero naturale, a dire il vero la funzione predecessore non è neanche strettamente necessaria ma in ogni caso può essere utile. Altre aggiunte per i naturali possono essere:

- la funzione `isZero`, che dato un numero naturale ritorna se è zero o no. $\Gamma \vdash isZero : Nat \to Bool$
- una regola di tipo per ogni naturale, per esempio: $\Gamma \vdash 1 : Nat$, $\Gamma \vdash 2 : Nat \ldots$
- le regole per le funzioni aritmetiche, per esempio la regola per la somma:

$$\frac{\Gamma \vdash N_1 : Nat \qquad \Gamma \vdash N_2 : Nat}{\Gamma \vdash N_1 + N_2 : Nat}$$

12.3.6 Tipo prodotto (o coppia)

Si tratta di un costruttore di tipi, il quale permette di costruire dei nuovi tipi a partire da quelli base. Di seguito prendiamo come esempio le enuple di due elementi (coppie) la cui forma è $A * B$ e la regola per il tipo è la seguente:

(Pair)

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A * B}$$

Le regole per il tipo dei distruttori sono:

(First)

$$\Gamma \vdash first : A * B \to A$$

(Second)

$$\Gamma \vdash second : A * B \to B$$

12.3.7 Tipo Unione

Anche questo si tratta di un costruttore di tipi e la notazione è $A + B$, le regole per i costruttori sono:

- inLeft
$$\Gamma \vdash inLeft : A \to A + B$$
- inRight
$$\Gamma \vdash inRight : B \to A + B$$

Le regole per i distruttori sono:

- isLeft
$$\Gamma \vdash isLeft : A + B \to Bool$$
- isRight
$$\Gamma \vdash isRight : A + B \to Bool$$
- asLeft
$$\Gamma \vdash asLeft : A + B \to A$$
- asRight
$$\Gamma \vdash asRight : A + B \to B$$

C'è però un problema nel trattare in questo modo il tipo union, se prendiamo l'espressione:

Possiamo evitare il type checking dinamico utilizzando il costrutto case:

$$\frac{\Gamma \vdash M : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash N_1 : B \qquad \Gamma, x_2 : A_2 \vdash N_2 : B}{\Gamma \vdash case\, M\, of\, (inLeft(x_1 : A_1) \to N_1)(inRight(x_2 : A_2) \to N_2) : B}$$

Esiste una sintassi alternativa per il case:

$$case\ M\ of\ x_1 : A_1\ then\ N_1 \mid x_2 : A_2\ then\ N_2$$

12.3.8 Tipi Struct

Possiamo costruire anche i tipi struct, non molto più complicati dei tipi enupla, infatti consiste in un tipo enupla con l'aggiunta di un'etichetta per ciascun elemento.

Il costruttore ha la seguente forma: `{l1 : A1, ... , ln : An}` e la regola è:

(Record)

$$\frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2 \qquad \dots \qquad \Gamma \vdash M_n : A_n}{\Gamma \vdash \{l_1 : M_1, \dots, l_n : M_n\} : \{l_1 : A_1, \dots, l_n : A_n\}}$$

(Record Select)

Per quanto riguarda il distruttore select la regola è la seguente:

$$\frac{\Gamma \vdash M : \{l_1 : A_1, \ldots, l_n : A_n\}}{\Gamma \vdash M.l_i : A_i}$$

Preso un record, si vuole esaminare una sua componente.

Esiste un meccanismo strano di utilizzo del record, alternativo a (Record Select), chiamato (Record With) che utilizza le etichette per ricorrere al tipo del record, la regola è la seguente:

$$\frac{\Gamma \vdash M : \{l_1 : A_1, \ldots, l_n : A_n\} \qquad \Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash N : B}{\Gamma \vdash case\ M\ of\ \{x_1 : A_1, \ldots, x_n : A_n\} \to N : B}$$

Si suppone di avere un M di tipo struct con un certo numero di campi (a sinistra), si usa poi la stessa lista di ipotesi cambiandogli il nome (per ipotesi intendiamo le variabili della struct) e usiamo quest'ultima per costruire l'espressione N (a destra), in seguito si costruisce l'espressione definita sotto, dove si valuta il valore di N che al suo interno può usare un x_n e in quel caso si utilizzerà il valore che M ha in quel campo.

12.3.9 Reference type

Si tratta dei tipi `ref` già descritti nei capitoli precedenti, a differenza dei tipi normali le variabili con tale tipo non contengono il valore bensì un riferimento ad un indirizzo di memoria con cui raggiungere il dato. Dato che lavorano con la memoria introducono uno stato e quindi nei linguaggi funzionali puri non sono presenti (per esempio Haskell), sono presenti però in ML.

(Ref)

Il costruttore di tipo è `Ref A` con la seguente regola:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash ref\ M : Ref\ A}$$

Dove `Ref M` definisce una nuova locazione inizializzata con valore M.

(Deref)

Da una `ref` è possibile accedere al contenuto:

$$\Gamma \vdash deref : (Ref\ A) \to A$$

Tale operazione corrisponde all'operatore ! in ML e * in C. Non c'è una regola ad hoc per il giudizio, usare l'applicazione trattandola come una funzione normale usando il tipo sopra descritto.

(Assign)

Un'altra operazione consiste nell'inserimento di un valore, la regola è:

$$\frac{\Gamma \vdash M : Ref\ A \qquad \Gamma \vdash N : A}{\Gamma \vdash M := N : Unit}$$

La presenza del tipo ref all'interno di un linguaggio funzionale permette la simulazione dei linguaggi imperativi:

```
var x = M; // dichiaro una variabile x con valore M
N          // espressione che usa x

// possiamo tradurlo nei linguaggi funzionali con
let x = ref M in N

// oppure, senza il let, con
(\ x . N) (ref M)
```

La locazione M definita dentro N permette poi l'inserimento di valori ed il recupero del dato da essa.
Inoltre possiamo tradurre la composizione di comandi `C1; C2` come:

```
(\ y : Unit -> C2) C1
```

Questo funziona nei linguaggi call-by-value, l'idea consiste nel valutare `C1` sfruttando i suoi eventuali effetti collaterali, possibili a cusa della presenza del tipo `ref`. Con l'argomento ritornato da `C1`, anche se non importante, valuto `C2`, permettendo quindi la valutazione sequenziale di `C1` e `C2`.

In generale quando si studiano i linguaggi imperativi è interessante analizzare come questi possano essere tradotti in un linguaggio funzionale con `ref`.

12.4 Altro esempio: linguaggio imperativo simil C

Consideriamo tre categorie sintattiche (nei funzionali sono tutte espressioni):

- espressioni

  ```
  E ::= const | id | E binop E | unop E
  ```

  costanti, identificatori, operazioni binarie e unarie

- comandi

  ```
  C ::= id = E | C; C | while E {C} | if E then C else C
      | I(E, ... E) | {D ; C}
  ```

assegnazione, composizione di comandi, ciclo e test, usare procedure passando argomenti (dopo averle definite), definizione di blocchi ({dichiarazione ; comando})

- dichiarazioni

```
D ::= A id = E | id(A1 id1, ... , An idn) { C } |
    epsilon | D; D
```

dichiarare variabili semplici (tipo, identificatore, espressione), procedure (nome procedura, lista parametri formali, corpo), epsilon indica una dichiarazione vuota (blocco vuoto), lista di dichiarazioni (separate da ;)

Avremo un sistema di tipo (e quindi regole) per ogni categoria sintattica.

12.4.1 Regole per le espressioni

Per le espressioni valgono le corrispondenti regole dei linguaggi funzionali.

(Id, (Var))

$$\Gamma,\ id : A,\ \Gamma' \vdash id : A$$

$$\Gamma \vdash true : Bool \qquad \Gamma \vdash false : Bool$$

(ite)

$$\Gamma \vdash (if\_then\_else\_) : Bool \to A \to A \to A$$

$$\Gamma \vdash 1 : Nat \qquad \Gamma \vdash 2 : Nat \qquad \Gamma \vdash 3 : Nat \qquad \ldots$$

Operazioni aritmetiche

$$\frac{\Gamma \vdash E_1 : Nat \qquad \Gamma \vdash E2 : Nat}{\Gamma \vdash E_1 + E_2 : Nat}$$

12.4.2 Regole per i comandi

I comandi hanno tutti tipo `Unit`. Il tipo non è importante (proprio perché è sempre `Unit`, ma il fatto che si possa assegnare il tipo tramite le regole assicura che il tipo è corretto.

(Assign)

Il tipo che si deve dedurre è `Unit`, ma si controlla che l'assegnamento sia fatto in modo corretto (stesso tipo).

$$\frac{\Gamma \vdash id : A \qquad \Gamma \vdash E : A}{\Gamma \vdash id = E : Unit}$$

(Sequence)

$$\frac{\Gamma \vdash C_1 : Unit \qquad \Gamma \vdash C_2 : Unit}{\Gamma \vdash C_1;\ C_2 : Unit}$$

(While)

$$\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash C : Unit}{\Gamma \vdash while\ E\ \{C\} : Unit}$$

(If Then Else)

$$\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash C_1 : Unit \qquad \Gamma \vdash C_2 : Unit}{\Gamma \vdash if\ E\ then\ C_1\ else\ C_2 : Unit}$$

(Procedure)

$$\frac{\Gamma \vdash id : (A_1 * \ldots * A_n) \to Unit \qquad \Gamma \vdash E_1 : A_1 \ldots \Gamma \vdash E_n : A_n}{\Gamma \vdash id\ (E_1, \ldots, E_n) : Unit}$$

(Blocco)

$$\frac{\Gamma \vdash D :: \Gamma_1 \qquad \Gamma,\ \Gamma_1 \vdash C : Unit}{\Gamma \vdash \{D; C\} : Unit}$$

La dichiarazione D crea l'ambiente Γ_1. Il comando verrà valutato sia con l'ambiente globale (Γ), sia con l'ambiente locale appena creato (Γ_1). Se ci sono più dichiarazioni, usare la regola sequenza di dichiarazioni.

12.4.3 Regole per le dichiarazioni

Le dichiarazioni necessitano di un nuovo giudizio. Una dichiarazione crea un ambiente che viene utilizzato nel blocco della dichiarazione. I giudizi sono della forma:

$$\Gamma \vdash D :: \Gamma_1$$

(Id)

Identificatori.

$$\frac{\Gamma \vdash E : A \qquad A \text{ tipo memorizzabile}}{\Gamma \vdash A\ id = E :: (id : A)}$$

(Proc)

$$\frac{\Gamma,\ id_1 : A_1,\ \ldots,\ id_n : A_n \vdash C : Unit}{\Gamma \vdash id(A_1\ id_1,\ \ldots,\ A_n\ id_n)\{C\} :: id : (A_1\ x \ldots\ x\ A_n) \to Unit}$$

(Recursive Proc)

$$\frac{\Gamma,\ id_1 : A_1, \ldots, A_n,\ id : (A_1 * \ldots * A_n) \to Unit \vdash C : Unit}{\Gamma \vdash id(A_1\ id_1, \ldots, A_n\ id_n)\{C\} :: id : (A_1\ x \ldots x\ A_n) \to Unit}$$

(Sequenza)

Dichiarazione come sequenza di dichiarazioni.

$$\frac{\Gamma \vdash D_1 :: \Gamma_1 \qquad \Gamma,\ \Gamma_1 \vdash D_2 :: \Gamma_2}{\Gamma \vdash D_1;\ D_2 :: \Gamma_1,\ \Gamma_2}$$

12.4.4 Regole per gli array

Nel caso degli array, negli assegnamenti non ci sono più solo gli identificatori a sinistra, ma possono anche esserci le celle dell'array. Bisogna distinguere tra ciò che sta a sinistra e ciò che sta a destra in un'assegnazione, quindi distinguiamo tra espressioni che possono indicare una locazione ed espressioni che possono indicare un valore:

- $LE ::= id \mid LE[RE]$

  le locazioni sono o un identificatore o una cella di un array, la cella può contenere a sua volta un altro array

- $RE ::= LE \mid const \mid RE\ binop\ RE \mid unop\ RE$

  i valori possono essere o una locazione (perché per esempio $A[5]$ può indicare il suo contenuto) o una costante o una qualsiasi espressione.

Per le espressioni distinguiamo due giudizi:

- $\Gamma \quad \vdash l \quad E : A$ (dove E denota una locazione di tipo A)
- $\Gamma \quad \vdash r \quad E : A$ (dove E denota un valore di tipo A)

Bisogna dunque riscrivere tutte le regole che riguardano le espressioni:

(Assign)

$$\frac{\Gamma \vdash l\ E_1 : A \qquad \Gamma \vdash r\ E_2 : A}{\Gamma \vdash l\ E_1 = E_2 : \mathit{Unit}}$$

Non è più identificatore uguale espressione, ma è espressione uguale altra espressione.

(Left-Right)

$$\frac{\Gamma \vdash l\ E : A}{\Gamma \vdash r\ E : A}$$

(Var)

$$\Gamma,\ x : A,\ \Gamma' \vdash l\ x : A$$

(Array)

Regola da aggiungere per accedere ad un elemento dell'array.

$$\frac{\Gamma \vdash l\ E : A[B] \qquad \Gamma \vdash r\ E_1 : B}{\Gamma \vdash l\ E[E_1] : A}$$

La parte a sinistra indica che un'espressione è di tipo array, ovvero $A[B]$ array di elementi di tipo A e con indici di tipo B. La parte a destra indica che il tipo degli elementi è B. Dunque se il tipo dell'indice corrisponde allora si può accedere al singolo elemento.

Dichiarazione

$$\overline{\Gamma \vdash id : A[B] :: id : A[B]}$$

Le restanti regole per le espressioni restano inalterate, diventando regole per giudizi right $\vdash r$.

Capitolo 13

Concorrenza

In questo capitolo andremo a parlare della concorrenza come feature aggiuntiva dei paradigmi esistenti, andando a catalogarne i vari aspetti sia nei linguaggi imperativi sia nei linguaggi funzionali (come Erlang e Haskell), questi ultimi avvantaggiati grazie all'assenza di stato.

13.1 Motivazioni della concorrenza

Le motivazioni che portano all'utilizzo della computazione concorrente è dovuta a varie ragioni:

- sfruttamento dell'hardware: con l'avanzare degli anni si è passato da incrementare le performance attraverso l'incremento della velocità dei processori, a incrementare il numero dei core e conseguente parallelismo, è necessario quindi scrivere programmi che sfruttino tale parallelismo.
- alcuni problemi vengono descritti meglio: ci sono dei compiti che vengono naturalmente gestiti meglio usando il parallelismo, come nei browser dove si può usare un thread per ogni parte della pagina da visualizzare, evitando per esempio che le immagini blocchino il caricamento del testo.
- permette di usare hardware distribuito e collegato via web

13.2 Livelli della concorrenza

La concorrenza può essere distinta in due livelli, **fisico** e **logico**.

13.2.1 Fisica

Nella concorrenza fisica diverse istruzioni vengono eseguite simultaneamente, implementata a livello hardware in diversi modi e con diversi livelli di granularità:

- pipeline
- superscalari (parallelismo a livello di istruzione)
- multicore (parallelismo a livello di processi)
- istruzioni vettoriali
- GPU (computazione vettoriale (SIMD))
- multicomputer
- reti di calcolatori

13.2.2 Logica

Si tratta della concorrenza a livello di programmazione:

- programmazione parallela: si cerca di parallelizzare l'esecuzione di un singolo problema, usato per esempio nel calcolo scientifico e nelle simulazioni. Per esempio si cerca di implementare un algoritmo in modo da sfruttare il parallelismo, che verrà a sua volta implementato con il multithread
- programmazione multithread: più thread o processi attivi contemporaneamente e che girano su una stessa macchina fisica che collaborano fra di loro di solito condividendo la memoria oppure scambiandosi messaggi
- programmazione distribuita: programmi concorrenti eseguiti su macchine separate che che collaborano comunicando attraverso messaggi, in questo caso non si assume che abbiano memoria condivisa

Esempio di programmazione distribuita: SOC e Cloud computing

L'idea del **cloud computing** è sfruttare macchine distribuite e collegate attraverso la rete. In genere i calcolatori mettono a disposizione delle funzionalità, che poi possono essere composte per ottenerne di più complesse. Un esempio tipico sono i web services.

13.2.3 Separazione tra i livelli di parallelismo

Il parallelismo logico e fisico non corrispondono, ovvero possono essere indipendenti l'uno dall'altro:

- nel caso del parallelismo fisico attraverso pipeline e processori superscalari il codice scritto viene parallelizzato indipendentemente dal fatto che sia sequenziale, si tratta quindi di un parallelismo invisibile al linguaggio di programmazione.

- nel caso si scriva un programma multithread questo può essere eseguito in modi diversi a seconda del tipo di parallelismo fisico presente:
  - ogni thread logico viene eseguito su core distinto
  - tutti i thread vengono eseguiti su un unico core (interleaving)
  - molti core ciascuno con molti thread in esecuzione

  Il risultato sarà lo stesso ma varieranno le prestazioni

13.3 Metodi per la programmazione concorrente

Per ottenere l'esecuzione concorrente la soluzione più standard è di utilizzare dei costrutti ad hoc, ma non è strettamente necessario, è infatti possibile ottenere la programmazione concorrente anche attraverso dei linguaggi standard. Questo è possibile utilizzando delle librerie per creare dei thread e passargli il codice da eseguire, un esempio è POSIX in C.

Una soluzione è utilizzare un compilatore ad hoc, il quale permetterà di eseguire in maniera parallela del codice sequenziale, questo è possibile se il compilatore è abbastanza furbo oppure se è possibile dargli delle indicazioni esplicite su quando è possibile parallelizzare, questo avviene in Fortran e OpenPM attraverso delle direttive (pragma).

13.3.1 Aspetti della programmazione concorrente

Sono presenti diversi modi per strutturare la concorrenza, in tutti sono presenti più thread (istruzioni attive), si distingue però su come avvengono:

- la comunicazione: scambio di informazioni
- la sincronizzazione: regolazione della velocità relativa

Un altro aspetto è come avviene la creazione dei thread.

13.3.2 Thread e processi

Per parlare di concorrenza è bene distinguere tra **thread** e **processi**:

- thread (del controllo): si tratta di una specifica computazione con spazio di indirizzamento comune
- processo (pesante): generico insieme di istruzioni in esecuzione, con il proprio spazio di indirizzamento, il quale può essere composto da più thread diversi

13.3.3 Creazione di nuovi thread

Esistono diversi modi per creare dei nuovi thread:

- fork, join: metodi classici per la creazione di thread, chiamando il metodo fork viene creato un nuovo thread.
- il costrutto `co-begin`: permette di mandare in esecuzione in maniera parallela più comandi
- loop paralleli: permettono di parallelizzare alcuni cicli for attraverso il compilatore.
- early replay: una procedura restituisce il controllo al chiamante prima della sua conclusione, chiamato e chiamante in esecuzione contemporanea

13.3.4 Meccanismi di comunicazione

Per comunicare i processi possono sfruttare diverse "strategie":

- **Memoria condivisa**: presuppone l'esistenza di una zona comune di memoria dove i diversi processi vanno a leggere e scrivere, la memoria comune può essere sia fisica, i processi scrivono sulla stessa locazione di memoria, oppure logica, ovvero la zona comune di memoria viene simulata attraverso delle librerie
- **Scambio di messaggi**: si utilizzano delle funzioni **send** e **receive** che permettono di inviare e ricevere dei messaggi, per utilizzare questa tecnica è necessario un canale di comunicazione che fornisca un percorso logico da mittente a destinatario
- **Blackboard**: si tratta di una versione intermedia fra le precedenti, si utilizza una parte di memoria condivisa su cui si può agire utilizzando chiamate send e receive

13.4 Meccanismi di sincronizzazione

I meccanismi di sincronizzazione permettono di controllare l'ordine relativo delle varie attività dei processi, essenziali nella concorrenza a causa del problema delle **race condition**.

13.4.1 Esempio di race condition

Un esempio di race condition è il caso di due processi che incrementano una variabile condivisa, dove l'ordine di interleaving causa anomalie:

```
1   //le istruzioni con "+" a sinistra sono del processo P1
2   //le istruzioni senza "+" a sinistra sono del processo P2
3   //supponiamo che x abbia valore n
4
5   + leggi x in reg1;  //reg1 = n
6     leggi x in reg2;  //reg2 = n
7     incrementa reg2;  //reg2 = n+1
8     scrivi reg2 in x; //x = n+1
9   + incrementa reg1;  //reg1 = n + 1
10  + scrivi reg1 in x; //x = n+1
11
12  //il valore finale di x e' n + 1, quindi e' incrementato di
        1
```

Le race condition non sono strettamente negative, molto spesso le varie combinazioni di ordine di esecuzione non causano problemi, solo alcuni casi vanno trattati in maniera specifica.

13.4.2 Meccanismi

Con memoria condivisa:

- **mutua esclusione**: ci sono delle sezioni critiche del codice che non sono accessibili contemporaneamente da più processi
- **sincronizzazione su condizione**: si sospende l'esecuzione di un thread fino al verificarsi di un'opportuna condizione. Per esempio se abbiamo un'operazione distribuita su più processi e thread, ogni thread aspetta la conclusione globale dell'operazione prima di procedere

Con scambio di messaggi non ci sono meccanismi di sincronizzazione espliciti perché è proprio lo scambio di messaggi che porta la sincronizzazione:

- **sincronizzazione implicita**: realizzata con le primitive di `send` e `receive`, si può ricevere solo dopo aver inviato

13.4.3 Implementazione della sincronizzazione

Per forza di cose un processo deve sospendersi e per fare ciò ci sono due modi:

- **attesa attiva** (busy waiting o spinning): è il più semplice, il processo entra in un loop in cui non fa nulla e aspetta che una condizione sia soddisfatta. Ha senso solo su multiprocessori, altrimenti ovviamente bloccherebbe gli altri processi.

- **sincronizzazione basata sullo scheduler**: interviene il sistema operativo che congela il thread

13.5 Sincronizzazione con memoria condivisa

13.5.1 Mutua esclusione mediante attesa attiva: lock

Associamo alla risorsa una variabile booleana che dice se è occupata oppure no e usiamo la funzione *atomica* (ovvero che non è interrompibile) `test-and-set(B)` per accederci: se è libera blocca la risorsa, altrimenti aspetta.

La struttura di un processo è la seguente:

```
process Pi {
    sezione non critica;
    acquisisci_lock(B);
    sezione critica;
    rilascia_lock(B);
    sezione non critica;
}
```

La funzione `acquisisci_lock` è definita come segue:

```
void acquisisci_lock(ref B: bool) {
    while test_and_set(B) do skip;
}
```

in cui `B` è la variabile booleana che vale `true` quando la risorsa è bloccata e `false` quando è libera. Si sfrutta un'istruzione di `read` e `write` atomica `test_and_set`.

La funzione `rilascia_lock` è definita come segue:

```
void rilascia_lock(ref B: bool) {
    B = false;
}
```

Algoritmi alternativi

La soluzione precedente ha problemi di *fairness* in quanto nel caso di più richieste l'assegnazione viene fatta arbitrariamente e quindi un processo potrebbe dover attendere all'infinito (starvation). Esistono algoritmi più complessi che non usano la funzione `test_and_set` e che assicurano la fairness, per esempio implementando delle priorità. Il problema di `test_and_set` è che un processo in attesa usa continuamente la memoria condivisa rallentando gli altri processi, la versione `test and test_and_set` migliora le prestazioni:

```
1  void acquisisci_lock(ref B: bool) {
2      while B do skip;
3      while test_and_set(B) do skip;
4  }
```

Il vantaggio è che si ferma su un'istruzione che fa solo la lettura ed è probabile che `B` sia in cache, mentre la versione precedente deve per forza accedere alla memoria principale bloccando il bus visto che deve anche scrivere.

13.5.2 Sincronizzazione su condizione: barriere

Il processo resta in attesa finché una condizione globale non viene soddisfatta. Esempio: un compito comune suddiviso tra più processi, ogni processo attende che tutti gli altri abbiano terminato prima di proseguire allo stadio successivo. Una possibile implementazione è introdurre un contatore che conta il numero di processi che devono ancora terminare. Si usa una sezione critica per aggiornare il contatore decrementando quando si ha finito e controllare quando diventa 0 per poter proseguire.

13.5.3 Problemi dell'attesa attiva

L'algoritmo di mutua esclusione presentato prima si basa sull'attesa attiva: il processo bloccato continua ad usare tempo di CPU, non viene sospeso esplicitamente. Conviene per attese brevi, ma è inefficiente su attese lunghe. Come già detto è poco sensato in sistemi a singolo processore.

13.5.4 Sincronizzazione basata sullo scheduler

Il processo che deve essere posto in attesa dal sistema operativo rilascia la CPU. I processi sono gestiti dallo scheduler dei processi (S.O.). La sincronizzazione con scheduler può essere implementata tramite semafori o tramite monitor.

Semafori

Il semaforo è un tipo di dato `sem` con l'insieme dei valori costituito dai numeri interi ≥ 0 e con due operazioni atomiche base chiamate `P` e `V`. Data la definizione:

```
sem s
```

la funzione:

```
P(s)
```

serve per l'accesso al semaforo:

- se `s > 0`, `s` viene decrementata (operazione atomica) e si può accedere alla sezione critica
- se `s = 0`, il processo viene sospeso

Una volta finite le operazioni nella sezione critica si usa la funzione:

```
V(s)
```

che serve per rilasciare il semaforo, nello specifico:

- `s` viene incrementata (operazione atomica)
- si attiva lo sblocco di eventuali processi in attesa, entrano in gioco le politiche di fairness

Esempio di uso dei semafori: filosofi a cena

Questo classico esempio illustra la possibilità di deadlock, ovvero quando per una serie di motivi tutti i processi sono in attesa di una qualche risorsa che però è occupata da un altro processo che a sua volta è in attesa. Il deadlock provoca un blocco del programma senza generare un errore. Nell'esempio dei filosofi a cena ogni filosofo è un processo che ciclicamente accede alle 2 forchette, mangia e le rilascia. Le forchette sono le risorse e possono essere simulate con un semaforo.

```
1  sem forchette[5];
2  for (i=1, i<=5, i+=1) {forchette[i] = 1};
3
4  process Filosofo[i]{  % i = 1, 3, 5
5      while true {
6          pensa;
7          P(forchette[i]); // acquisisce forchetta di destra
8          P(forchette[i-1]) // acquisisce forchetta di
               sinistra
9          mangia;
10         V(forchette[i]); // rilascia forchetta di destra
11         V(forchette[i-1]) // rilascia forchetta di sinistra
12     }
13 }
```

La soluzione consiste nel prendere prima le forchette in posizione dispari. Una volta ottenuta una forchetta dispari, un filoso può essere bloccato solo da un filosofo mangiante.

Monitor

L'idea è fare qualcosa di più strutturato rispetto ai semafori, che sono semplici contatori condivisi in cui la gestione corretta delle risorse condivise è affidata al programmatore. Ciò che fanno i monitor è inglobare la risorsa condivisa all'interno di un oggetto (monitor) e implementare i meccanismi di controllo tramite procedure pubbliche per l'uso delle risorse condivise, non si può accedere direttamente ad esse. Le caratteristiche di un monitor rispetto ad un oggetto normale sono che è condiviso tra più thread e che pone il vincolo di poter gestire solo una procedura alla volta (e quindi la mutua esclusione è garantita).

I monitor vengono considerati *componenti passivi* perché le sue procedure vengono eseguite solo se vengono invocate da qualcun altro.

All'interno dei monitor troviamo le *variabili condizionali* che rappresentano la disponibilità o meno di una risorsa condivisa. Permettono ad un thread di:

- entrare nel monitor
- esaminare lo stato della risorsa
- e, se necessario, mettersi in attesa che lo stato cambi liberando l'accesso al monitor

Possiamo dunque avere più processi all'interno del monitor, ma solo uno può essere attivo, gli altri sono in attesa.

Nei monitor la variabile condizionale è implementata nel seguente modo:

- dichiarazione

```
cond namevar
```

può essere vista come una coda dei thread in attesa, le operazioni su di essa sono:

- interrogazione per vedere se ci sono processi in attesa (e quindi vedere se è libera)

```
empty(namevar)
```

- sospensione, se la variabile è occupata

```
wait(namevar)
```

- rilascio della risorsa e conseguente risveglio di un processo in attesa

```
signal(namevar)
```

quindi in questo momento potenzialmente ci sono due processi attivi nel monitor: quello che ha segnalato e il processo appena risvegliato. Questo può essere risolto in due modi:

- il processo che segnala continua e si aspetta che esca dal monitor prima di risvegliare il secondo processo
- il processo che segnala si sospende ed aspetta che si concluda il processo risvegliato

la prima soluzione è la più logica, ma sappiamo che esiste anche la seconda

13.6 Sincronizzazione con scambio di messaggi

13.6.1 Sincronizzazione implicita

I processi non hanno memoria condivisa per comunicare, l'unico modo è lo scambio di messaggi. Con questa modalità non servono meccanismi di sincronizzazione poiché questa è implicita nell'invio e nella ricezione di messaggi (non possiamo ricevere senza che qualcuno abbia inviato). La comunicazione può essere:

- **sincrona**: chi trasmette riceve un acknowledgement da chi ha ricevuto. Le funzioni di send e receive sono bloccanti. Al più può esserci solo un messaggio nel canale di comunicazione (semplice da implementare). Gli errori vengono gestiti immediatamente. Più difficile evitare i deadlock.
- **asincrona**: chi trasmette non si aspetta una risposta e procede. Solo la funzione receive è bloccante. Se chi trasmette è più veloce di chi riceve è necessario un buffer dei messaggi inviati (implementazione più complicata). Gestione degli errori più complicata.

È possibile simulare la comunicazione sincrona con quella asincrona:

- dopo aver spedito il messaggio si attende (receive) un acknowledgement da parte del ricevente

e viceversa:

- si crea un processo buffer intermedio che implementa la coda di comunicazione, ricevendo ciclicamente i messaggi del mittente e mettendoli in coda e che contemporaneamente cerca di spedirli al processo ricevente

13.6.2 Meccanismi di naming

Meccanismi per specificare a chi inviare un messaggio o da chi ricevere. Il meccanismo più esplicito è costituito da due funzioni, send e receive, con cui specificare mittente e destinatario.
La funzione send ha due argomenti:

- processo a cui inviare
- dato inviato

La funzione receive ha anch'essa due argomenti:

- processo da cui ricevere
- variabile dove inserire il dato ricevuto

Porte

Un altro meccanismo è quello delle porte, usato nelle reti con il protocollo TCP. La comunicazione tra due processi può essere divisa in più canali logici associati a porte diverse, ogni dato viene inviato in uno specifico canale. Si definiscono degli identificatori di porta e ogni porta viene associata ad un tipo di comunicazione/servizio. Si specifica l'indirizzo IP e la porta per identificare univocamente una comunicazione.

Esempio di utilizzo delle porte in Ada

Ogni task corrisponde ad un thread/processo. Quando si definisce un task si definisce l'interfaccia del processo che contiene la lista delle porte con cui esso può interagire.

```
1  task TypeTask is
2      entry portaIn (dato : in integer);
3      entry portaOut (dato : out integer);
4  end TypeTask
```

le porte diventano molto simili ad una chiamata di procedura

```
1  task body TypeTask is
2      ...
3      accept portaIn(dato : in integer) do ... end portaInf
4      ...
5      accept portaOut(dato : out integer) do ... end portaInf
6      ...
7  end TypeTask
```

Canali

I canali sono concettualmente simili alle porte, ma più generali e astratti, non si fanno ipotesi sul numero di processi coinvolti o sulla direzione dei dati.

Dichiarazione di un canale:

```
channel nomeCh (Type)
```

Per inviare i dati si specifica il nome del canale e i dati, informazioni specifiche sul canale vengono date in altre sezioni del codice.

```
send NomeCanale(dati)
```

Per ricevere specifichiamo una variabile dove inserire il dato ricevuto:

```
receive NomeCanale(var)
```

Ci sono più tipi di canali in base a:
verso

- monodirezionali
- bidirezionali

numero di connessioni

- link (un mittente - un destinatario)
- input port (più mittenti - un destinatario)
- mailbox (più mittenti - più destinatari)

sincronizzazione

- sincroni
- asincroni

Esempio di interazione tra client e server

```
1  channel richiesta (int client, char dati);
2  channel risposta1 (char ris);
3  channel risposta2 (char ris);
4
5  process Client1{
6      char valori;
7      char risultati;
8      .... // Definizione dei valori
9      send richiesta(1, valori);
```

```
10      receive risposta1(risultati};
11      .... // Uso dei risultati
12  }
13
14  process Client2{
15      char valori;
16      char risultati;
17      .... // Definizione dei valori
18      send richiesta(2, valori);
19  }
20
21  process Server{
22      int cliente;
23      char valori;
24      char risultati;
25      .... // Inizializzazione
26      while true {
27          receive richiesta (cliente, valori);
28          if cliente = 1 then
29              {
30              ... // Elaborazione valori
31              send risposta1(risultati);
32              }
33              if cliente = 2 then
34                  {
35                  ... // Elaborazione valori
36                  send risposta2(risultati);
37                  }
38      }
39  }
```

Un'alternativa a questa soluzione è passare il nome di un canale al posto dell'identificativo di processo, questo permetterebbe una comunicazione privata: il client invia un nome privato di canale su cui ricevere la risposta. Inoltre questa soluzione permetterebbe al server di gestire un numero arbitrario di processi in modo semplice.

La receive è bloccante, per evitare i blocchi si possono usare delle primitive per controllare lo stato della coda di input e nel caso decidere di fare altro invece di restare in attesa.

13.6.3 RPC

Un ulteriore meccanismo di comunicazione è RPC, remote procedure call, utilizzata nei sistemi distribuiti e client server. Consiste nel chiamare

una procedura, con relativi argomenti, in una macchina remota, è necessario quindi che tale macchina abbia dei meccanismi per rendere pubbliche alcune procedure, spesso ciò avviene tramite i moduli:

```
call NomeModulo.NomeProcedura(parametriAttuali)
```

La macchina remota deve inoltre avere dei meccanismi per attivare dei processi in caso di arrivo della richiesta, possibile in Java e realizzabile tramite **stub**, ovvero un processo locale al programma che si occupa di eseguire l'effettiva comunicazione con la macchina remota.

Se nella macchina remota esiste già un processo attivo che gestisce la risposta, questo viene chiamato **rendez-vous**, è disponibile in ADA attraverso le porte dove a ciascuna di esse è associato del codice, quindi l'interrogazione di una di questa porta all'esecuzione di quest'ultimo.

```
1  task body TypeTask is
2      ...
3      accept portaIn(dato : in integer) do ... end portaInf
4      ...
5      accept portaOut(dato : out integer) do ... end portaInf
6      ...
7  end TypeTask
```

13.7 Non determinismo

Tutto il codice visto fino adesso era deterministico, con la concorrenza però si introduce del non determinismo, dovuto alle race condition; è quindi utile introdurre dei costrutti non deterministici, di cui ne sono esempio i comandi con guardia.

13.7.1 Comandi con guardia

I comandi con guardia sono un costrutto non deterministico, consistono in una serie di comandi preceduti da una guardia, viene valutata la guardia di ciascun comando ed in base al risultato di tali valutazioni possono accadere tre cose:

- nussuna guardia ritorna true: il comando fallisce
- una sola guardia ritorna true: il comando associato viene eseguito
- più di una guardia ritorna true: un solo comando, scelto casualmente, viene uno a caso (l'idea è che la scelta di quale eseguire non sia deterministica)

```
guardia -> comando
[]
guardia -> comando
[]
...
[]
guardia -> comando
```

Un possibile uso è quello di evitare le scelte:

```
x <= y -> max = y
[]
y <= x -> max = x
```

Uno degli usi più importanti però consiste nell'utilizzo di tale costrutto per gestire la concorrenza eseguendo nella guardia anche un controllo sullo stato di un ipotetico canale evitando così di usare lo scheduler.

13.7.2 Esempio: server con due client

In questo esempio vogliamo che un server gestisca le richieste di scrittura e lettura di due client diversi.

La soluzione banale è utilizzare un ordinamento delle richieste, che può portare però ad una situazione di deadlock, per evitare ciò utilizziamo i comandi con guardia.

```
channel lettura (int dati);
channel scrittura (int dati);

process ClientLettura{
    int risultati;
    ....
    send lettura(risultati);
    .... // Uso di risultati
}

process ClientScrittura{
    int valori;
    .... // Definizione di valori
    send scrittura(valori);
    ....
}

process Server{
    int v;
```

```
20      int r;
21      while true{
22          receive lettura(v) -> servi richiesta lettura
23          []
24          receive scrittura (r) -> servi richiesta scrittura
25      }
26  }
```

13.7.3 Composizione parallela

Si tratta di un meccanismo astratto per la definizione di thread:

```
P || Q || R
```

Dove tale scrittura indica l'esecuzione parallela dei processi `P`, `Q` e `R`, con implementazione che può essere:

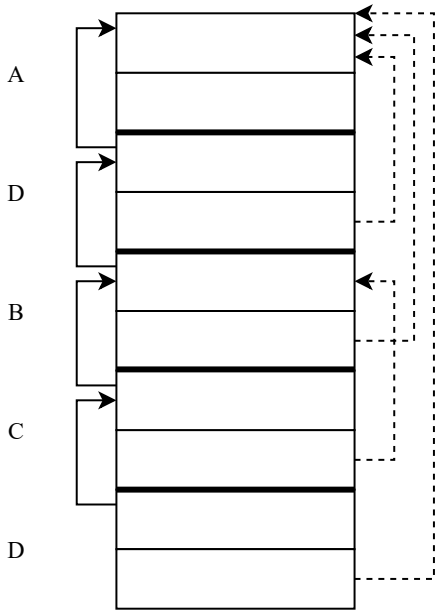
- **true cuncurrency**: parallelismo vero e proprio, ovvero più processi eseguiti nello stesso istante
- **interleaving**: dove si simula il parallelismo alternando le istruzioni dei vari processi.

Appendice A

Esercizi

Esercizio 1 *Considerato il seguente frammento di programma, con scope statico, mostrare la struttura dei record di attivazione, con i link statici e dinamici, dopo la sequenza di chiamate di funzione* `A(), D(), B(), C(), D()`.

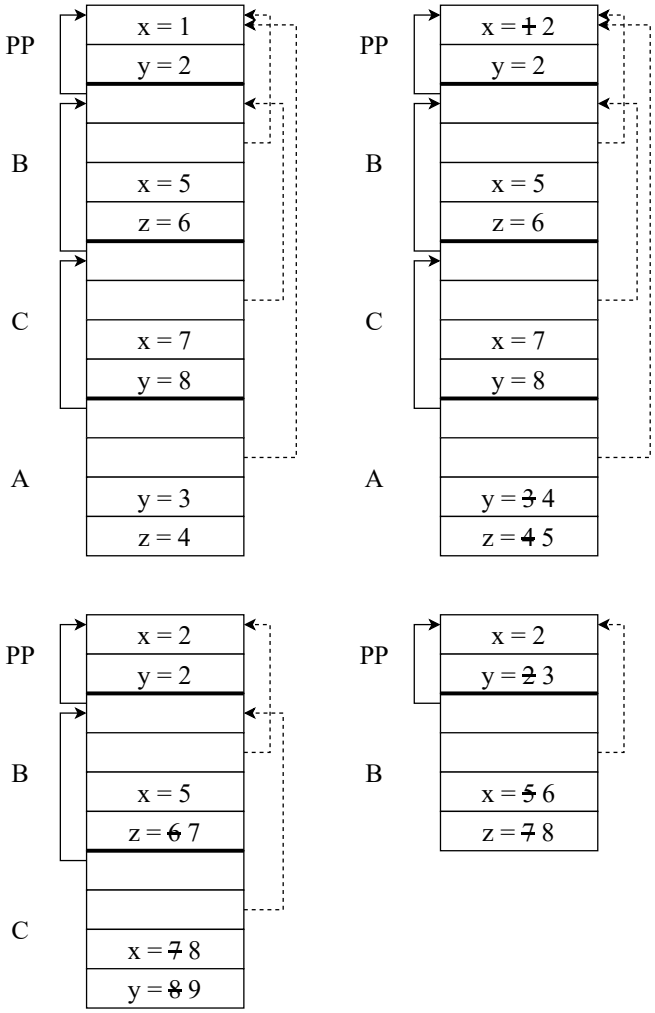
```
1  void A(){
2      void B(){
3          void C(){ }
4      }
5      void D(){ }
6  }
```



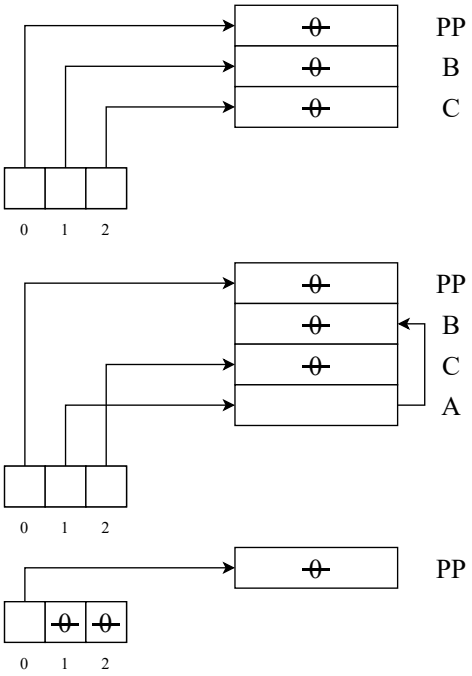
Esercizio 2 *Mostrare l'evoluzione dello stack di attivazione del seguente codice nei diversi meccanismi di scope (statico, dinamico), e di implementazione (catena statica, display, A-List, CRT).*

```
1   int x = 1, y = 2;
2   void A(){
3       int y = 3, z = 4;
4       x++; y++; z++;
5   }
6   void B(){
7       int x = 5, z = 6;
8       void C(){
9           int x = 7, y = 8;
10          A();
11          x++; y++; z++;
12      }
13      C();
14      x++; y++; z++;
15  }
16  B();
```

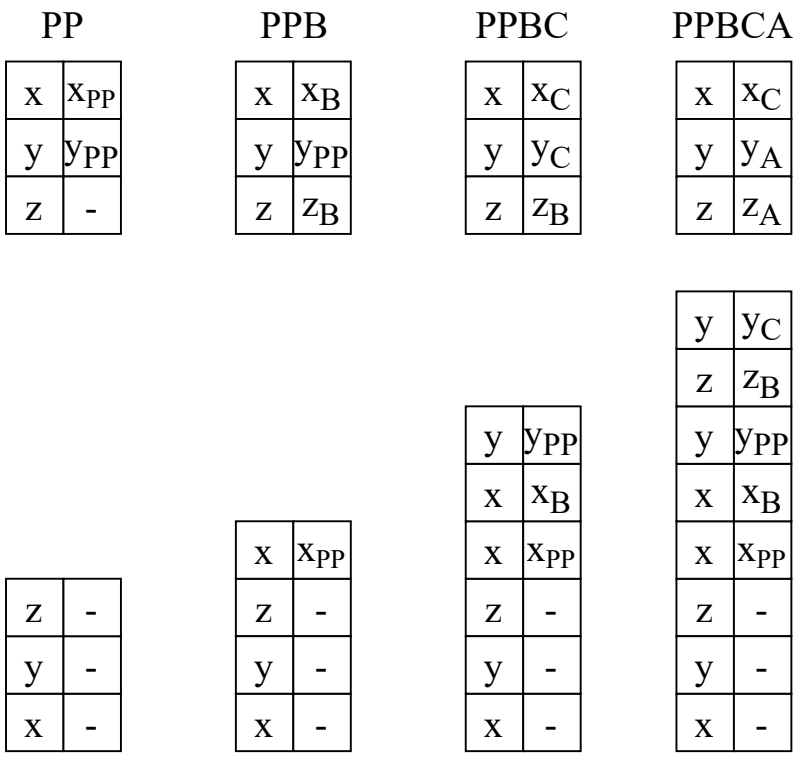
2.1 Scope statico normale



2.2 Scope statico con display



2.3 Scope dinamico con CRT e stack dei rif. nascosti



Esercizio 3 *Esercizio sui sistemi di tipi funzionali, esercizio 5.A dell'esame del 9 settembre 2019*

In questo esercizio è richiesto di costruire la derivazione di tipo di un'espressione, di seguito proveremo a spiegare come si fa. L'espressione data è la seguente:

$$\backslash f : (Nat \to Bool) \to (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)$$

Come primo passo scriviamo l'espressione con ipotesi nulla, questo siccome all'inizio non abbiamo ipotesi sull'espressione, inoltre inizialmente il tipo dell'espressione ci è ignoto.

$$\vdash \backslash f : (Nat \to Bool) \to (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)\ : ?$$

L'espressione è una dichiarazione di funzione, applichiamo la regola corrispondente.

$$\frac{f : (Nat \to Bool) \vdash (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)\ : ?}{\vdash \backslash f : (Nat \to Bool) \to (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)\ : ?}$$

Tra le ipotesi è comparso il tipo dell'argomento della funzione, se guardiamo a destra della deduzione si nota che è presente un'altra definizione di funzione, riapplichiamo la regola.

$$\frac{\dfrac{f : (Nat \to Bool), n : Nat \vdash if_{Bool}\ (f\ n)\ then\ false\ else\ true\ : ?}{f : (Nat \to Bool) \vdash (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)\ : ?}}{\vdash \backslash f : (Nat \to Bool) \to (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)\ : ?}$$

Il prossimo passo consiste nell'applicare la regola per l'if then else, per chiarezza con Γ indicheremo le ipotesi su f e n.

($\Gamma = f : (Nat \to Bool), n : Nat$)

$$\frac{\dfrac{\dfrac{\Gamma \vdash f\ n : Bool \qquad \Gamma \vdash false : Bool \qquad \Gamma \vdash true : Bool}{f : (Nat \to Bool), n : Nat \vdash if_{Bool}\ (f\ n)\ then\ false\ else\ true\ : ?}}{f : (Nat \to Bool) \vdash (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)\ : ?}}{\vdash \backslash f : (Nat \to Bool) \to (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)\ : ?}$$

L'ultima cosa da fare è applicare la regola per l'applicazione $f\ n$.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : (Nat \to Bool) \qquad \Gamma \vdash n : Nat}{\Gamma \vdash f\ n : Bool} \qquad \Gamma \vdash false : Bool \qquad \Gamma \vdash true : Bool}{f : (Nat \to Bool), n : Nat \vdash if_{Bool}\ (f\ n)\ then\ false\ else\ true\ : ?}}{f : (Nat \to Bool) \vdash (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)\ : ?}}{\vdash \backslash f : (Nat \to Bool) \to (\backslash n : Nat \to if_{Bool}\ (f\ n)\ then\ false\ else\ true)\ : ?}$$

Le ipotesi affermano che f è di tipo $(Nat \to Bool)$ e n è di tipo Nat quindi, secondo la regola dell'applicazione, possiamo concludere che $f\ n$ è di tipo $Bool$. Adesso possiamo scendere e completare ogni ? con il tipo dedotto secondo la regola applicata fino ad arrivare all'espressione iniziale il cui tipo è:

$$(Nat \to Bool) \to Nat \to Bool$$

Nel caso dei tipi imperativi il procedimento è analogo, basta applicare le relative regole.

Appendice B

Domande esame

Raccolta di tutte le domande di teoria proposte negli esami dal 18-2 al 19-4.

Domanda 1 *Quali sono i difetti del comando GOTO?*

Il comando GOTO permette, e in qualche modo incoraggia, la scrittura di codice poco strutturato, di difficile comprensione e dove eventuali errori sono difficilmente diagnosticabili.

Domanda 2 *Descrivere la struttura di una definizione di type class in Haskell.*

Nelle definizione di type class viene definita la lista delle funzioni, con relativi tipi, che è necessario fornire per inserire un tipo in una type class. Opzionalmente, vengono definite sovra-classi e funzioni derivate.

Domanda 3 *Cosa distingue una comunicazione sincrona da una asincrona?*

Nella comunicazione sincrona, send e receive devono avvenire quasi contemporaneamente, nella asincrona no. Come conseguenza, la send sincrona è bloccante, e la send asincrona deve essere supportata da un buffer di dimensione arbitraria.

Domanda 4 *Cosa si intende per pragmatica di un linguaggio di programmazione?*

La pragmatica descrive come un particolare linguaggio di programmazione viene usato; consiste in una serie di esempi, norme e convenzioni su come vada scritto del buon codice.

Domanda 5 *Quali controlli sul codice vengono svolti dall'analisi semantica?*

Durante l'analisi semantica vengono svolti tutti quei controlli sulla struttura del codice che non possono essere espressi con una grammatica libera dal contesto. Principalmente controllo di tipi, ma anche controlli per esempio numero degli argomenti usati per una funzione.

Domanda 6 *Cosa sono le funzioni di ordine superiore?*

Le funzioni di ordine superiore sono funzioni che hanno come argomenti, o che restituiscono come risultato, altre funzioni.

Domanda 7 *Cosa si intende per valutazione corto-circuito in un'operazione booleana?*

Per valutazione corto-circuito si intende quando valutare solo una parte dell'espressione è sufficiente per determinarne il risultato. Per esempio se si valuta `a && b` e `a` è falsa, sappiamo subito che l'espressione sarà falsa.

Domanda 8 *Cosa prevede il principio dell'incapsulamento?*

Il principio di incapsulamento prevede di rendere visibili all'esterno solo una parte degli attributi e dei metodi, nascondendo l'effettiva implementazione. In questo modo, due implementazioni corrette sono indistinguibili dall'esterno.

Domanda 9 *Nella gestione dello scoping statico, cosa si intende per catena statica?*

La catena statica è la lista dei link statici, ovvero i puntatori ai record di attivazione degli avi della procedura attiva.

Domanda 10 *Cos'è il dope vector?*

Il dope vector è la descrizione di un array contenente le seguenti informazioni:

- dimensione di un elemento
- puntatore all'inizio dell'array
- limiti dell'indice
- numero di dimensioni

Domanda 11 *In un linguaggio ad oggetti cosa distingue i metodi pubblici, da quelli protetti?*

I metodi pubblici sono visibili all'esterno della classe, mentre i metodi protetti sono visibili solo alla classe stessa e alle classi figlie e/o (in base allo specifico linguaggio) anche alle classi appartenenti allo stesso modulo.

Domanda 12 *Che tipo di linguaggi vengono riconosciuti dagli analizzatori sintattici?*

I linguaggi generati da grammatiche libere dal contesto.

Domanda 13 *Cos'è il Frame Pointer?*

Il frame pointer è il puntatore che punta all'inizio dell'ultimo record di attivazione all'interno dello stack di attivazione.

Domanda 14 *Cos'è una dangling reference?*

È quando un puntatore punta alla locazione di memoria di un oggetto che però è stata precedentemente deallocata senza eliminare il puntatore in questione.

Domanda 15 *In cosa l'aritmetica dei puntatori differisce dall'aritmetica standard? Si mostri un esempio in C in cui l'aritmetica dei puntatori differisce dall'aritmetica standard.*

L'aritmetica dei puntatori differisce da quella standard per il fatto che gli incrementi sono moltiplicati per la dimensione del tipo puntato. Per esempio il seguente codice:

```
a + 3
```

incrementa il puntatore `a` di `3 *` (dimensione tipo puntato) invece che di `3`.

Domanda 16 *Come si caratterizza l'ereditarietà singola? Come si caratterizza l'ereditarietà multipla?*

L'ereditarietà singola permette ad una classe di ereditare solo da una superclasse. Il vantaggio è che non si presenta il problema del name clash.
L'ereditarietà multipla invece permette ad una classe di ereditare da più superclassi contemporaneamente, semplificando il riutilizzo del codice, ma si può presentare il problema del name clash.

Domanda 17 *Cos'è una remote procedure call (RPC)?*

L'RPC consiste nel chiamare una procedura in una macchina remota che deve quindi esporre all'esterno la procedura ed avere dei meccanismi per rilevare e rispondere alla richiesta.

Domanda 18 *In un linguaggio imperativo, cosa sono i valori memorizzabili?*

I valori memorizzabili sono i valori che si possono inserire in una locazione di memoria.

Domanda 19 *Per quali operazioni i linguaggi regolari sono chiusi?*

Il linguaggi regolari sono chiusi rispetto alle operazioni di: unione, concatenazione, chiusura di Kleene, complementazione, intersezione.

Domanda 20 *In quale parte della memoria viene memorizzato un array?*

Se la shape è statica, si memorizza nel RdA, se è determinata al momento della dichiarazione si memorizza il dope vector nella parte iniziale del RdA e l'array vero e proprio nella parte finale, se la forma è dinamica invece si memorizza tutto nella heap.

Domanda 21 *Quali caratteristiche deve avere un linguaggio di programmazione affinché le funzioni siano considerate oggetti di secondo livello?*

Affinché le funzioni siano considerate oggetti di secondo livello, il linguaggio di programmazione deve permettere alle funzioni di prendere come argomento altre funzioni.

Domanda 22 *Cosa si intende per scoping dinamico?*

Con scoping dinamico si intende che, per accedere ad una variabile non locale, si segue la catena dinamica, ovvero la lista dei puntatori alle funzioni chiamanti (link dinamici) nello stack di attivazione.

Domanda 23 *Quali sono le caratteristiche del polimorfismo di sottotipo? Quali sono le caratteristiche del polimorfismo parametrico?*

Il polimorfismo di sottotipo permette di inserire il sottotipo in un contesto dove è richiesto il sovratipo.

Il polimorfismo parametrico permette di definire funzioni che accettano qualsiasi tipo. Il vantaggio è che non è necessario ridefinire una funzione per ogni tipo diverso.

Domanda 24 *Quali sono le caratteristiche del passaggio dei parametri per riferimento?*

Nel passaggio dei parametri per riferimento viene passato un puntatore all'oggetto specificato nella chiamata alla procedura, per cui ogni modifica ad esso fatta all'interno della procedura si ripercuote anche sull'oggetto iniziale. Questa modalità comporta una maggiore efficienza nel caso di passaggio di strutture dati di grande dimensione in quanto non bisogna crearne una copia.

Domanda 25 *A cosa servono le eccezioni? Cosa accade quando viene generata un'eccezione?*

Le eccezioni servono a gestire errori o situazioni non previste tramite un codice ad hoc.
Nel momento in cui viene generata un'eccezione si cerca il suo gestore, se non si trova nel blocco corrente si esce da ogni procedura chiamata finché non viene trovato, al limite si arriva al programma principale e se ancora non viene trovato si termina il programma mandando un segnale d'errore.

Domanda 26 *Nella gestione dello scoping statico, cosa si intende per display?*

Il display è un array in cui la posizione corrisponde con la profondità statica del programma (per esempio la cella numero 1 corrisponde alla profondità statica 1) in cui si salvano i puntatori all'ultimo record di attivazione della procedura attiva in quel livello, ovvero quelle che sono attive e visibili in un dato istante.

Domanda 27 *Cos'è il P-code?*

Il P-Code è il codice intermedio del Pascal ottenuto con la prima fase della compilazione. Un codice intermedio è un codice non specifico per un particolare calcolatore.

Domanda 28 *Nella memoria, che funzione svolge l'heap?*

La heap serve a memorizzare i dati dinamici, ovvero quelli che possono cambiare dimensione a run time.

Domanda 29 *Quali sono le caratteristiche di un strong type system? Quali sono le caratteristiche di un weak type system?*

Con un strong type system quasi tutti gli errori di tipo vengono rilevati e segnalati, mentre un weak type system non segnala alcuni errori di tipo che potrebbero causare problemi più avanti durante l'esecuzione.

Domanda 30 *Cosa sono i tipi intervallo? Cosa sono i tipi enumerazione?*

Un tipo enumerazione è un tipo definito dall'utente in cui si elencano tutti i valori che il suddetto tipo può assumere.

I tipi intervallo sono dei sottoinsiemi di un tipo base ordinabile, come per esempio i numeri da 0 a 10 che sono un sottoinsieme del tipo intero.

Domanda 31 *Nella programmazione concorrente, cosa sono i canali?*

I canali sono un mezzo di comunicazione tra più processi/thread (simili alle porte, ma più astratti), accessibili con le primitive `send` e `receive` per lo scambio di messaggi. Ci sono più tipi di canali in base a: verso, numero di connessioni e sincronizzazione.

Domanda 32 *Si mostri un esempio in C in cui l'aritmetica dei puntatori differisce dall'aritmetica standard?*

```
1  int *a = ...;
2  int b = ...;
3
4  a + 3;
5  b + 3;
```

Il risultato di `a + 3` è `a + 3 * (dimensione int)`, mentre `b + 3` somma semplicemente il valore `3`.

Domanda 33 *A cosa servono le dichiarazioni di import-export nei moduli.*

Le dichiarazioni di import servono a scegliere quali moduli e quali definizioni esposte dal modulo importare nel programma corrente, mentre le dichiarazioni export servono a specificare che cosa esporre all'esterno del modulo corrente.

Domanda 34 *Cos'è una grammatica libera da contesto?*

Una grammatica libera dal contesto è una quadrupla costituita da: simboli terminali, simboli non terminali, insieme di regole (produzioni) e simbolo iniziale, che permette di definire dei linguaggi liberi dal contesto.

Domanda 35 *Nello scoping statico, come si determina il binding valido?*

Se la variabile non è locale bisogna risalire la catena statica finché non la si trova.

Domanda 36 *Quali controlli sulla sintassi non possono essere svolti attraverso le grammatiche libere?*

I seguenti controlli non possono essere svolti con le grammatiche libere:

- estrazione di informazioni per comporre la symbol table, nella quale associo gli identificatori di tipo

- numero dei parametri procedura, identificatore dichiarato prima di essere usato, cicli for non modificano la variabile di ciclo, indice di matrice fuori limite, etc...
- esecuzione del type checking

Domanda 37 *Cosa sono i semafori?*

I semafori sono un un tipo di dato per gestire la concorrenza, nella pratica i processi usano due operazioni atomiche sulla variabile per vedere se la risorsa è libera: se il semaforo è maggiore di 0 allora bloccano la risorsa e decrementano il semaforo, se invece è 0 rimangono in attesa. Quando devono rilasciare la risorsa incrementano di nuovo il semaforo.

Domanda 38 *Cos'è la programmazione strutturata?*

La programmazione strutturata è una metodologia di programmazione che segue le best practice:

- progettazione gerarchica, top–down (il problema viene scomposto in sottoproblemi)
- modularizzare il codice (mantenere la struttura top–down anche nel codice, un sottoprogramma per un sottoproblema)
- uso di nomi significativi (migliore manutenibilità)
- uso estensivo dei commenti
- tipi di dati strutturati
- uso dei costrutti strutturati per il controllo
  - ogni costrutto, pezzo di codice, ha un unico punto di ingresso e di uscita
  - le singole parti della procedura sono modularizzate
  - i diagrammi di flusso non sono più necessari, in quanto il programma si auto-descrive poiché strutturato

Domanda 39 *Cos'è un interprete di un linguaggio di programmazione?*

L'interprete di un linguaggio di programmazione è un programma che riceve in input il codice e l'input dell'utente e traduce passo passo le singole istruzioni, producendo l'output.

Domanda 40 *Il programma YACC che tipo di automi genera?*

Genera degli automi a pila LALR.

Domanda 41 *Cosa si intende per scoping statico?*

Per scoping statico si intende che la ricerca di una variabile non locale avviene seguendo la catena statica, ovvero la lista dei puntatori agli antenati della procedura attiva così come sono definiti nel codice.

Domanda 42 *Cosa si intende per equivalenza strutturale tra i tipi?*

Due tipi sono equivalenti strutturalmente se l'unfolding delle loro definizioni porta alla stessa espressione di tipo.

Domanda 43 *In un linguaggio imperativo, cosa sono i valori denotabili?*

I valori denotabili sono i valori a cui è associabile un nome.

Domanda 44 *Cosa produce l'analisi lessicale di un programma?*

L'analisi lessicale produce lo stream di token, generato a partire dai lessemi, che verrà dato in input alla fase di analisi sintattica.

Domanda 45 *Qual è la differenza tra "coercion" e "casting"?*

La coercion è una conversione implicita del tipo fatta dal compilatore, mentre il casting è una conversione di tipo esplicitata nel codice del programma. Per esempio nella somma tra Int e Float, al tipo Int può essere applicata una conversione implicita in Float tramite coercizione oppure eplicita nel codice tramite casting.

Domanda 46 *Quali sono le caratteristiche del passaggio dei parametri per nome?*

Nel passaggio dei parametri per nome, l'espressione che denota il parametro attuale non viene valutata finché non viene usato il parametro attuale. I problemi sono la cattura delle variabili, risolvibile passando anche l'ambiente di valutazione dell'espressione, e la valutazione multipla, risolvibile valutando una volta sola l'espressione salvando il risultato e riutilizzandolo ogni volta che bisogna valutare il parametro formale.

Domanda 47 *Cosa sono le espressioni regolari?*

Un'espressione regolare è una sequenza di simboli che identifica le stringhe generate da un linguaggio regolare.

Domanda 48 *Cosa si intende per selezione dinamica dei metodi?*

Per selezione dinamica dei metodi si intende la scelta tra più metodi in caso di omonimia, per esempio quando una classe eredita due metodi con lo stesso nome da due super classi, oppure quando viene eseguito l'overriding dei metodi.

Domanda 49 *A quale scopo vengono introdotte le definizioni di type class in Haskell?*

Le definizioni di type class hanno lo scopo di migliorare il polimorfismo parametrico ponendo delle condizioni sui tipi di dato che le usano, facendo in modo che implementino sempre determinate operazioni. È così possibile definire delle funzioni che accettano solo tipi di dato che appartengono ad una type class specifica.

Domanda 50 *Cosa si intende per semantica di un linguaggio di programmazione?*

La semantica è l'insieme dei significati da attribuire alle frasi (sintatticamente corrette) costruite nel linguaggio scelto.

Domanda 51 *Descrivere le regole dello scoping statico.*

Nel caso dello scoping statico, per i nomi non locali, ci si riferisce sempre alla struttura statica del codice, ovvero com'è scritto al momento della compilazione.

Domanda 52 *Nello stack di attivazione, cos'è il link statico?*

Il link statico è il puntatore all'ultimo record di attivazione della procedura genitore, ovvero quella che contiene la dichiarazione della procedura in esecuzione.

Domanda 53 *Nella rappresentazione delle espressioni, come viene definita la notazione polacca inversa?*

La notazione polacca inversa è una notazione postfissa che ha la caratteristica di non necessitare di parentesi se l'arietà delle funzioni è prefissata, per esempio:

```
a b c * +
```

esegue la somma tra `a` e la moltiplicazione tra `b` e `c`. Equivalente a `(a (b c *) +)` e a `(a + (b * c))`.

Domanda 54 *Perché l'aritmetica sui puntatori può essere problematica?*

L'aritmetica sui puntatori può essere problematica perché non c'è alcuna garanzia di correttezza e se non usata attentamente potrebbe portare ad accedere a zone della memoria non consentite, provocando un problema di sicurezza.

Domanda 55 *Quali sono i vantaggi della valutazione lazy rispetto alla valutazione eager?*

La valutazione lazy ha una maggiore efficienza perché non forza la valutazione di un'espressione quando non è strettamente necessario, inoltre permette di evitare alcuni errori causati da una valutazione non necessaria.

Domanda 56 *Com'è strutturato un RdA (record di attivazione)?*

Un record di attivazione contiene:

- variabili locali
- parametri in ingresso e uscita
- indirizzo di ritorno (passaggio del controllo)
- link dinamico (al record di attivazione della procedura chiamante)
- link statico (al record di attivazione della procedura genitore, ovvero alla procedura a cui, in presenza di scoping statico, fa riferimento per le variabili non locali)
- risultati intermedi (uno spazio di memoria ausiliario che può essere necessario per valutare espressioni complesse)
- salvataggio dei registri (i registri della CPU contengono dati temporanei, quando il controllo passa ad un'altra procedura quei dati devono essere salvati in memoria)

Domanda 57 *In che modo la programmazione orientata agli oggetti permette di realizzare l'information hiding?*

Perché nelle classi si possono usare i modificatori di visibilità (`public`, `private`, `protected`) che permettono di nascondere campi e metodi all'esterno. Inoltre anche l'ereditarietà permette di nascondere l'implementazione degli oggetti.

Domanda 58 *Cosa si intende per memorizzazione per righe di una matrice?*

Si intende che vengono memorizzate come una concatenazione di righe.

Domanda 59 *Cosa distingue l'iterazione determinata da quella indeterminata?*

Nell'iterazione determinata il numero di ripetizioni è finito e viene deciso all'inizio del ciclo, mentre l'iterazione indeterminata permette di avere cicli che non terminano mai, dato che il numero di ripetizioni è determinato da una condizione logica.

Domanda 60 *Nella compilazione, in cosa consiste la tecnica del bootstrapping?*

Consiste nell'utilizzare un compilatore di un linguaggio per compilare un nuovo compilatore per lo stesso linguaggio, che magari è più efficiente del precedente. Di solito questa operazione viene eseguita due volte in modo da ricompilare il nuovo compilatore con la versione più efficiente.

Domanda 61 *Cos'è la BNF (Backus Naur Form)?*

Non trattata a lezione nell'anno accademico in cui questo pdf è stato scritto.

Domanda 62 *Quali sono i vantaggi dello scoping statico, rispetto a quello dinamico?*

Indipendenza dalla posizione: l'ambiente in cui viene chiamata una funzione non ne modifica il comportamento. Indipendenza dai nomi locali: se nello scoping dinamico si cambiasse il nome di una variabile si modificherebbe il comportamento del programma.

Domanda 63 *Quali sono le caratteristiche del passaggio dei parametri per valore?*

Quando un parametro viene passato per valore il parametro attuale viene copiato in quello formale senza nessuna dipendenza tra loro.

Domanda 64 *Nei comandi in assegnazione, cosa si intende per L-value?*

Il L-value è l'espressione a sinistra dell'assegnamento che denota la locazione di memoria nella quale inserire il valore.

Domanda 65 *In un linguaggio di programmazione, cosa s'intende per ambiente?*

L'ambiente è l'insieme dei legami validi all'interno di un blocco, può essere modificato attraverso le dichiarazioni che aggiungono nuovi legami.

Domanda 66 *Cosa sono le classi astratte? (chiamate "interface" in Java)*

Una classe astratta è una classe con almeno un metodo astratto, ovvero un metodo dichiarato (etichettato come `abstract`), ma non implementato.

P.S.: le classi astratte **NON** sono le interface di Java, le classi astratte possono avere campi e possono avere metodi implementati, mentre le interface no.

Domanda 67 *Cosa sono le operazioni di test-and-set?*

`test_and_set` è un'operazione atomica che testa una variabile booleana (lock) di un programma, se il test ritorna falso, imposta a vero e blocca la risorsa, altrimenti aspetta.

Domanda 68 *Qual è l'uso del costrutto "foreach"?*

L'idea del Foreach è di ripetere il ciclo su tutti gli elementi di un oggetto enumerabile, come array, liste, insiemi, alberi. Il vantaggio maggiore è che permette di separare l'algoritmo di scansione della struttura dati dalle operazioni da svolgere sui singoli elementi.

Domanda 69 *Nel passaggio dei parametri per nome, cosa si intende per chiusura?*

La chiusura è la coppia costituita dall'espressione passata come parametro e dall'ambiente in cui deve essere valutata.

Domanda 70 *Nei linguaggi funzionali, come differisce la valutazione eager da quella lazy?*

Nella valutazione eager ogni espressione viene sempre valutata subito, mentre in quella lazy le espressioni vengono valutate solo se la valutazione viene forzata.

Domanda 71 *Dal punto di vista dei sistemi di tipi, in cosa differiscono Haskell e Scheme?*

Nonostante i due linguaggi siano entrambi fortemente tipati, differiscono per il fatto che in Haskell il controllo di tipo viene fatto staticamente, assegnando a tempo di compilazione un tipo per ciascuna variabile cercando di mantenere la consistenza, in Scheme invece l'assegnazione di tipo viene eseguita a run-time eseguendo dei controlli man mano che il codice viene eseguito.

Domanda 72 *Cos'è un lessema?*

Un lessema è l'unità più piccola e significativa (come operatori e identificatori) identificati dallo scanner durante l'analisi lessicale.

Domanda 73 *Quali sono i vantaggi della compilazione rispetto all'interpretazione?*

I vantaggi della compilazione rispetto all'interpretazione sono che è più efficiente a run time perché l'elaborazione viene fatta tutta prima, gli errori si trovano subito e non c'è bisogno del codice sorgente durante l'esecuzione.

Domanda 74 *Presentare un esempio di polimorfismo ad hoc.*

Un esempio classico di polimorfismo ad hoc (overloading) è la funzione `(+)` che, in base al tipo dei parametri, per esempio int e int oppure float e float ha un comportamento diverso. Tramite l'overloading si evita quindi di dover definire una funzione per ogni tipo.

Domanda 75 *Quali sono le problematiche dell'ereditarietà multipla?*

La problematica dell'ereditarietà multipla è il name clash, ovvero quando si ereditano più metodi con lo stesso nome da classi diverse, il problema è quindi la scelta di quale usare. Un esempio pratico di name clash è nel problema del diamante, in cui abbiamo due classi `A` e `B` che ereditano entrambe dalla classe `Top` un metodo `f()`, su cui solo `B` esegue un override. A questo punto la classe `Bottom` eredita da `A` e `B` e ha il problema di scegliere l'implementazione del metodo `f()`.

Domanda 76 *Nei comandi in assegnazione, cosa si intende per R-value?*

L'R-value è l'espressione a destra dell'assegnamento, denota il valore che verrà assegnato alla locazione di memoria.

Domanda 77 *Cosa si intende per calcolo vettoriale?*

Per calcolo vettoriale si intende l'insieme di operazioni tra vettori che possono essere implementate in hardware per ottenere un'efficienza maggiore dove il contesto lo necessita.

Domanda 78 *Che tipo di dati vengono memorizzati nell'heap?*

I dati dinamici, ovvero quelli che possono cambiare dimensione a run time, per esempio (se permesso dal linguaggio specifico) liste e array.

Domanda 79 *Cosa distingue i linguaggi class based dai linguaggi prototype based?*

I linguaggi prototype based non hanno bisogno delle classi per creare gli oggetti, essi infatti non hanno una struttura ben predefinita e sono visti come dei record dove oltre ai campi ci possono essere i metodi.

Domanda 80 *Cosa produce l'analisi sintattica di un programma?*

L'analisi sintattica di un programma riceve in input lo stream di token, lo analizza e costruisce l'albero della sintassi, ovvero l'albero di derivazione dello stream di token della grammatica libera dal contesto definita per il linguaggio.

Domanda 81 *Elencare alcune motivazioni per la programmazione concorrente.*

- Maggiore efficienza dovuta al fatto che l'hardware moderno è progettato per sfruttare l'esecuzione concorrente.
- Alcuni problemi vengono descritti meglio, come quello del caricamento di una pagina web in cui i vari elementi vengono assegnati a thread diversi.
- Permette di usare hardware distribuito e collegato via web.