Linguaggi di programmazione

Introduzione al corso

Presentazione del corso

Docente: Pietro Di Gianantonio

In sintesi: una trattazione generale dei linguaggi di programmazione

Obiettivi: completare le conoscenze acquisite finora:

- assembly ARM
- Scheme
- Java
- C

Una minuscola parte dei linguaggi disponibili,

- difficile impararli tutti;
- tuttavia si può espandere le proprie conoscenze imparando nuovi linguaggi,

Generalizzazione

Nel caso, dopo aver compreso un certo numero di linguaggi di programmazione si osserva l'esistenza di idee comuni:

- un linguaggio ha sempre molte similitudini con diversi altri,
- nell'imparare un nuovo linguaggio si possono sfruttare le conoscenze acquisite su altri linguaggi

Queste idee comuni possono essere acquisite

- tramite esempi: studio un certo numero di linguaggi di programmazione;
- oppure presentate in maniera generale, astratta, sistematica, (argomento di questo corso)

Aspetti descritti:

Quali sono queste idee generali:

- paradigmi di programmazione: come si svolge la computazione
- costrutti di programmazione: quali sono le componenti base dei programmi,
- gestione dei nomi, ambiente
- gestione della memoria
- meccanismi di controllo di flusso
- chiamate di funzioni passaggio dei parametri
- meccanismi di implementazione: compilatori, interpreti, . . .
- controllo dei tipi

Obiettivi

Fornire un quadro generale dei linguaggi di programmazione senza dover imparare i singoli linguaggi.

Mettere in risalto:

- gli aspetti comuni (e non) dei linguaggi,
- punti critici della loro comprensione.

Rendere più facile l'apprendimento di nuove linguaggi:

- una volta compreso i concetti base, imparare un nuovo linguaggio diventa un lavoro mnemonico più che concettuale.

Altri obiettivi

Fare un uso migliore dei linguaggi:

- avere le idee chiare su alcuni meccanismi complessi: passaggi dei parametri, uso della memoria
- usare i linguaggi nella loro completezza,
  un linguaggio evolvendo introduce nuove feature
  spesso chi programma ne utilizza un minima parte
  ma conoscere tutte le potenzialità permette la scrittura di codice più efficace
- comprendere i costi di implementazione: scegliere tra modi alternativi di fare la stessa cosa: ricorsione di coda

Altri obiettivi

Capire come implementare features che non supportate esplicitamente:

- mancanza di strutture di controllo adeguate, ricorsione in Fortran,
  - trasformare un algoritmo ricorsivo in uno iterativo, eliminazione meccanica della ricorsione

Libri di testo

- M. Gabbrielli, S. Martini. Linguaggi di programmazione - Principi e paradigmi. McGraw-Hill
- articoli e manuali reperibili nella pagina web del corso.

Corso e contenuti in gran parte standard.

Diversi libri di testo con contenuti e ordine di presentazione sovrapponibili

- Michael Scott. Programming language pragmatics Elsevier, MK Morgan Kaufmann.
- Sebasta. Concept in progamming language Pearson

Pro e contro

Gabbrielli-Martini,

- semplice, chiaro, senza banalizzare, pregio principale, specie in confronto ad altri libri di testo,
- astratto: vengono spesso date le definizioni formali dei diversi concetti;
- mancano:
  - argomenti più complessi;
- pochi riferimenti ai linguaggi di programmazione più usati,
  - si preferisce usare un pseudo linguaggio,
  - esempi reperibili negli altri libri di testo,
  - a lezione farò qualche riferimento in più.

Pagina web del corso

www.dimi.uniud.it/pietro/linguaggi

raggiungibile dalla mia home page,

copia della pagina disponibile anche su elearning,

stabile e abbastanza completa,
probabilmente verrà aggiornata, completata nel corso dell'anno.

Organizzazione

- periodo didattico: secondo
- orario, eventualmente modificabile,
  nel caso di sovrapposizioni con altri corsi
- orario di ricevimento (via Teams) mercoledì: 11:30 – 13:00.

Esame

Tradizionale:

- scritto
  - domande di teoria, (se l'esame sarà fatto in presenza)
  - esercizi, saranno modificati rispetto agli anni passati
    - una maggiore varietà
    - minore difficoltà
- esercizi da svolgere a casa e discutere durante l'esame orale,
- orale obbligatorio per tutti.

Perché tanti linguaggi di programmazione?

Più paradigmi di programmazione:

Imperativo:

- von Neumann
  (Fortran, Pascal, Basic, C)
- orientato agli oggetti
  (Smalltalk, Eiffel, C ++?)
- linguaggi di scripting
  (Perl, Python, JavaScript, PHP)

Dichiarativo:

- funzionale: descrivo insiemi di funzioni
  (Scheme, ML, pure Lisp, FP)
- logico, basato su vincoli:
  descrivo un insieme di predicati,
  il programma dato delle variabil (Prolog, VisiCalc, RPG)

Perché tanti linguaggi di programmazione?

- evoluzione: nel tempo si definiscono nuove costrutti, tecniche, principi di programmazione,
- fattori economici: interessi proprietari, vantaggio commerciale
- diverse priorità: codice efficiente, pulizia del codice, flessibilità
- diversi usi:
  calcolo scientifico (Fortan),
  analisi dei dati (R),
  sistemi embedded (C),
  applicazioni web (JavaScript, PHP),
  ...

Cosa rende un linguaggio di successo?

- buon supporto: librerie, codice preesistente, IDE: editor, debugger
  (tutti i linguaggi più diffusi)
- supporto da uno sponsor importante
  (C#, Visual Basic, F#, Objetive C, Switft, Go)
- ampia diffusione a costi minimi, portabilità
  (JavaScript, Java, Pascal)
- espressivo, flessibile, potente
  (C, Lisp, APL, Perl)
- possibilità di ottenere codice efficiente / compatto
  (Fortran, C)
- facile da implementare
  (BASIC, Forth)

Qualità di un linguaggio:

- Leggibilità
  - chiarezza, naturalità, semplicità:
  - supporto all'astrazione:
  - è facile modificare del codice.
- Scrivibilità

  è facile passare dagli algoritmi al codice,
  - semplice da imparare:
    (BASIC, Pascal, Scheme)
  - polimorfismo: stesso codice su dati diversi;
  - ortogonalità:
    aspetti diversi restano indipendenti
    (es. tipi dato e passaggio parametri a procedure), conseguenza: si integrano in modo naturale, poche eccezione
- Affidabilità
  - facilità di verifica, non ci sono errori nascosti
- Costo
  - efficienza

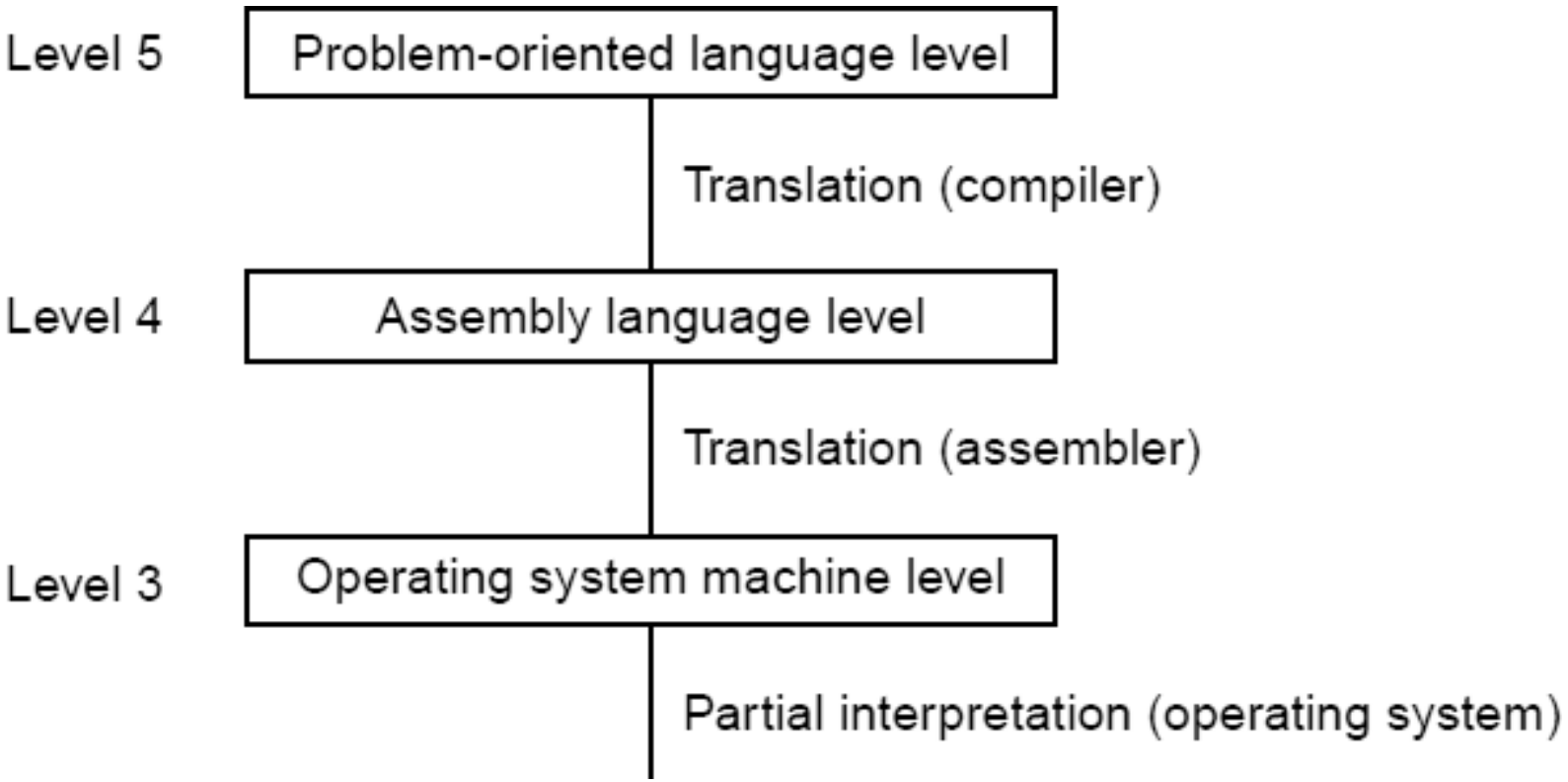
Aspetti di un linguaggio di programmazione

- sintassi: quali sequenze di caratteri costituiscono programmi, la loro struttura,
- semantica: come si comporta un programma, l'effetto della loro esecuzione,
- pragmatica: utilizzo pratico; come scrivere buon codice; convenzioni, stili nello scrivere i programmi,
- implementazione: come il codice viene convertito in istruzioni macchina, eseguito,
- librerie: codice fornito con il linguaggio per implementare funzionalità base,
- tools: per editing, debugging, gestione del codice.

Nel corso consideriamo principalmente i primi 4 aspetti, gli ultimi due importanti ma molto variabili.

Macchina astratta.

- Sistema di calcolo diviso in un gerarchia di macchine virtuali (astratte) $\mathcal{M}_i$.
- Ciascuna costruita sulla precedente, a partire dal livello hardware.
- Ciascuna caratterizzata dal linguaggio, $\mathcal{L}_i$, che riesce ad eseguire.



Esecuzioni del codice

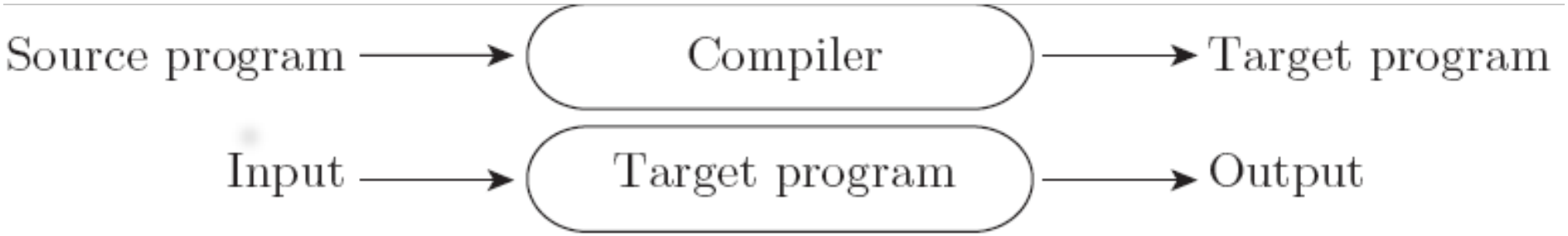
Con quali meccanismi si esegue un programma, per esempio, scritto in $\mathcal{L}_{Java}$,
quindi relativo alla macchina astratta $\mathcal{M}_{Java}$ (macchina Java)

- si sfrutta uno dei livelli di macchina sottostanti,
  per esempio una $\mathcal{M}_{JVM}$ (Java Virtual Machine)
- che qualcuno ha già implementato,
- si esegue una traduzione nel linguaggio relativo $\mathcal{L}_{JVM}$ (Java ByteCode)

Compilazione vs. Interpretazione

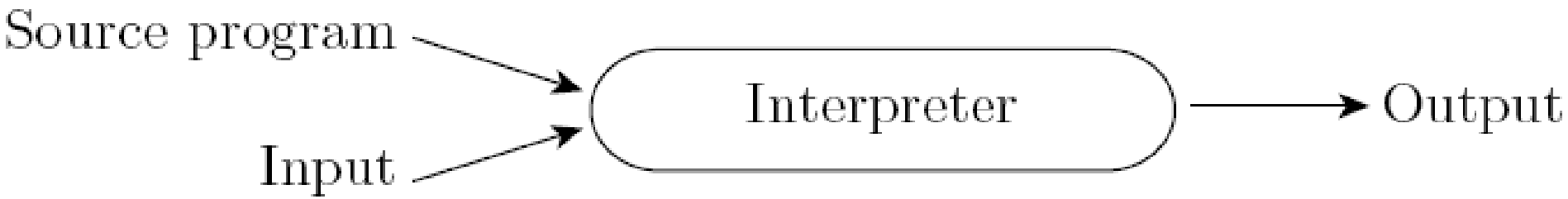
Compilazione pura:

- Il compilatore traduce il programma sorgente di alto livello in un programma di destinazione equivalente
(spesso in linguaggio macchina + SO,
- Programma sorgente e compilatore non necessari durante l'esecuzione del codice.



Interpretazione pura

- L'interprete riceve programma sorgente e dati, e traduce, passo passo, le singole istruzione che vengono eseguite immediatamente.
- L'interprete, programma sorgente, sono presenti durante l'esecuzione del programma.
- L'interprete è il luogo del controllo durante l'esecuzione.



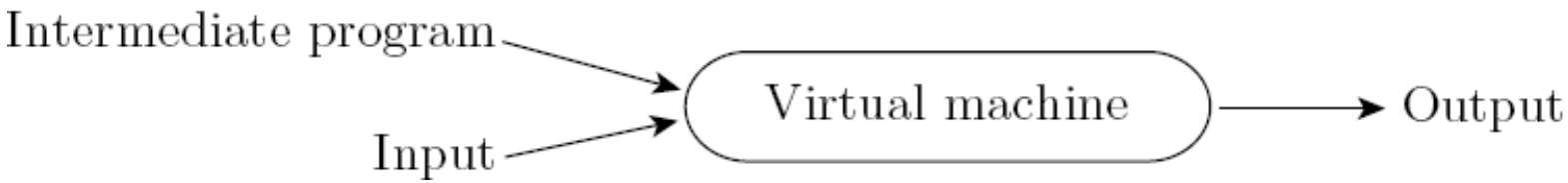
Compilazione vs. Interpretazione

- Compilazione
  - Migliori prestazioni: si evitano traduzioni e controlli a tempo di esecuzione
  - sono possibili controlli prima dell'esecuzione (type-checking statico) errori messi subito in evidenza
- Interpretazione:
  - maggiore flessibilità (Scheme),
  - più semplice da implementare,
  - esecuzione diretta del codice,
  - più semplice il debugging.

Compilazione vs. Interpretazione

Nei casi reali, la traduzione in codice macchina avviene con più passi, entrano in gioco più macchine virtuali intermedie
tra linguaggio di programmazione e codice macchina.

- una combinazione tra compilazione e interpretazione
  - linguaggi interpretati:
    pre-processing seguita dall'interpretazione
  - linguaggi compilati:
    generazione codice intermedio
    esempi: Pascal P-code, Java bytecode, Microsoft COM +



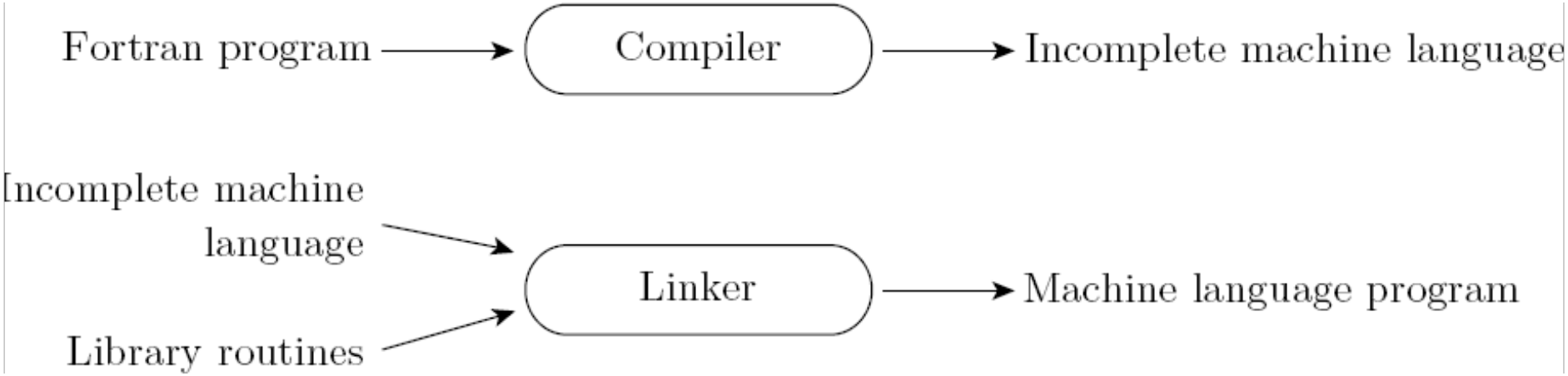
Preprocessing vs. Compilazione

Lo schema precedente può descrivere un pre-processing (semplice elaborazione dell'input)

- in questo caso caso parliamo di linguaggi interpretati,
- compilazione: traduzione da un linguaggio ad un altro,
  - prevede un'analisi complessiva dell'input
  - viene riconosciuta la struttura sintattica del programma, controllo di errori
- nel preprocessing questi aspetti non sono presenti,
  - trasformazione sintattica e locale del programma

Supporto a run-time:

- raramente compilatore produce solo codice macchina, anche istruzioni virtuali
  - chiamate al sistema operativo
  - chiamate a funzioni di libreria
    es. funzioni matematiche (sin, cos, log, ecc.), input-output
- traduzione, non a livello di codice macchina, ma a livello di macchina virtuale intermedia: sistema operativo, livello di libreria
- un programma linker unisce codice library subroutine



Assemblaggio post-compilazione

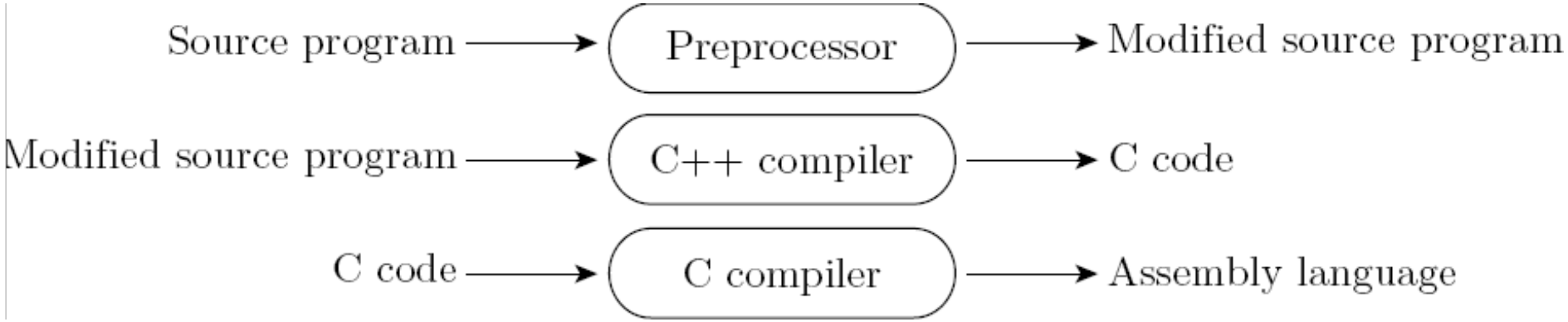
Il compilatore produce assembly (non codice macchina)

- facilita il debugging (assembly più leggibile)
- isola il compilatore da modifiche nel formato delle istruzioni (solo l'assemblatore, condiviso tra molti compilatori, deve essere modificato)

Con questa tecnica si usa la macchina virtuale assembly.

Traduzioni da linguaggio a linguaggio (C++)

Prime implementazioni C++ generano un programma intermedio in C:



Con questa tecnica si usa la macchina virtuale C.

Compilazione dinamica, just-in-time

- compilazione svolta all'ultimo momento.
- casi tipici: programmi in bytecode,
  si migliorano le prestazioni rispetto all'interprete
  - Java byte (JVM)
  - analogamente un compilatore C# produce
    .NET Common Intermediate Language (CIL),
- l'interprete, durante l'esecuzione del programma, decide di tradurre, trasformare in codice macchina, blocchi di codice,
  - per migliorare l'efficienza,
  - preserva flessibilità, sicurezza,
- altre esempi: Lisp o Prolog invocano il compilatori just-in-time, per tradurre il nuovo sorgente creato in linguaggio macchina o per ottimizzare il codice per un particolare set di input.
- meccanismo simile usato nell'esecuzione di codice x86 all'interno

Bootstrap:

sollevarsi dal suolo tirando i lacci dei propri stivali (Barone Munchausen)

Implementare il compilatore

- scrivo un nuovo compilatore per C, come programma in C.
- circolo vizioso evitato usando versioni differenti
  - compilo il nuovo compilatore con un vecchia versione del compilatore
  - se il nuovo compilatore produce codice più efficiente,
    ricompilo usando la nuova versione, per ottenere un compilatore più efficiente

Pascal e P-code

P-code: codice intermedio (come il Java bytecode).

Meccanismo per semplificare la creazioni di un compilatore Pascal, si parte da:

- Compilatore Pascal, scritto in Pascal: $\mathcal{C}_{Pascal}^{Pascal \to PCode}$,
- Compilatore Pascal, scritto in P-Code: $\mathcal{C}_{PCode}^{Pascal \to PCode}$ (traduzione del predente)
- Interprete PCode, scritto in Pascal: $\mathcal{I}_{Pascal}^{PCode}$

Esecuzione tramite PCode

Per ottenere un interprete, a mano costruisco:

- Interprete PCode, scritto nel mio linguaggio macchina: $\mathcal{I}_{LM}^{PCode}$

Preso un programma Pascal *PrPa* posso:

- ottenere la sua traduzione in PCode:
  - $PrPC = \mathcal{I}_{LM}^{PCode}(\mathcal{C}_{PCode}^{Pascal \to PCode}, PrPa)$
- eseguire la traduzione:
  - $\mathcal{I}_{LM}^{PCode}(PrPC, Dati)$

Compilatore Pascal

Per ottenere un compilatore Pascal scritto in linguaggio macchina:

- a mano, trasformo il $\mathcal{C}_{Pascal}^{Pascal \to PCode}$ in $\mathcal{C}_{Pascal}^{Pascal \to LM}$, solo una piccola parte del codice va modificata

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

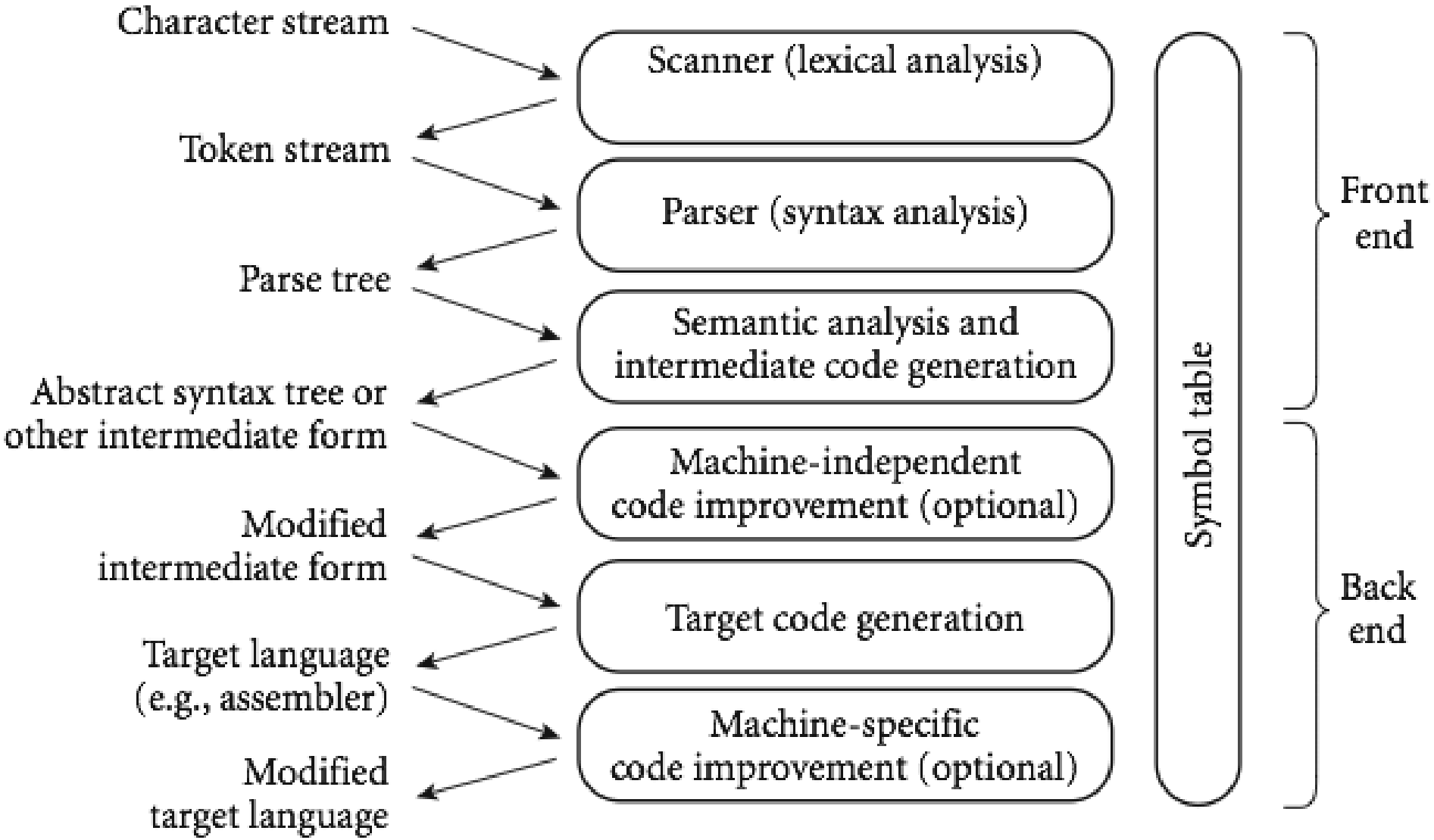
  .
- $\mathcal{C}_{LM}^{Pascal \to LM} = \mathcal{I}_{LM}^{PCode}(\mathcal{C}_{PCode}^{Pascal \to LM}, \mathcal{C}_{Pascal}^{Pascal \to LM})$

  .

Notare come l'uso di un formalismo aiuti la spiegazione di un meccanismo complesso.

Una panoramica della compilazione

Compilazione divisa in più fasi.



Analisi lessicale - Scansione:

- scanner: lavoro è semplice,
- divide il programma in lessemi, le unità più piccole e significative,
- per ogni lessema produce un token
- semplifica le fasi successive
  possibile progettare un parser che esamina caratteri anziché token come input, ma inefficiente
- le singoli classi di lessemi descritti linguaggi regolari,
- lo scanner implementa un DFA

Analisi sintattica - Parsing

- si analizza l'intero programma, definendo la sua struttura
- sintassi descritta mediante linguaggi liberi dal contesto,
- riconosciuti tramite PDA
- viene costruito l'albero della sintassi astratta
  una rappresentazione ad albero della struttura del programma

L'analisi semantica

- esegue controlli sul codice statici (es type checking) non implementabili dal parser (contex free grammar)
- altri controlli (ad esempio, indice di matrice fuori limite) eseguibili sola a tempo di esecuzione (dinamici)

Modulo intermedio

Si produce codice intermedio

- codice intermedio: indipendente dal processore, facilità di ottimizzazione o compattezza (richieste contrastanti)
- codice intermedio assomiglia spesso a codice macchina per qualche macchina astratta;

  per esempio: una macchina stack o una macchina con molti registri

L'ottimizzazione

trasforma il programma in codice intermedio e in uno equivalente (?) con esecuzioni più velocemente o con meno memoria

- il termine è improprio; si migliora il codice:
- fase facoltativa

La fase di generazione del codice produce linguaggio assembly o linguaggio macchina rilocabile (a volte)

Alcune ottimizzazioni specifiche della macchina

- uso di istruzioni speciali, modalità di indirizzamento, ecc.

se presenti sono eseguite durante o dopo la generazione del codice target

Tabella dei simboli

tutte le fasi si basano su una tabella dei simboli

- tiene traccia di tutti gli identificatori nel programma e di ciò che il compilatore sa di loro
- può essere conservata per poi essere utilizzata da un debugger,

  anche dopo che la compilazione è stata completata

Grammatiche e automi

Analizzatori lessicali e sintattici

Descrizione di un linguaggio di programmazione

sintassi	quali sequenze di caratteri costituiscono programmi, la loro struttura
semantica	significato, come si comporta un programma, l'effetto della sua esecuzione
pragmatica	utilizzo pratico; come scrivere buon codice; convenzioni, stili nello scrivere i programmi.
implementazione	come eseguire il codice sorgente

Descrizione della sintassi linguaggio di programmazione

- Quali sequenze di carattere formano un programma, e quali no;
- più nel dettaglio, qual'è la struttura sintattica di un programma divisione del programma in componenti e sotto-componenti quali: dichiarazioni, blocchi, cicli, singole istruzioni.
- Descrizione, completamente formale, ottenuta tramite grammatiche.
- Le nozioni che seguono sono già state presentate a Fondamenti di Informatica.

Grammatiche

Formalismo nato per descrivere linguaggi, anche naturali

- Panini, IV secolo a.C., definisce una grammatica formale per il Sanscrito,
  in occidente nessuna teoria così sofisticata sino al XX secolo;
- Noam Chonsky, anni 50, descrizione formale dei linguaggi naturali

Formalismo che mi permette di generare:

- tutte le frasi sintatticamente corrette della lingua italiana
- tutti i programmi sintatticamente corretti,

Ma anche mettere in evidenza la struttura della frase, del programma.

Grammatica costituita da:

- un insieme di simboli terminali, elementi base,
  a secondo dei casi posso essere:
  - l'insieme di parole delle lingua italiana,
  - l'inseme dei simboli base di un linguaggio di programmazione:
    simboli di operazione, identificatori, separatori,
  - lettere di un alfabeto $\mathcal{A}$,
- un insieme di simboli non terminali, categorie sintattiche,
  a seconda dei casi posso essere:
  - nomi, verbi, articoli,
    predicato-verbale, complemento-di-termine,
    proposizione, subordinata, ecc.
  - identificatori, costanti, operazione-aritmetica
    comando-di-assegnazione, espressione-aritmetica,
    ciclo, dichiarazione-di-procedura, ...

Grammatica costituita da:

- un insieme di regoli generazione,
  spiegano come sono composti i non-terminali,
  come posso espandere un non terminale esempi di regole (generative):
  - proposizione $\rightarrow$ soggetto predicato-verbale
  - proposizione $\rightarrow$ soggetto predicato-verbale complemento-oggetto
  - programma $\rightarrow$ dichiarazioni programma-principale
  - assegnamento $\rightarrow$ identificatore “=” espressione
- le grammatiche vengono divise in classi
  in base alla complessità delle regole ammesse
  - grammatiche regolari, libere da contesto, dipendenti dal contesto
- regole più sofisticate:
  - permettono di definire più linguaggi, linguaggi più complessi
  - ma determinare se una parola appartiene alla grammatica diventa più complesso

Formalmente

Un linguaggio su alfabeto $\mathcal{A}$, è un sottoinsieme di $\mathcal{A}^*$, (stringhe su $\mathcal{A}$)

Definizione: Grammatica libera (dal contesto)

Caratterizzata da:

- T: insieme di simboli terminali (alfabeto del linguaggio)
- NT: insieme di simboli non terminali (categorie sintattiche)
- R: insieme di regole di produzione
- S: simbolo iniziale $\in$ NT

Regole R (libere da contesto) nella forma:

$$V \rightarrow w$$

con V $\in$ NT e w $\in$ (T $\cup$ NT)*

Esempio: stringhe palindrome

a partire dalle lettere a, b,

- terminali: a, b
- non-terminali: P
- regole:
  - $P \rightarrow$
  - $P \rightarrow a$
  - $P \rightarrow b$
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- simbolo iniziale: P

Esempio: espressioni aritmetiche

a partire dalle variabili a, b, c

- terminali: a, b, c, +, -, *, (,)
- non-terminali: E, T, F, Var (espressione, termine, fattore)
- regole:
  - $E \to T$
    $E \to E + T$
    $E \to E - T$
  - $T \to F$
    $T \to T * F$
  - $F \to (E)$
    $F \to Var$
  - $Var \to a$, $Var \to b$, $Var \to c$
- simbolo iniziale: E

Formulazione alternativa: BNF (Backus-Naur Form)

- Sviluppata per l'Algol60
- non terminali marcati da parentesi: $\langle$E$\rangle$
  (non devo definirli esplicitamente)
- $\rightarrow$ sostituita da ::=
- regole con stesso simbolo a sinistra raggruppate

  $\langle$E$\rangle$ ::= $\langle$T$\rangle$ | $\langle$E$\rangle$ + $\langle$T$\rangle$ | $\langle$E$\rangle$ - $\langle$T$\rangle$
- si usa una diversa meta-sintassi

Esistono notazioni miste:

`E` $\rightarrow$ `T | E + T | E - T`

Derivazioni e alberi di derivazione

- Derivazione: come passare da simbolo iniziale a parola finale, una regola alla volta,
  - $E \to E + T \to T + T \to F + T \to Var + T \to a + T \to a + F \to$ $\to a + Var \to a + b$
  - $E \to E + T \to E + F \to E + Var \to E + b \to T + b \to F + b \to$ $\to Var + b \to a + b$
- Albero di derivazione:
  - rappresentazione univoca della derivazione,
  - mette in evidenza la struttura del termine.
- Formalmente gli alberi di derivazione sono alberi ordinati ossia:
  - grafo orientato,
  - aciclico, connesso
  - ogni nodo ha al più un arco entrante,
  - gli archi uscenti sono ordinati,

Albero di derivazione, definizione formale

L'albero di derivazione su una grammatica $\langle$ T, NT, R, S $\rangle$ soddisfa:

- radice etichettata con il simbolo iniziale S,
- foglie etichettate con 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 $\rightarrow$ w, in R

Alberi di derivazione fondamentali perché descrivo la struttura logica della stringa analizzata.

I compilatori costruisco alberi di derivazione.

Grammatiche ambigue:

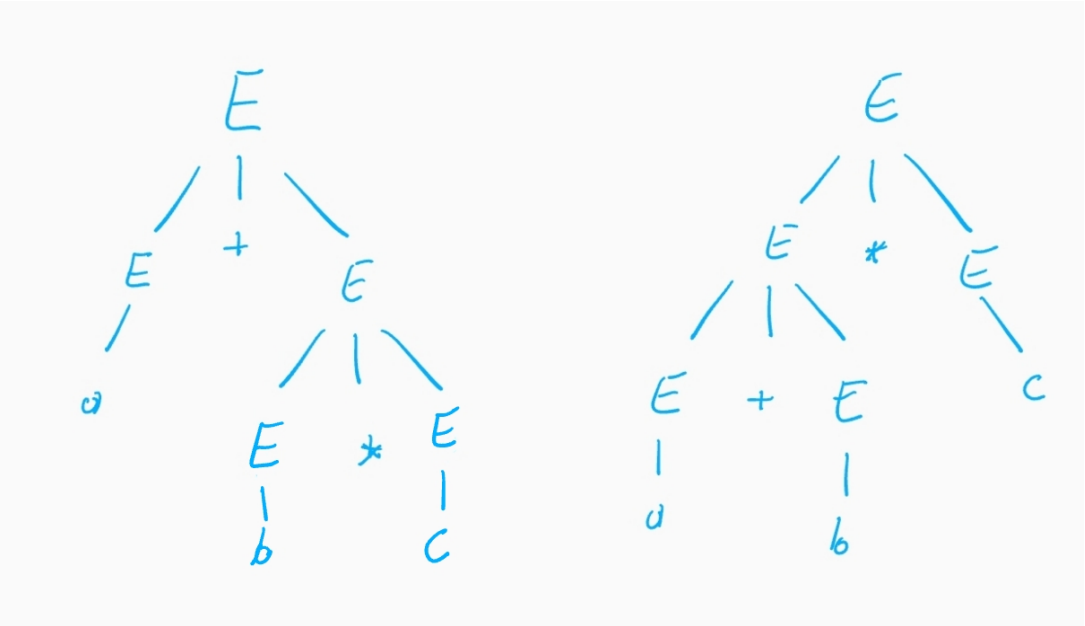
Una grammatica alternativa per le espressioni aritmetiche:

`E` $\rightarrow$ `E + E | E - E | E * E | ( E ) | a | b | c`

Le due grammatiche generano lo stesso linguaggio ma la seconda è ambigua:

`a + b * c`

ha due alberi di derivazioni nella seconda grammatica



Grammatiche ambigue

- Grammatiche per cui esiste una stringa con due alberi di derivazione differenti
- i due alberi di derivazione inducono
  - due interpretazioni diverse della stessa stringa
  - due meccanismi di valutazione differente
- Queste ambiguità vanno evitate (come in matematica)

Disambiguare una grammatica

Due soluzioni:

- rendere la grammatica non ambigua
  - tipicamente attraverso: nuovi non-terminali, non terminali
  - ottengo una grammatica che genera lo stesso linguaggio ma più complessa
- convivere con grammatiche ambigua
  - si forniscono in informazione aggiuntive su come risolvere le ambiguità
  - tipicamente, si specifica:
    - ordine di precedenza degli operatori,
    - per un singolo operatore, o per operatori nella stessa classe di equivalenza, se associativo a sinistro o a destra.

  soluzione usata anche nei parser

Altri esempi di ambiguità

Sempre la grammatica

`E` $\rightarrow$ `E + E | E - E | E * E | ( E ) | a | b | c`

e le stringhe

```
a - b - c
a - b + c
```

bisogna stabilire se si associa a sinistra:

```
(a - b) - c
(a - b) + c
```

oppure si associa a destra:

```
a - (b - c)
a - (b + c)
```

Altri esempi di ambiguità

Grammatica

⟨stat⟩ ::= IF ⟨bool⟩ THEN ⟨stat⟩ ELSE ⟨stat⟩
| IF ⟨bool⟩ THEN ⟨stat⟩ | ...

ambigua su:

IF ⟨bool1⟩ THEN IF ⟨bool2⟩ THEN ⟨stat1⟩ ELSE ⟨stat2⟩

due interpretazioni:

Altri esempi di ambiguità

Grammatica

⟨stat⟩ ::= IF ⟨bool⟩ THEN ⟨stat⟩ ELSE ⟨stat⟩
| IF ⟨bool⟩ THEN ⟨stat⟩ | ...

ambigua su:

IF ⟨bool1⟩ THEN IF ⟨bool2⟩ THEN ⟨stat1⟩ ELSE ⟨stat2⟩

due interpretazioni:

IF ⟨bool1⟩ THEN (IF ⟨bool2⟩ THEN ⟨stat1⟩ ELSE ⟨stat2⟩)

IF ⟨bool1⟩ THEN (IF ⟨bool2⟩ THEN ⟨stat1⟩) ELSE ⟨stat2⟩

Abstract syntax tree

Gli alberi di derivazione contengo informazioni utili per interpretare, valutare, dare semantica alle stringhe.

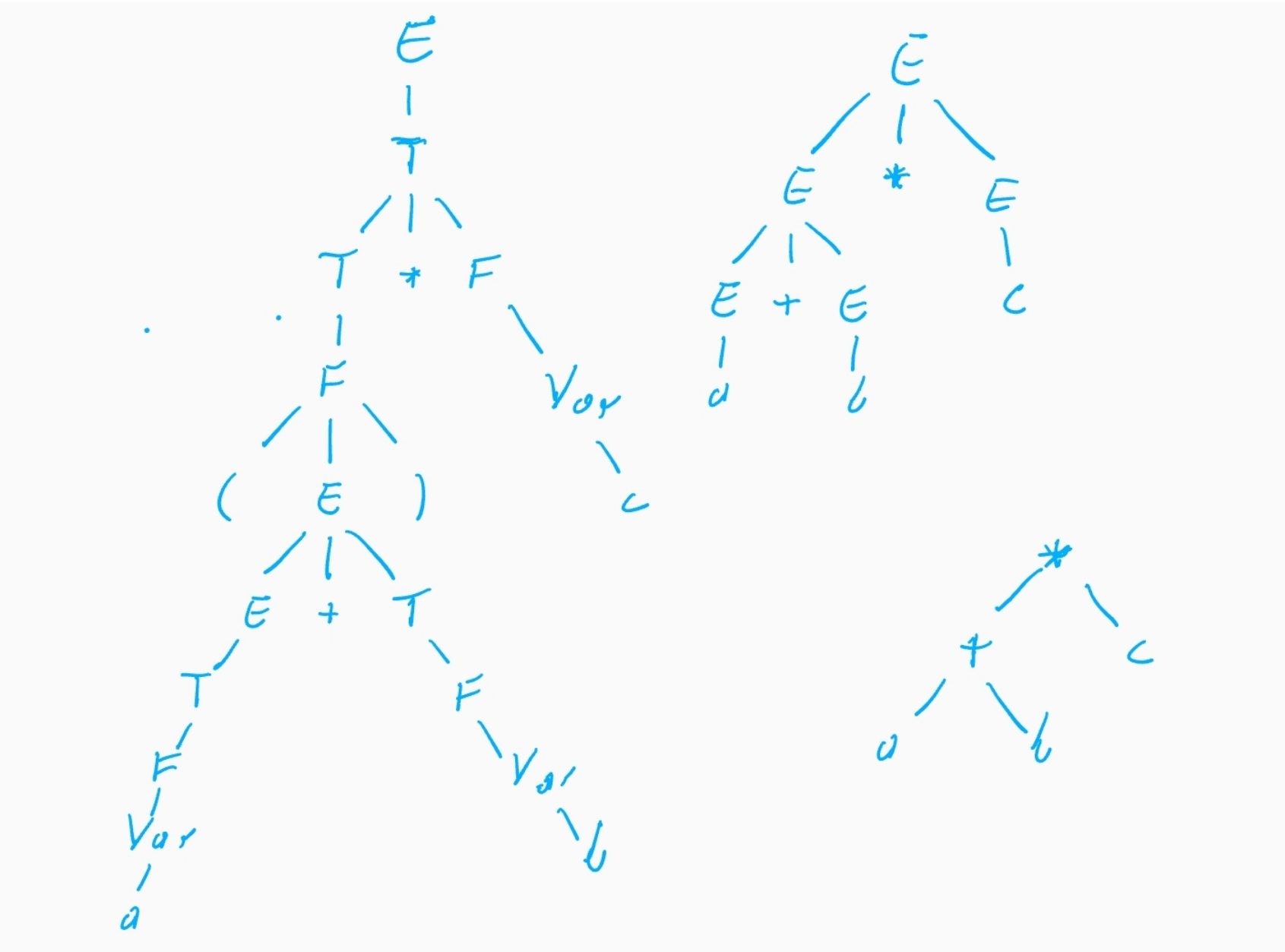
Ma :

- A volte necessario complicare grammatica ed espressioni per definire la giusta interpretazione

  (a + b) * c

  - parentesi ( ) necessarie per definire la precedenza
  - questi orpelli sono inutili una volta costruito l'albero sintattico dell'espressione.
- Gli alberi di derivazione possono essere ridondanti, contenere informazione inutili per l'interpretazione dell'espressione:

Esempio



Abstract syntax tree

- Abstract syntax tree: albero sintattico a cui sono stati eliminati
  - nodi ridondanti ai fini dell'analisi,
- rappresentazione più compatta, contiene solo le informazioni utili,
  - permette una computazione più pratica, efficiente
  - rappresentazione usata nei compilatori.
- il significato di una stringa di caratteri è meglio evidenziato dal suo abstract tree che dall'espressione stessa

Sintassi astratta

- Sintassi che genera gli abstract syntax tree.
  Esempio, per le espressioni aritmetiche:

  E $\rightarrow$ E + E | E - E | E * E | Var

- sintassi minimale, mette in luce la parte importante della sintassi concreta
- sintassi intrinsecamente ambigua,
  - prevede dei meccanismi esterni alla grammatica per risolvere le ambiguità

Classi di grammatiche

Oltre alle grammatiche libere dal contesto esistono anche:

Grammatiche

- a struttura di frase
- dipendenti da contesto
- libere da contesto
- lineari sinistre, lineari destre
- regolari

differenze:

- diverse grado di libertà nel definire le regole
- con grammatiche più generali:
  - una classe più ampia di linguaggi definibili
  - più complesso decidere se un parola appartiene al linguaggio, costruire albero di derivazione

Classe di grammatiche usate nei parser

Grammatiche libere dal contesto sono compromesso ottimale tra espressività e complessità.

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

Vincoli sintattici contestuali:

Tuttavia con grammatiche libere dal contesto non posso eliminare dall'insieme di programmi riconosciuti (accettati)

programmi che non rispettano alcuni vincoli sintattici (contestuali) come:

- identificatori dichiarati prima dell'uso
- ugual numero di parametri attuali e formali
  - controllo non possibile perché il linguaggio $fa^n b^* fa^n b^* fa^n$ non è libero da contesto
- non modificare variabile di controllo ciclo for
- rispettare i tipi nelle assegnazioni

Soluzione (dei compilatori):

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

Fase di analisi semantica chiamata anche semantica statica

- controlli sul codice eseguibili a tempo di compilazione
- in contrapposizione alla
  semantica dinamica: controlli eseguiti durante l'esecuzione.

Semantica

- termine usato un po' impropriamente nelle espressioni: semantica statica, semantica dinamica
- semantica di un programma definisce il suo significato, descrive il suo comportamento a run-time più di un semplice controllo degli errori.
- Semantica definita quasi sempre informalmente in linguaggi naturale
  un approccio formale è possibile:
  - semantica operazionale strutturata : programma descritto da un sistema di regole (di riscrittura) descrivono il risultato della valutazione di un qualsiasi programma
  - semantica denotazionale: descrivo con delle strutture matematiche (funzioni) il comportamento del programma.

Front end del compilatore

Similmente ai linguaggi naturali nella cui sintassi:

- si descrivono l'insieme di parole valide, il dizionario, divise in categorie (articoli, nomi, verbi, . . .)
- regole per costruire frasi da parole

nei linguaggi formali, nei compilatore:

- descrizione delle parti elementare lessema , analisi lessicale
- descrizione della struttura generale, a partire da lessemi, analisi sintattica

La separazione rende più efficiente l'analisi del testo.

Analisi lessicale (scanner, lexer)

Nella stringa di caratteri riconosco i lessemi, e per ogni lessema costituisco 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, ;)]
```

La sequenza di token generati viene passata all'analizzatore sintattico (parser)

Analisi lessicale

Bisogna definire per ogni categoria sintattica:

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

la corrispondente sintassi
ossia quali stringe di caratteri possono essere: un identificatore, un letterale ...

Come esprimere la sintassi di una categoria sintattica: espressione regolare

Linguaggi e operazioni su linguaggi

Sia $\mathcal{A}$ un alfabeto, un insieme di simboli.

Un linguaggio con alfabeto $\mathcal{A}$, è definito come

- un insieme di stringhe di elementi $\mathcal{A}$ (parole su $\mathcal{A}$)

Sui linguaggi posso definire le operazioni di:

- unione: $L \cup M$
- concatenazione $L\,M = \{ s\,t \mid s \in L , t \in M \}$

  dove $s\,t$ indica la concatenazione della stringa s con la stringa t
- chiusura di Kleene $L^* = \{ s_1\ s_2 \ldots\ s_n \mid \forall\ i\ .\ s_i \in L\}$

A partire da queste operazioni (e da un insieme di constanti, costruisco

- l'insieme delle espressioni regolari, ossia l'algebra di Kleene

Espressioni regolari permettono una rappresentazione sintetica dei linguaggi.

Espressioni regolari, algebra di Kleene

Espressioni (algebriche), L, M, N, . . . costruite a partire da

- un insieme di constanti:
  - i simboli di un alfabeto $\mathcal{A}$,
  - dal simbolo ϵ rappresentante la stringa, parola, vuota.
- l'insieme delle operazione sui linguaggi:
  - concatenazione: LM o L · M
  - unione: L | M
  - chiusura di Kleene: L*

Sintassi espressioni regolari

Oltre alle operazioni base, le espressioni regolari sono formate da

- parentesi tonde, ( ), per determinare l'ordine di applicazione,

Convenzioni, regole per evitare di dover inserire troppe parentesi:

- concatenazione e unione, associative

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

- esiste un ordine di precedenza tra gli operatori
  - in ordine decrescente:

    `*, · , |`

    `a|bc* = a | (b (c*))`

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) ->

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* ->

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) ->

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }
- ab*(c|ε) ->

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }
- ab*(c|ε) -> {“a”, “ac”, “ab”, “abc”, “abb”, “abbc”, . . . }

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }
- ab*(c|ε) -> {“a”, “ac”, “ab”, “abc”, “abb”, “abbc”, . . . }
- (0|(1(01*0)*1))* =

Da espressione regolari a linguaggi, esempi:

- a|b* = a|(b*) -> {“a”, ε, “b”, “bb”, “bbb”, . . . }
- (a|b)* -> { ε, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, . . . }
- a|bc* = a|(b(c*)) -> {“a”, “b”, “bc”, “bcc”, “bccc”, . . . }
- ab*(c|ε) -> {“a”, “ac”, “ab”, “abc”, “abb”, “abbc”, . . . }
- (0|(1(01*0)*1))* = { ε, “0”, “00”, “11”, “000”, “011”, “110”, “0000”, “0011”, “0110”, “1001”, “1100”, “1111”, “00000”, . . . }

  tutti i numeri, scritti numeri in base 2, che sono multipli di 3.

Da espressioni regolari a linguaggi

Ogni espressioni regolare rappresenta un linguaggio.

Definizione formale:

- $\mathcal{L}(\ \varepsilon\ ) = \{\ \varepsilon\ \}$
- $\mathcal{L}(\text{a}) = \{\ \text{“a”}\ \}$
- $\mathcal{L}\ (\text{L} \mid \text{M}) = \mathcal{L}(\text{L}) \cup \mathcal{L}\ (\text{M})$
- $\mathcal{L}\ (\text{L M}) = \mathcal{L}(\text{L})\ \mathcal{L}\ (\text{M})$
- $\mathcal{L}\ (\text{L}^*) = (\mathcal{L}(\text{L}))^*$

Estensioni

Oltre a operazioni base posso definire altre operazioni

- chiusura positiva: $L^+ = L\ L^*$
- zero o un istanza: $L? = \varepsilon \mid L$
- n concatenazioni di parole in L: $L\{n\} = L\ L \ldots L$
- uno tra: [acdz] = a | c | d | z
- range: [a-z] = a | b | c | . . . | z
- opposto: [ˆa-z] - tutti i caratteri meno le lettere minuscole.

Espressioni più compatte, ma stessa classe di linguaggi.

- Esistono molte altre estensioni.
- Espressioni regolari usate:
  in applicativi per manipolare stringhe, text-editor,
  in funzioni di libreria di molti linguaggi di programmazione

Definizione tramite equazioni

Usata in alcune applicativi, librerie.

Permette una scrittura più chiara:

- digit := [0-9]
- simple := {digit}$^+$
- fract := {simple} . {simple} | . {simple}
- exp := {fract} e (ε | + | -) {simple}
- num := {simple} | {fract} | {exp}

al posto di

- num := [0-9]$^+$ | [0-9]$^+$.[0-9]$^+$ | .[0-9]$^+$ |
  [0-9]$^+$.[0-9]$^+$ e (ε | + | -) [0-9]$^+$ | .[0-9]$^+$ e (ε | + | -) [0-9]$^+$

Notare le parentesi graffe distinguono

- “non terminali”, identificatori come {digit}

Risultati teorici

Teorema di equivalenza

Linguaggi regolari posso essere descritti in molti diversi:

- espressioni regolari
- grammatiche regolari
- automi finiti non deterministici, NFA non determistic finite automata.
- automi finiti deterministici (macchine a stati finiti) DFA determistic finite automata.

Teorema di minimalità

Esiste l´automa deterministico minimo (minor numero di stati)

Nuovo formalismo DFA, esempio

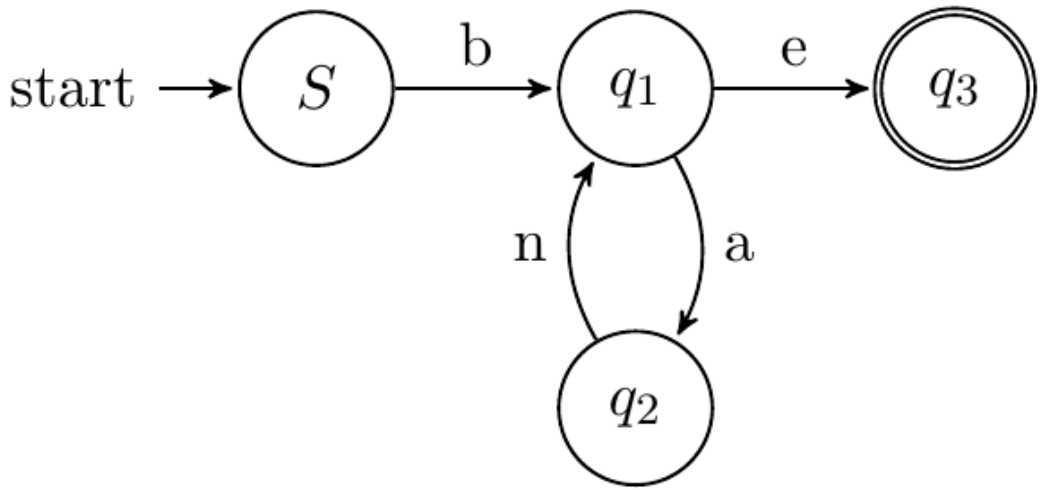


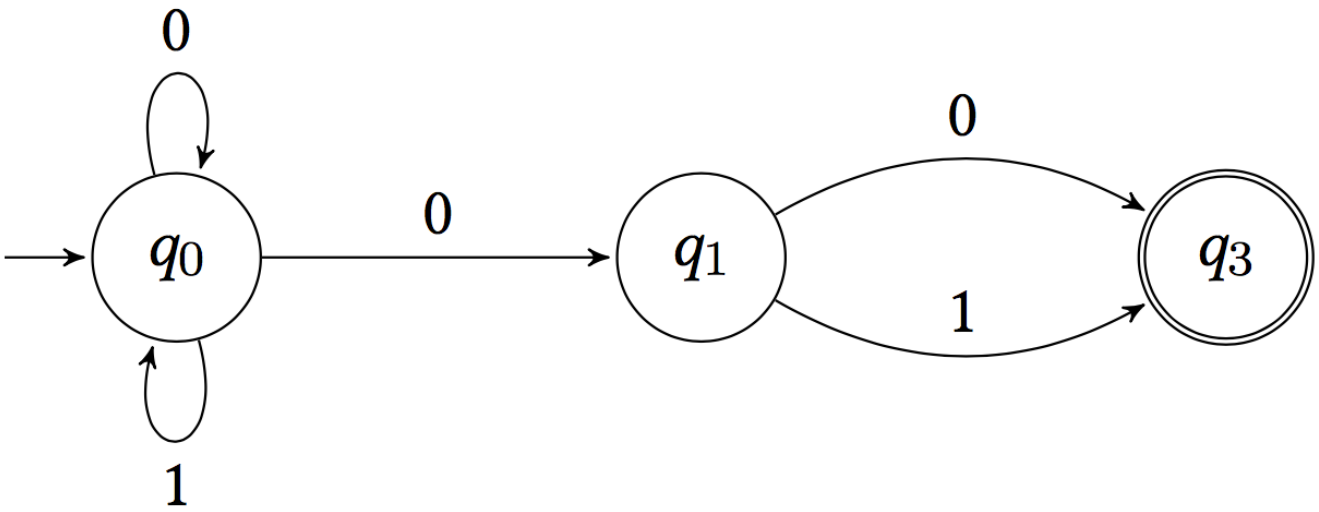
Figure 1: Finite State Automaton, accepting the pattern $b(an)+e$

Riconoscitore per b(an)*e

Se anche q_2 stato finale il linguaggio generato diventa

- b(an)*(e|a)
- b(an)*e | b(an)*a

Alternativa NFA, esempio



Riconoscitore per (0|1)*0(0|1)

Nei NFA (non-deterministic finite automata) più possibili alternative

- più archi in uscita con la stessa etichetta,
- stringa accettate se esiste una sequenza di transazioni che la riconosce.

Applicazioni dei risultati teorici

Per costruire un riconoscitore per un'espressione regolare

- Dall'espressione regolare costruisco:
  - NFA equivalente, da questi il
  - DFA equivalente, da questi il
  - l'automa minimo, (DFA minino),

  tutte costruzioni effettive.
- Dall'automa minimo costruisco un programma per decidere se una parola appartiene a un espressione regolare.
- Programma simula il comportamento dell'automa minimo contiene un tabella che descrive le transazione del automa minimo, e ne simula il comportamento.

Scanner, lexer

Lo scanner deve risolvere un problema più complesso del semplice riconoscimento di una singola espressione regolare.

- Dati
  - un insieme di espressioni regolari,
    classi di lessemi (es. identificatori, numeri, operazioni, . . .),
  - una stringa di ingresso
- lo scanner deve dividere la stringa d'ingresso in lessemi, ciascuno riconosciuto da un espressione regolare.

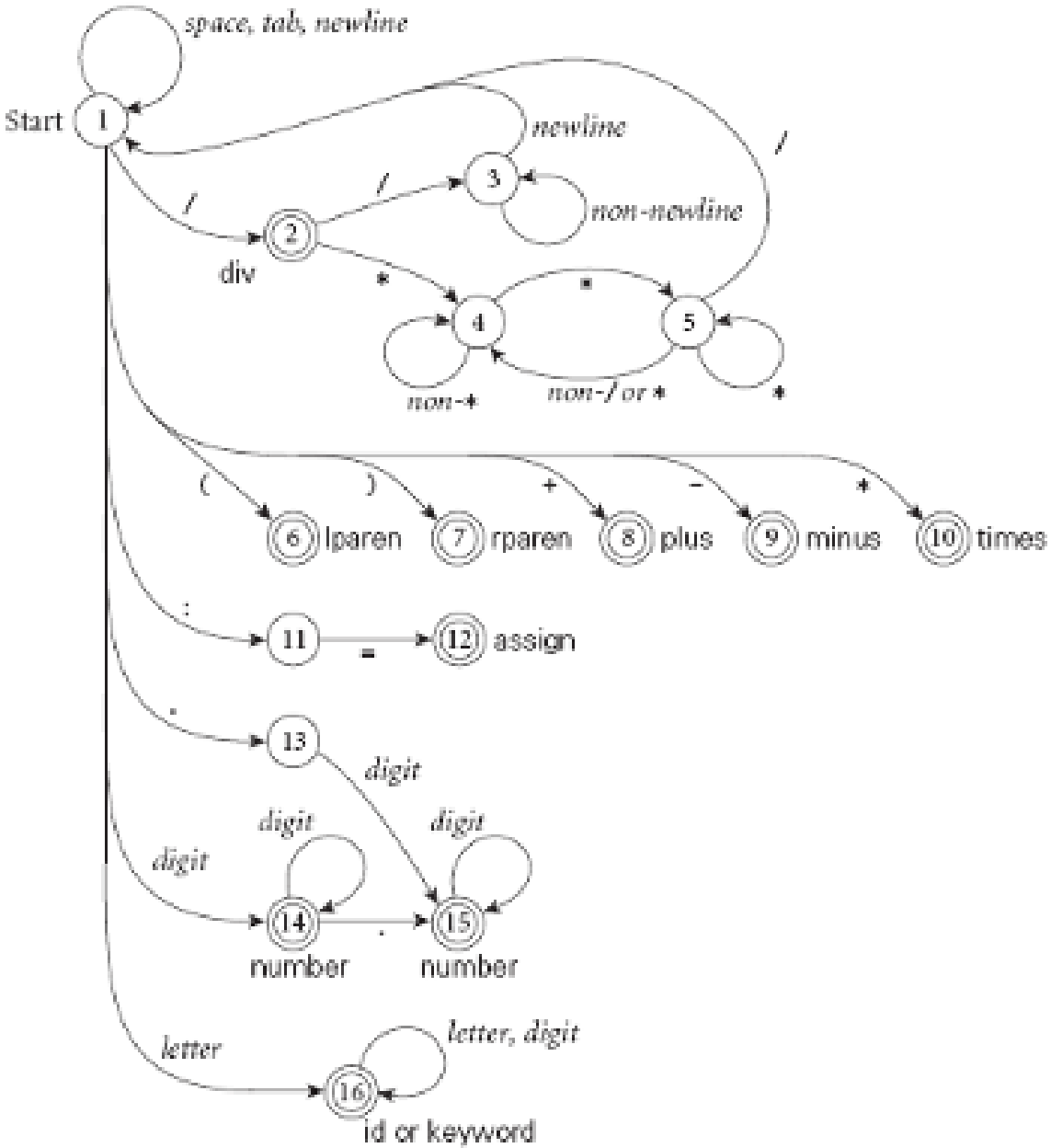
Problemi:

- quando termina un lessema, soluzione standard: la sequenza più lunga che appartiene una qualche espressione regolare,
  - la stringa '3.14e+sa' divisa in '3.14' 'e' '+' 'sa'
  - la stringa '3.14e+5a' divisa in '3.14e+5' 'a'
    per deciderlo posso essere necessari più simboli di lookahead;
- cosa fare se un lessema appartiene a più classi.

Come costruire uno scanner

- Costruisco un automa per ogni espressione regolare,
- sulla stringa di ingresso,
  - simulo il funzionamento parallelo degli automi,
  - riconosco un lessema quando nessun automa può continuare.

Automa per le componenti di espressioni aritmetiche



Generatori di scanner (analizzatori lessicali)

La costruzione degli scanner può essere automatizzata.

Classe di programmi che:

- dato
  - un insieme di espressioni regolari
  - delle corrispondenti azioni da compiere (codice da eseguire)
- costruiscono un programma che:
  - data una stringa,
  - riconosce i lessemi sulla stringa
  - su ogni lessema esegue l'azione corrispondente
    tipicamente costruire un token, ma anche altro.

(F)LEX

Diffuso generatore di scanner per Unix (linux)

Prende in input un file di testo con struttura

```
definizioni   (opzionale)
%%
regole
%%
funzioni ausiliari  (opzionale)
```

la parte regole, la più importante, serie di regole nella forma

```
espressione-regolare    azione
```

- espressioni-regolare sono quelle di unix (grep, emacs), ricca sintassi
- azione istruzione C, istruzioni multiple appaiono tra { } eseguita quando viene riconosciuta la corrispondente espressione (esistono strumenti equivalenti per gli altri ling. progr.)

Esempio

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

genera un programma che:

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

i caratteri non riconosciuti da nessuna espressione regolare vengono stampati in uscita (restano inalterati)

Regole

Nel codice delle regole posso usare le variabili:

- yytex : stringa (array) contenente il lessema riconosciuto, puntatore al primo carattere
- yyleng : lunghezza del lessema
- yyval : usate per passare parametri con il parser

Definizioni

`definizioni`

contiene la definizione di alcune espressioni regolari

nella forma

```
nome    espressione-regolare
```

esempio

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

notare le parentesi graffe in {digit}+

nomi definiti possono essere usati nelle regole

Sintassi espressioni regolari

- metacaratteri: `* | ( ) + ? [ ] - ^ . $ { } / \ " % < >`
- `{ide}` : identificatore di espressione regolare fa inserito tra { }
- `e{n,m}` : con n, m naturali: da n a m ripetizioni di e, anche `e{n,}`, `e{,n}`, `e{n}`
- `[^abd]` : tutti i caratteri esclusi a b d
- `\n` : newline, `\s` : spazio generico, `\t` : tab
- `\*` : il carattere `*`, `\` trasforma un metacarattere in un carattere standard
- `"a+b"` : la sequenza di caratteri `a+b` (+ non più metacarattere)
- `.` : tutti i caratteri meno newline
- `^` inizio riga
- $ fine riga

Funzioni ausiliare

- Nella parte ¨funzioni ausiliarie¨ definibile del codice C da usare nelle regole.
- Il codice C da inserire in testa al programma generato,
  - viene inserito nella parte ‘definizioni’
  - tra le parantesi ‘%{ }%’

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.

Uso standard

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

sostituisci lessema con token

Funzionamento

- Si considerano tutte le espressioni regolari e si seleziona quella con match più lungo, la parte lookahead conta nella misura
- a parità di lunghezza, conta l'ordine delle regole
- vengono impostate yytext, yyleng e eseguita l'azione
- nessun matching: regola di default: copio carattere input in output

Esempio: 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);
         }
%%
```

Esempio, 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 ausiliarie */
void main() {
  yylex();
  printf("There were %d characters in %d lines\n",
  charcount,linecount);
}
```

Funzioni base

Devono essere definite le funzioni:

- `yyerror(const char *str)` viene chiamata in condizioni di errore, tipicamente stampa messaggio errore usando la stringa argomento
- `yywrap()` viene chiama a fine file di input, tipicamente restituisce `0` o `1`
- `main()`

Con opportune opzioni, possono essere create le versioni default.

Uso

```
> flex sorgente.l
```

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

opzione ‘-ll’ necessaria per creare un programma stand-alone

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

Utilizzabile per automatizzare del text editing.

Analisi sintattica (Parsing) - Analizzatore sintattico (Parser)

A partire da

- una grammatica libera da contesto
- una stringa di token

costruisco

- l'albero di derivazione della stringa, a partire dal simbolo iniziale

Automi a pila - la teoria

Le grammatiche libere possono essere riconosciuti da automi a pila non deterministici.

- Automi con un uso limitato di memoria:
  - insieme finito di stati,
  - una pila, in cui inserire elementi finiti,
  - passo di computazione, in base a:
    - stato,
    - simbolo in testa alla pila,
    - simbolo in input
  - decido
    - nuovo stato,
    - sequenza di simboli da inserire nella pila, in sostituzione del presente
    - se consumare o meno l'input

Parola accettata se alla fine della scansione, si raggiunge una configurazione di accettazione:

- pila vuota
- stato finale

Deterministici – Non deterministici

Per gli automi a pila non vale l'equivalenza tra:

- deterministici (ad ogni passo una sola azione)
- non-deterministici (più alternative possibili)

Per le grammatiche libere sono necessari, in generale, automi non-deterministici

Complessità del riconoscimento

- Un automa a pila non deterministico, simulato tramite backtracking, porta a complessità esponenziali.
- Esistono due algoritmi [Earley, Cocke-Yunger-Kasami] capaci di riconoscere qualsiasi linguaggio liberi in tempo $O(n^3)$
- Un automa a pila deterministico risolve il problema in tempo lineare.

In pratica:

- complessità $O(n^3)$ non accettabile per un compilatore.
- ci si limita ai linguaggi riconoscibili da automi a pila deterministici:
- classe sufficientemente ricca da contenere quasi tutti i linguaggi di programmazione (C++ è un eccezione `x * y ;`)

Due tipi di automi a pila: LL e LR, due metodi di riconoscimento.

Automi LL(n)

Costruisce l´albero di derivazione in modo top-down:

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

Esempio di parsing, data la grammatica:

$$
\begin{array}{lcl}
S & \rightarrow & aAB \\
A & \rightarrow & C \mid D \\
B & \rightarrow & b \\
C & \rightarrow & c \mid \epsilon \\
D & \rightarrow & d
\end{array}
$$

la stringa adb viene riconosciuta con i seguenti passi:

Output	Pila	Input
Start	$S\$$	$adb\$$
$S \rightarrow aAB$	$aAB\$$	$adb\$$
	$AB\$$	$db\$$
$A \rightarrow D$	$DB\$$	$db\$$
$D \rightarrow d$	$dB\$$	$db\$$
	$B\$$	$b\$$
$B \rightarrow b$	$b\$$	$b\$$
	$\$$	$\$$
OK!		

Seconda derivazione

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

la stringa abb viene rifiutata con i seguenti passi:

Output	Pila	Input
Start	$S\$$	$abb\$$
$S \to aAB$	$aAB\$$	$abb\$$
	$AB\$$	$bb\$$
$A \to C$	$CB\$$	$bb\$$
$C \to \epsilon$	$B\$$	$bb\$$
$B \to b$	$b\$$	$bb\$$
	$\$$	$b\$$

Errore!

Più nel dettaglio

Parsing guidato da una tabelle che in base:

- al simbolo in testa alla pila
- ai primi n simboli di input non ancora consumati,

  normalmente n =1

Determina la prossima azione da svolgere,
tra queste possibilità:

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

Analizzatori LL

- La teoria verrà presentata nel corso di Linguaggi e Compilatori (laurea magistrale)
- Relativamente semplice capire la teoria e costruire automi (anche a mano, se a partire da semplici grammatiche)
- La costruzione prevede:
  - dei passaggi di riformulazione di una grammatica per ottenerne una equivalente (determina lo stesso linguaggio)
  - dalla nuova grammatica, un algoritmo determina:
    - se è LL(1)
    - la tabelle delle transizioni (descrizione dell'automa)
- Meno generali dell'altra classe di automi LR(n), quelli effettivamente usati nei tool costruttori di parser.

Significato del nome LL(n)

- esamina la stringa from Left to right
- costruisci la derivazione Leftmost
- usa n simboli di lookahead

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

- $S \rightarrow aAB \rightarrow aDB \rightarrow adB \rightarrow adb$

Una derivazione è destra (rightmost) se ad ogni passo espando sempre il non terminale più a destra

- $S \rightarrow aAB \rightarrow aAb \rightarrow aDb \rightarrow adb$

Gli automi LL(n) generano sempre la derivazione sinistra.

Analizzatori LR

Approccio bottom-up:

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

Esempio - Grammatica non LL

$$
\begin{aligned}
E &\rightarrow T \mid T + E \mid T - E \\
T &\rightarrow A \mid A * T \\
A &\rightarrow \mathbf{a} \mid \mathbf{b} \mid (E)
\end{aligned}
$$

	Pila	Input	Azione	Output
1	$\$$	$\mathbf{a} + \mathbf{b} * \mathbf{b}\$$	shift	
2	$\$\mathbf{a}$	$+ \mathbf{b} * \mathbf{b}\$$	reduce	$A \rightarrow \mathbf{a}$
3	$\$A$	$+ \mathbf{b} * \mathbf{b}\$$	reduce	$T \rightarrow A$
4	$\$T$	$+ \mathbf{b} * \mathbf{b}\$$	shift	
5	$\$T +$	$* \mathbf{b}\$$	shift	
6	$\$T + \mathbf{b}$	$* \mathbf{b}\$$	reduce	$A \rightarrow \mathbf{b}$
7	$\$T + A$	$* \mathbf{b}\$$	shift	
8	$\$T + A *$	$\mathbf{b}\$$	shift	
9	$\$T + A * \mathbf{b}$	$\$$	reduce	$A \rightarrow \mathbf{b}$
10	$\$T + A * A$	$\$$	reduce	$T \rightarrow A$
11	$\$T + A * T$	$\$$	reduce	$T \rightarrow A * T$
12	$\$T + T$	$\$$	reduce	$E \rightarrow T$
13	$\$T + E$	$\$$	reduce	$E \rightarrow T + E$
14	$\$E$	$\$$	stop	OK!

Più nel dettaglio

- Ad ogni passo si sceglie tra un azione di:
  - shift inserisco un token in input nella pila
  - reduce riduco la testa della pila applicando una riduzione al contrario
- Nella pila introduco una coppia <simbolo della grammatica, stato>

  l'azione da compiere viene decisa guardando
  - la componente stato in testa alla pila (non serve esaminarla oltre)
  - n simboli di input, per l'automa LR(n)

Automi LR

esiste un algoritmo che a partire da:

- grammatica libera L

mi permette di:

- stabilire se L è LR
- costruire l'automa a pila relativo
  - insieme degli strati
  - tabella delle transazioni

  come deve comportarsi l'automa ad ogni passo.

In realtà esistono tre possibili costruzioni.

Varie costruzioni per automi LR

Le costruzioni differiscono per:

- complessità della costruzione
- numero degli stati dell'automa pila generato
  complessità dell'algoritmo
- ampiezza dei linguaggi riconoscibili

In ordine crescente per complessità e ampiezza di linguaggi riconosciuti:

- SLR(n)
- LALR(n)
- LR(n)

n parametro, indica il numero di caratteri lookahead, crescendo n si ambia l'insieme di linguaggi riconosciuti.

Analizzatori LALR

Compromesso ideale tra numero di stati e varietà dei linguaggi riconosciuti

Costruzione piuttosto complessa: da spiegare e da implementare.

Esempio di applicazione di risultati teorici:

- Donald Knuth: 1965, parser LR,
  (troppi stati per i computer dell'epoca)
- Frank DeRemer: SLR and LALR,
  (pratici perché con meno stati).

LALR usato dai programmi generatori di parser:
Yacc, Bison, Happy.

Yacc (Yet Another Compiler-Compiler)

Generatore di parser tra i più diffusi:

Riceve in input una descrizione astratta del parser:

- descrizione di una grammatica libera
- un insieme di regole da applicare ad ogni riduzione

Restituisce in uscita:

- programma C che implementa il parser
  - l'input del programma sono token generati da uno scanner (f)lex
  - simula l'automa a pila LALR
  - calcola ricorsivamente un valore da associare a ogni simbolo inserito nella pila:
    - albero di derivazione,
    - altri valori

Programmi equivalenti per costruire parser in altri linguaggi:
ML, Ada, Pascal, Java, Python, Ruby, Go, Haskell, Erlang.

Struttura codice Yacc:

```
%{ prologo  %}
definizioni
%%
regole
%%
funzioni ausiliarie
```

Una produzione della forma

$\texttt{nonterm} \rightarrow \texttt{corpo\_1 | } \cdot \cdot \cdot \texttt{ | corpo\_k}$

diventa in Yacc con le regole:

```
nonterm : corpo_1   {azione semantica_1 }
          ...
          | corpo_k {azione semantica_k }
        ;
```

Azione semantica

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

`Ccode` Codice C che tipicamente

- a partire dai valori, calcolati in precedenza, per `num` e `fact`
- calcolo il valore da associare ad `exp`

Esempio Valutazione Espressioni Aritmetiche

Costruisco un programma che valuta

- una serie espressioni aritmetiche
- divise su più righe di input

espressioni composte da:

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

valgono le usuali regole di precedenza tra operazioni

Prologo e definizioni

```
/* PROLOGO */
%{
#include "lex.yy.c"

void yyerror(const char *str){
    fprintf(stderr,"errore: %s\n",str);}
int yywrap() {return 1;}
int main() { yyparse();}
%}
/* DEFINIZIONI */
%token NUM

%left '-' '+'
%left '*' '/'
%left NEG   /* meno unario */
```

Esempio - Regole

```
%% /* REGOLE E AZIONI SEMANTICHE */
          /* si inizia con il simbolo iniziale */
input:    /* empty */
        | input line
;
line :  '\n'
        | exp '\n' { printf("The value is %d \n", $1); }
;
exp : NUM   { $$=$1;    }
    | exp '+' exp   { $$=$1+$3; }
    | exp '-' exp   { $$=$1-$3; }
    | exp '*' exp   { $$=$1*$3; }
    | exp '/' exp   { $$=$1/$3; }
    | '-' exp %prec NEG { $$ =-$2;  }
    | '(' exp ')'   { $$ = $2;  }
;
```

Esempio - Codice LEX associato

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%%
[ \t]    ; // ignore all whitespace
[0-9]+       {yylval = atoi(yytext); return NUM;}
\n               {return *yytext;}
"+"              {return *yytext;}
[\-\*\/\(\)]     {return *yytext;}
```

Definizione dei token

- l'insieme dei possibili token definiti nel file Yacc con la dichiarazione `%token NUM`
- singoli caratteri possono essere token
  - non necessario definirli
  - codificati con in loro codice ASCII
  - gli altri token codificati con intero `>257`
- token diventano i terminali delle grammatica libera in Yacc
- Yacc crea un tabella `y.tab.h`
  - contiene la lista dei token e la loro codifica come interi
  - lex la dichiarazione
    #include "y.tab.h"
    accede ai dati in questa tabella

Funzioni ausiliare

Nel file Yacc è necessario definire le funzioni

- `yyerror` procedura invocata in caso di errori nelle sintassi dell'input
  - in input una stringa da usare nel messaggio di errore
- `yywrap` chiamata al termine del file di input
  - di default restituisce l'intero 0 o 1
  - può essere usata per gestire più file di input,
- `main` per creare un programma stand alone

La compilazione YACC crea una funzione C

- `yyparser` che implementa il parser LALR

Integrazione tra lex e Yacc (flex e Bison)

- lex non crea un programma stand alone ma una funzione `yylex()` chiamata all'interno di `yyparser`
- `yylex()` restituisce un token ad ogni chiamata e, indirettamente, un valore
  - token è il valore restituito esplicitamente dalla funzione:
    - intero, codifica la classe del lessema
  - token diventato i terminali della grammatica usata in Yacc
  - il valore attraverso la variabile `yylval` globale e condivisa

Differenze tra YACC e LALR

YACC produce codice C che implementata un automa LALR ma:

- non esiste uno stretto controllo che la grammatica sia LALR:
  - grammatiche non LALR vengono accettate ma:
    - si costruisce un automa a pila dove per alcuni casi più scelte possibili (automa non-deterministico)
    - YACC genera codice che ne sceglie una, esegue solo quella
    - si possono fare scelte sbagliate
    - si possono rifiutare parole valide
- grammatiche ambigue possono essere usate,
  - attraverso definizione di priorità si possono togliere ambiguità,
  - automi non LALR: attraverso le priorità si indicano le scelte da fare

Differenze tra YACC e LALR

- il codice C, non solo riconosce la stringa ma la "valuta¨
  - valutazione bottom-up - come il riconoscimento,
  - risultati parziali inserite nella pila, `$$`, `$1`, `$2`, . . . fanno riferimento alla pila
  - la parte “azioni” delle regole specificano che valore associare una parola
    a partire dai valori delle sotto-parole.
  - attraverso le azioni posso costruire l’albero di derivazione ma anche altro

Secondo esempio: sintassi di comandi ad un termostato.

File LEX

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+              { yylval=atoi(yytext); return NUMERO; }
riscaldamento       return TOKRISCALDAMENTO;
acceso|spento       { yylval=strcmp(yytext,"spento");
                      return STATO;}
obiettivo           return TOKOBIETTIVO;
temperatura         return TOKTEMP;
[ \t\n]+            /* ignora spazi bianchi e fine linea */;
%%
```

File Yacc Prologo e dichiarazioni

```
%{
#include <stdio.h>
#include <string.h>

void yyerror(const char *str)
{ fprintf(stderr,"errore: %s\n",str);}
int yywrap() {return 1;}
int main() { yyparse();}
%}

%token NUMERO TOKRISCALDAMENTO STATO TOKOBIETTIVO TOKTEMP

%%
```

File YACC: regole

```
comandi: /* vuoto */
        | comandi comando
        ;

comando: interruttore_riscaldamento
        |  imposta_obiettivo
        ;
interruttore_riscaldamento: TOKRISCALDAMENTO STATO
        { if($2)     printf("\t Riscaldamento acceso \n");
          else       printf("\t Riscaldamento spento \n"); }
        ;
imposta_obiettivo:  TOKOBIETTIVO TOKTEMP NUMERO
        { printf("\t Temperatura impostata a %d \n",$3); }
        ;
```

Modifica tipo YYSTYPE

Nel file C le variabili `yylval`, `$$`, `$1`, `...` hanno tipo `YYSTYPE`
- per default intero
- posso YYSTYPE con la dichiarazione

```
%union {
nodeType        *expr;
int             value;
}
```

definisce un tipo `union` associato alla variabile LEX (`yyval`)
e alle variabile YACC (`$$`, `$1`, `...`)

con le seguenti dichiarazioni, specifico a che tipi componenti sono associati diversi terminali e non terminali.

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

dichiarazioni da inserire nella parte definizioni (prologo) del file YACC.

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`

Nomi e ambiente

Blocchi e regole di scope

Nomi

Meccanismi di astrazione

- astrarre dalla macchina fisica
- nascondere dettagli, mettere in evidenza le parti importanti

fondamentali per gestire la complessità del software,

Uso dei nomi: un meccanismo di astrazione base,
presente già in assembly

Nome: sequenza di caratteri usata per denotare qualche cos'altro.

Permettono una rappresentazione sintetica, astratta, mnemonica di:

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

Sintassi

I nomi possono essere sequenze di caratteri significativi,

non solo identificatori, ma anche simboli semplici

```
+   -   <=   ++
```

Esempio

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

Nomi e oggetti denotabili

- Bisogna distingue tra:
  - nome (stringa di caratteri),
  - l'oggetto rappresentato, denotato.
- In linguistica: significante e significato
- Oggetto denotabile: oggetto associabile un nome,

  - `linguaggi programmazione diversi possono avere oggetti denotabili diversi`

Legame, ambiente.

- Legame (binding): associazione esistente tra nome e oggetto
- Ambiente (environment): insieme dei legami esistenti
  dipende da:
  - uno specifico punto del programma

  può dipendere:
  - dal codice eseguito in precedenza, storia del programma.

Creazione del binding

Possiamo separare tra:

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

Creazione del binding

Il binding può essere creato in vari momenti:

- definizione del linguaggio,
- scrittura del codice,
- caricamento del programma in memoria,
- esecuzione,
- ...

Ma distinguiamo principalmente tra:

- binding statico prima dell'esecuzione della prima istruzione
- binding dinamico durante l'esecuzione

Ma anche l'oggetto denotato viene incrementalmente definito in vari momenti,

- es. dichiarazione di una variabile

```
int a
```

Ambiente e store:

Il valore di una variabile x dipende da due funzioni:

- ambiente: definisce quale locazione di memoria di memoria contiene il dato di x
- store (memoria): determina il dato effettivo.

Accesso al valore in due passaggi.

- I comandi posso modificare lo store (assegnazione), ma non l'ambiente.
- Ambiente modificabile attraverso dichiarazioni.

Alcuni linguaggi non prevedono l'esistenza di uno store (funzionali puri).

Dichiarazione:

Nomi e legami i quasi sempre definiti attraverso

Dichiarazioni: meccanismo (implicito o esplicito) col quale si crea un legame (si modifica l'ambiente)

```
int x = 0;
typedef int T;
int inc (T x) {
    return x + 1;
}
```

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

Blocchi

Nei linguaggi moderni l'ambiente è strutturato

- Blocco: regione programma che può contenere dichiarazioni locali a quella regione

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

Possono essere:

- associati ad una procedura
- anonimi (o in-line)

Vantaggi i blocchi

Gestione locale dei nomi:

```
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

- definire nomi locali indipendenti da altre dichiarazioni.
- strutturare il programma,

Con un'opportuna allocazione della memoria (vedi dopo):

- ottimizzano l'occupazione di memoria
- permettono la ricorsione

Annidamento

I blocchi possono essere annidati:

```
{
    int x = 0;
    int y = 2;
    {
        int x = 1;
        x = y;
    }
    write(x);
}
```

Regola di visibilità:

- 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.

Esempio

```
A:{
    int a = 1;
    B:{
        int b = 2;
        int c = 3;
        C:{
            int c=4;
            int d = a+b+c;
            write(d);
        }
    }
    D:{
        int e = a+b+c;
        write(e);
    }
}
```

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

Dichiarazioni come

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

possono generare errore oppure no

Validità di una dichiarazione,

- a partire dalla dichiarzione
- 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)
- tipicamene a `b` viene assegnato `1` (C, Java)
- può essere assegnato `2`

Suddividiamo l'ambiente

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

- ambiente locale: associazioni create all'ingresso nel blocco
  - variabili locali
  - parametri formali
- ambiente globale relativa al programma principale:
  - dichiarazioni esplicite di variabili globali
  - associazioni esportate da moduli ecc.
- ambiente non-locale, non-globale :
  - associazioni ereditate da altri blocchi

Operazioni sull'ambiente

- Creazione associazione nome-oggetto denotato (naming)
  - entrata in un blocco, dichiarazione locale in blocco
- Distruzione associazione nome-oggetto denotato (unnaming)
  - uscita da blocco con dichiarazione locale
- Riferimento oggetto denotato mediante il suo nome (referencing)
  - uso di un nome, nel codice,
- Disattivazione associazione nome-oggetto denotato
  - entrata in un blocco con dichiarazione che maschera
- Riattivazione associazione nome-oggetto denotato
  - uscita da blocco con dichiarazione che maschera

Operazioni sugli oggetti denotabili

- Creazione
- 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 denotabile, come costanti, tipi, non vengono ne creati ne distrutti.

- creazione, distruzione fanno riferimento a dati in memoria (variabili), codice (procedure)

Eventi base in un binding

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

- 1-7 tempo di vita dell'oggetto
- 2-6 tempo di vita dell'associazione

Tempo di vita

La vita di un oggetto non necessariamente coincide con la vita dei legami per quell'oggetto

Vita dell'oggetto più lunga di quella del legame:

- passaggio per riferimento, di una variabile in una procedura

```
var A:integer;
procedure P (var X:integer);
   begin ...
   end;
...
P(A);
```

L'esecuzione di P crea un legame tra X e un oggetto esistente prima e dopo l'esecuzione.

Tempo di vita

- Stesso esempio in C

```
int A
void P(int *X){
   ...
}
...
P(&A)
```

Mascheramento:

- un ulterioree esempio di oggetto che sopravvive al legame
- in questo caso il legame si disattiva ma non sparisce completamente.

Tempo di vita

Vita dell'oggetto più breve di quella del legame:

- puntatore ad area di memoria dinamica deallocata.

```
int *X, *Y, z;
X = (int *) malloc (sizeof (int));
Y = X;
*Y = 5;
free (X);
z = *Y;
```

- Dangling reference: riferimenti pendenti, causa di errori.

Regole di scope

In presenza di procedure

le regole di scope diventano più complesse.

```
int x=10;
void incx () {
   x++;
}
void foo () {
   int x = 0;
   incx();
}
int main () {
foo();
write(x);
}
```

- quale `x` incrementa `incx`?

Due alternative

Un nome non-locale alla procedure `incx` fa riferimento a:

- alla prima dichiarazione in blocco che include sintatticamente `incx`
- all'ultima dichiarazione "eseguita" prima di `incx`

Scope statico

Prima alternativa:

- un nome non locale è risolto nei blocchi che testualmente lo racchiudono a partire da quelli più interni:

```
int x = 0;
void foo (int n) {
    x = x + n;
}
foo(2);
write(x);
{
    int x = 0;
    foo (3);
    write (x);
}
write (x);
```

Scope statico

Prima alternativa:

- un nome non locale è risolto nei blocchi che testualmente lo racchiudono a partire da quelli più interni:

```
int x = 0;
void foo (int n) {
    x = x + n;
}
foo(2);
write(x);
{
    int x = 0;
    foo (3);
    write (x);
}
write (x);
```

Stampa: 2 0 5

Scope dinamico

- un nome non locale è risolto usando la sequenza di blocchi attivi, a partire da quelli attivati più recentemente

```
int x = 0;
void foo (int n) {
    x = x + n;
}
foo(2);
write(x);
{
    int x = 0;
    foo (3);
    write (x);
}
write (x);
```

Scope dinamico

- un nome non locale è risolto usando la sequenza di blocchi attivi, a partire da quelli attivati più recentemente

```
int x = 0;
void foo (int n) {
    x = x + n;
}
foo(2);
write(x);
{
    int x = 0;
    foo (3);
    write (x);
}
write (x);
```

Output: 2 3 2

Scope statico: indipendenza dalla posizione

```
int x=10;
void incx () {
   x++;     }
void foo (){
   int x = 0;
   incx();  }
foo();
```

- `incx` è dichiarato nello scope della `x` più esterna,
- `incx` è chiamato nello scope della `x` più interna,
- `incx` può essere chiamata in molti contesti diversi,
- l'unico modo per `incx` di comportarsi in modo uniforme è che il riferimento a `x` nelle due chiamate di incx sia sempre quello più esterno (scope statico).

Scope statico: indipendenza dai nomi locali

```
int x=10;
void incx () {
   x++;    }
void foo (){
   int y =0;
   incx();   }
foo();
```

In questa seconda versione `foo` usa un diversa variabile locale

- `y` invece di `x`.

Questa modifica di nomi di variabili locali:

- modifica il comportamento del programma con lo scope dinamico,
- non ha alcun effetto in scope statico.

Scope dinamico: specializzare una funzione

Supponiamo che “visualizza” sia una procedura che

- rende a colore sul video un certo testo
- usa come parametro una costante non-locale “colore”

Con scope dinamico posso modificare il suo comportamento con:

```
{
const colore = rosso;
visualizza(testo);
}
```

Tutte le variabili non locali diventano dei parametri impliciti della procedura:

- maggiore flessibilità nell’uso delle procedure
  - simulo il meccanismo dei “valori di default”
- maggiore difficoltà nel determinare il comportamento della procedura

Scope statico vs dinamico

Scope 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, ...

Scope 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 piuttosto complesse
- Lisp (alcune versioni), Perl, APL, Snobol, ...

Aliasing

Nomi diversi denotano lo stesso oggetto, causato da:

- passaggio dei parametri a una procedura per riferimento:

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

- puntatori:

```
int *X, *Y;
X = (int *) malloc (sizeof (int));
*X = 0;
Y = X;
*Y = 1;
write(*X);
```

Overloading

Lo stesso nome può avere significati diversi a seconda del contesto

(accade spesso nei linguaggi naturali)

Normalmente, per i nomi di alcune funzioni predefinite:

- somma '+' è un classico esempio:
  - somma tra interi
  - somma floating-point
  - concatenazione tra stringhe

Il contesto, tipo degli argomenti, determinano il significato corretto.

Overloading costituisce una forma di polimorfismo.

Determinare l'ambiente

L'ambiente è dunque determinato da:

- regola di scope (statico o dinamico)
- regole specifiche, p.e.
  - quando è visibile una dichiarazione nel blocco in cui compare?

Più avanti discuteremo:

- regole per il passaggio dei parametri
- regole di binding (shallow o deep)
  - intervengono quando una procedura P è passata come parametro ad un'altra procedura mediante il formale X

Costrutto let in Scheme

Scheme permette dichiarazioni attraverso diversi costrutti:

- `let`
- `let*`
- `letrec`

con diversa validità delle dichiarazioni.

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

Esempio

- Qual'è la valutazione di:

```
(let ((a 1))
  (let ((a 2)
        (b a))
    b))
```

- e di:

```
(let ((a 1))
  (let* ((a 2)
         (b a))
    b))
```

Esempio

- Qual'è la valutazione di:

```
(let ((a 1))
  (let ((a 2)
        (b a))
    b))
```

- e di:

```
(let ((a 1))
  (let* ((a 2)
         (b a))
    b))
```

rispettivamente 1 e 2

Mutua ricorsione (di funzioni)

forza l'utilizzo di un nome prima che questi venga dichiarato,

per essere possibile, devo permettere eccezioni al vincolo:

- un nome deve essere dichiarato prima di essere usato

Java: dichiarazione di metodi

```
{void f(){
    ...
    g(); // g non ancora dichiarato
    ...
}
void g(){
     ...
     f();
     ...
}
...
```

Mutua ricorsione di definizione di tipo

Pascal per tipi puntatore:

```
type lista = ^elem;
     elem = record
          info : integer;
               next : lista;
          end
```

posso definire un tipo puntatore prima di aver definito il tipo puntato.

In C

```
struct child {
  struct parent *pParent;
};
struct parent {
  struct child *children[2];};
```

in `struct` posso inserire un campo puntatore a `struct` non ancora definiti

Dichiarazioni incomplete di tipo:

in alcuni linguaggi permettono la mutua ricorsione.

- In C:

```
typedef struct elem element;
struct elem {
    int info;
    element *next;
}

typedef struct child ch;
struct child {
  struct parent *pParent;
};
struct parent {
  ch *children[2];
};
```

Dichiarazioni incomplete di tipo:

In Ada

```
type elem;
type lista is access elem;
type elem is record
             info: integer;
             next: lista;
     end
```

Dichiarazioni incomplete di funzione

In C:

```
void eval_tree(tree t);  /* solo dichiarazione */

void eval_forest(forest f){
    ...
    eval_tree(t2);
}
void eval_tree(tree t){ /* definizione completa */
    ...
    eval_forest(f2);
}
```

La prima dichiarazione incompleta di eval_tree

serve in eval_forest a sapere come usare

eval_tree: che tipo di parametri passare, che risultato restituisce.

Moduli e information hiding

Programmi di grosse dimensioni pongono il problema di nascondere parte dei nomi.

- usare per caso lo stesso nome in due parti codice porta a comportamenti imprevedibili
- evitare conflitti comporta un cognitive overloading,

I blocchi annidati (e procedure) risolvono il problema ma non completamente.

Gestione della memoria

Stack di attivazione, Heap

La gestione della memoria

Come il compilatore-interprete, organizza i dati necessari all'esecuzione del programma.

Alcuni aspetti organizzativi già visti nel corso in assembly

Sia per codice macchina

- scritto a mano o
- generato da compilatori

struttura la memoria in memoria in modo simile.

Uso della Memoria RAM

Nell'uso tipico, codice ARM prevede divisione della memoria nei seguenti intervalli consecutivi.

- `0 - 0xFFF`: riservata al sistema operativo.
- `0x1000 - ww` : codice programma (`.text`),
- `ww - xx` : costanti, variabili programma principale (`.data`),
- `xx - yy` : heap per dati dinamici (liste, alberi)
- `yy - zz` : stack per chiamate di procedura, stack di attivazione.

Il registro `r13, sp`, (stack pointer) punta alla cima dello stack

Il registro `r11, fp`, (frame pointer) punta al "frame" della procedura in esecuzione.

Tipi di allocazione della memoria

Tre meccanismi di allocazione della memoria:

- statica: memoria allocata a tempo di compilazione
- dinamica: memoria allocata a tempo d'esecuzione, divisa in:
  - pila (stack): oggetti allocati con politica LIFO
  - heap: oggetti allocati e de-allocati in qualsiasi momento

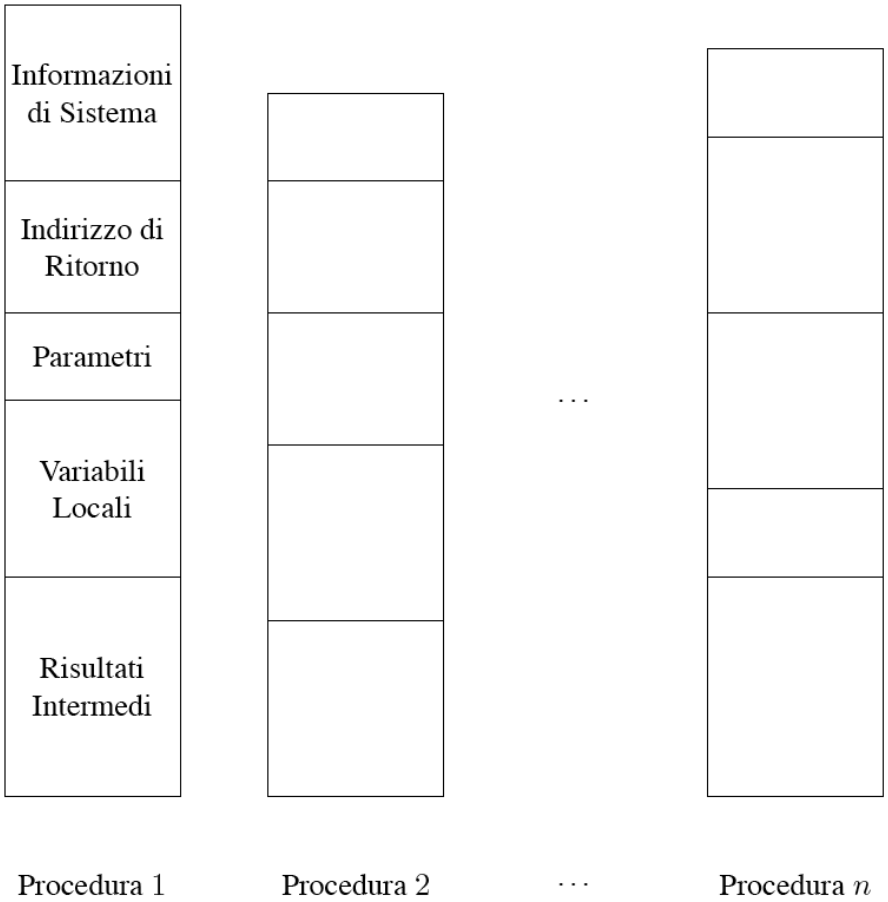
Normalmente un programma usati tutti e tre i meccanismi.

Allocazione statica

- Gli oggetti hanno un indirizzo assoluto, mantenuto per tutta la durata dell'esecuzione.
- Solitamente sono allocati staticamente:
  - variabili globali
  - costanti determinabili a tempo di compilazione
  - tabelle usate dal supporto a run-time (per type checking, garbage collection, ecc.)
- Alcuni linguaggi di programmazione (vecchie versione Fortran), usano l'allocazione statica per tutti i dati, in questo caso:

Allocazione statica del programma

- Ad ogni variabile (locale o globale) assegnato un indirizzo univoco.
- Le variabili locali di una procedura mantengono il valore anche dopo la fine della procedura.



Svantaggi della allocazione statica completa.

Non permette la ricorsione:

- 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).

Forza ad usare più spazio del necessario:

- costringe ad allocare spazio per tutte le variabili di tutto il codice,
- di volta in volta, solo una piccola parte di queste sono attiva (quelle associate alle procedure aperte).

Non permette strutture dati dinamiche.

Vantaggi: accesso diretto, veloce, a tutte le variabili.

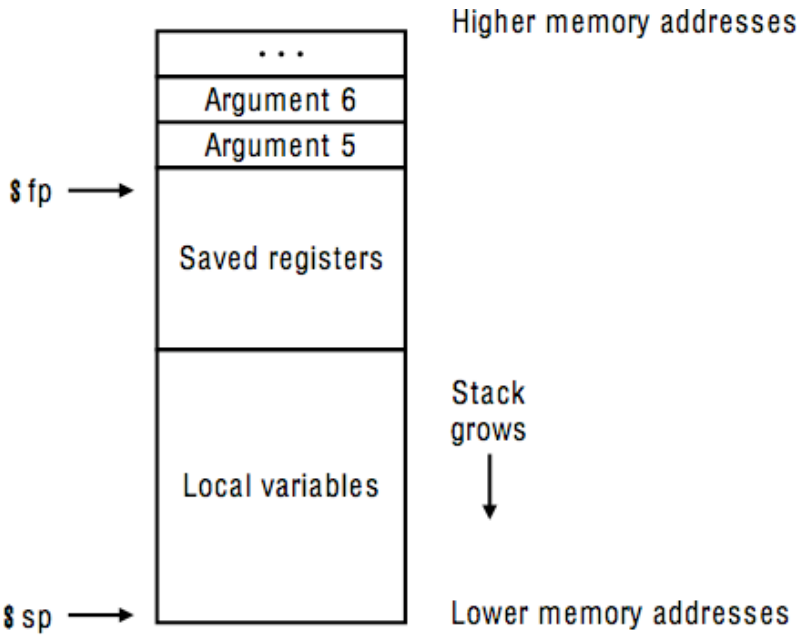
Stack di attivazione

Parte di memoria gestita come una pila

destinata contenere i dati locali alle procedure

- ad ogni chiamata, allocato dello spazio, record di attivazione
- spazio de-allocato all'uscita dalla procedura

Formato di un record di attivazione:



Allocazione dinamica: record di attivazione.

- Ogni istanza di procedura in esecuzione possiede un record di attivazione (RdA o frame), spazio di memoria per contenere:
  - variabili locali
  - parametri in ingresso e in uscita
  - indirizzo di ritorno
  - link dinamico (al frame della procedura chiamante)
  - link statico (al frame della procedura genitore) (non sempre presente)
  - risultati intermedi
  - salvataggio dei registri

Stack di attivazione

- Analogamente, ogni blocco, con dichiarazione, può avere un suo record di attivazione (più semplice, meno informazioni di controllo)
  - variabili locali
  - risultati intermedi
  - link dinamico (al frame della procedura chiamante)
- Stack di attivazione:
  - una pila (LIFO) contenente i RdA
  - la struttura dati naturale per gestirli.

Esempio: gestione della memoria con blocchi anonimi

```
{   int x=0;
    int y=x+1;
    { int z=(x+y)*(x-y);
    };
};
```

- Push record con spazio per `x`, `y`
- Setta valori di `x`, `y`
  - Push record blocco interno
    spazio per `z` e, eventualmente, risultati intermedi
  - Setta valore per `z`
  - Pop record per blocco interno
- Pop record per blocco esterno
- Nota: nel blocco interno l'accesso alle variabili non locali `x` e `y`, vanno cercate in blocchi esterni (risalire la catena dinamica)

Accesso ai dati nell stack di attivazione - versione semplificata per blocchi anonimi

Dati (variabili, risultati intermedi) locali del blocco in esecuzione, si usa

- Frame Pointer (FP):
  - punta all´indirizzo base dell'ultimo frame (RdA)
  - i dati dell'ultimo frame sono accessibili per offset rispetto allo FP:
    - indirizzo_dato = FP + offset
    - offset determinabile staticamente, a tempo di compilazione

Dati non locali al blocco in esecuzione

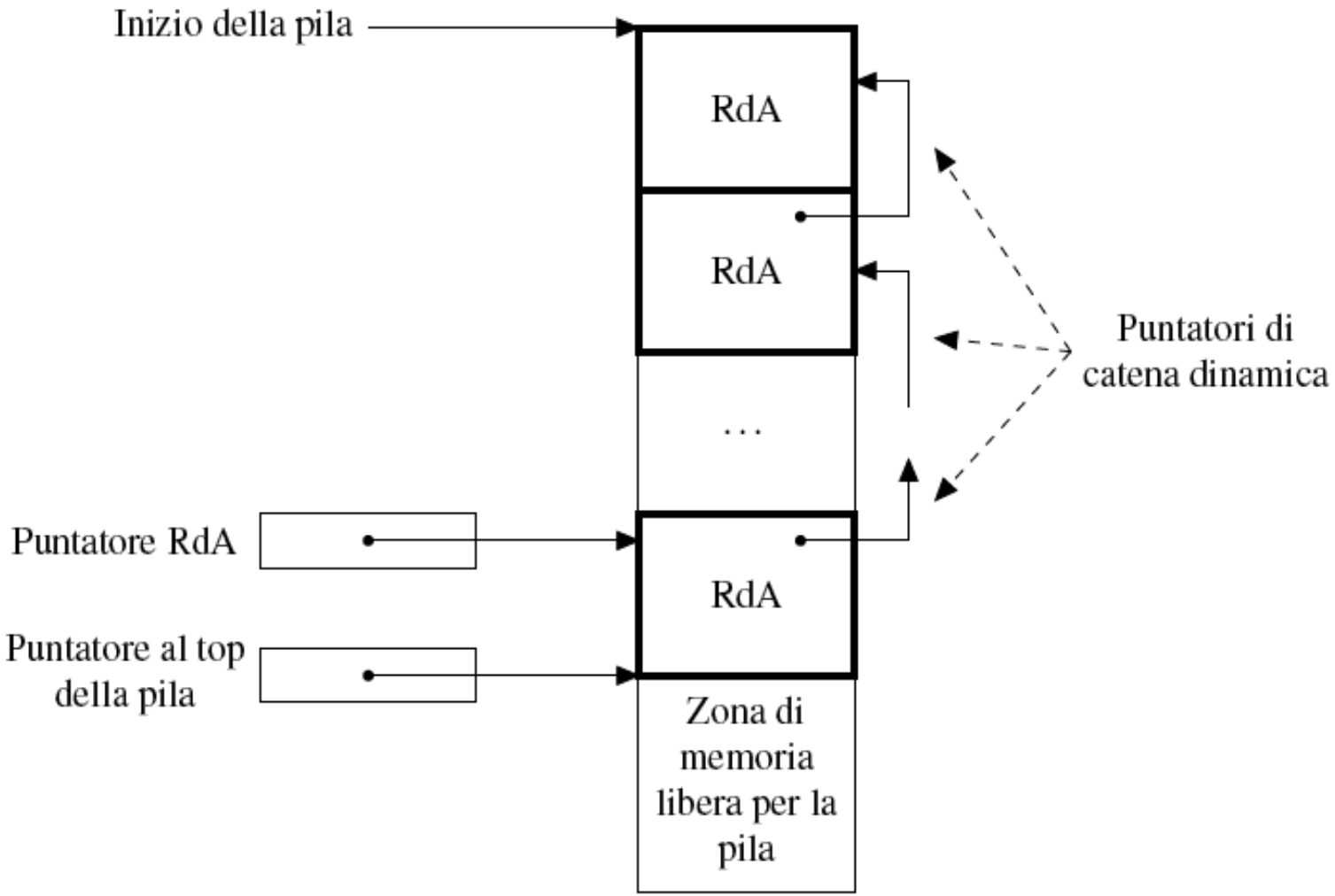
- Necessario determinare l'indirizzo base del RdA del dato,
- Link dinamico (o control link) (Puntatore di Catena Dinamica)
  - puntatore al precedente record sullo stack
- Risalgo la catane dei link

Gestione della pila

Per la gestione dello stack di attivazione uso anche

- Stack Pointer (SP)
  - punta alla fine della pila, primo spazio di memoria disponibile

Gestione della pila



Gestione dei blocchi in-line

Operazioni per la gestione (aggiornamento di FP, SP e link dinamici)

- Ingresso nel blocco: Push
  allocazione dello spazio e scrittura dei link
  - link dinamico del nuovo RdA := FP
  - FP = SP
  - SP = SP + dimensione nuovo RdA
- Uscita dal blocco: Pop, Elimina l'ultimo RdA
  recupera spazio e ripristina puntatori
  - SP = FP
  - FP := link dinamico del RdA tolto dallo stack

Nella pratica

In molti linguaggi si preferisce evitare l'implementazione a pila per i blocchi anonimi:

- tutte le dichiarazioni dei blocchi annidati sono raccolte dal compilatore
- viene allocato spazio per tutti i blocchi
- spreco di memoria
- maggiore velocità: meno azioni sulla pila

Esempio di ricorsione, stack di attivazione indispensabile.

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

Nel RdA troviamo;

- parametri ingressi: n
- parametri in uscita - risultato: fact(n)
- link statici, dinamici e indirizzo di ritorno, back_up registri processore

Tanti RdA quanto il valore iniziale di n

Gestione dello stack di attivazione

Chiamata procedura

- allocazione RdA
- passaggio parametri
- inizializzazione variabili locali
- gestione informazione di controllo: link statici, dinamici
- registri: salvataggio

Uscita procedura

- passaggio risultato
- gestione informazine di controllo:
- registri: ripristino
- deallocazione RdA

Gestione dello stack di attivazione

La gestione della pila è compiuta mediante:

- sequenza di chiamata - chiamante
- prologo - chiamato
- epilogo - chiamato
- sequenza di ritorno - chiamante

La ripartizione tra chiamante e chiamato, è in parte arbitraria

- inserire il codice nella procedura chiamata porta a generare meno codice
  - istruzione presenti una volta sola, al posto di tante volte quante chiamate alla procedura.

Allocazione dinamica con heap

- Heap: zona di memoria le cui parti (blocchi) possono essere allocate (e de-allocate) a seconda della necessità
- Necessario quando il linguaggio permette:
  - tipi di dato dinamici
  - oggetti di dimensioni variabili
  - oggetti che sopravvivono alla procedura che gli ha creati.
- La gestione dello heap,
  problemi:
  - gestione efficiente dello spazio: frammentazione
  - velocità di ricerca spazi liberi

Heap suddiviso in blocchi di dimensione fissa

- in origine: tutti i blocchi collegati nella lista libera
- allocazione di uno o più blocchi
- de-allocazione: restituzione alla lista libera
- il vincolo della dimensione fissa, rende il meccanismo troppo rigido:
  - non fornisce blocchi di elevata dimensione per strutture dati contigue di dimensione elevata.
  - non posso implementare `malloc` di C

Heap: blocchi di dimensione variabile

- inizialmente: lista libera costituita ad unico blocco,
  poi lista formata da blocchi di dimensione variabile
- allocazione: determinare un blocco libero della dimensione opportuna
  - che viene diviso in:
    - parte assegnata
    - resto blocco libero
- de-allocazione: restituzione del blocco aggiunta alla lista dei blocchi liberi

Gestione della heap

Problemi:

- le operazioni devono essere efficienti
- evitare lo spreco di memoria
  - frammentazione interna
  - frammentazione esterna

Frammentazione: presente anche nella gestione memoria virtuale tramite segmentazione

- su uno spazio di memoria lineare
- vengono assegnati blocchi di dimensione variabile
- che, dopo un certo tempo, possono liberarti.

Le due soluzioni hanno problemi simili,

Frammentazione

- Frammentazione interna: lo spazio richiesto è X,
  - viene allocato un blocco di dimensione Y > X,
  - lo spazio Y-X è sprecato,
  - meno problematica della:
- Frammentazione esterna:
  - la continua allocazione e deallocazione di blocchi porta alle creazioni di blocchi liberi di piccole dimensioni
    - differenze tra il blocco libero usato e lo spazio effettivamente allocato
  - spazio sprecato per l'esistenza di piccoli difficilmente usabili, non è possibile allocare un blocco di grandi dimensioni, anche con tanta memoria libera

Gestione della lista libera: unica lista

- Ad ogni richiesta di allocazione: cerca blocco di dimensione opportuna
  - first fit: primo blocco grande abbastanza
  - best fit: quello di dimensione più piccola, tra quelli sufficienti.
- Se il blocco scelto è abbastanza più grande di quello che serve viene diviso in due e la parte inutilizzata è aggiunta alla LL
- Quando un blocco è de-allocato, viene restituito alla LL
  (se un blocco adiacente è libero i due blocchi sono “fusi” in un unico blocco).

Gestione lista libera: First fit o Best Fit ?

Vantaggi - Svantaggi:

- First fit: più veloce, occupazione memoria peggiore
- Best fit: più lento, occupazione memoria migliore

Gestione della lista libera: più liste

- Con unica LL costo allocazione lineare nel numero di blocchi liberi
- liste libere multiple, per migliore i tempi di ricerca:
  - ogni lista contiene blocchi liberi di dimensione simile.
  - prendo il primo blocco disponibile nella lista opportuna

Buddy system: n liste;

- la lista k-esima ha blocchi di dimensione 2^k
- se si desidera un blocco di dimensione 2^j, e la lista relativa è vuota:
  - si cerca un blocco, nella lista successiva, di dimensione doppia, viene diviso in due parti,
  - se anche la lista successiva è vuota, la procedura si ripete ricorsivamente.
- quando un blocco di 2^k e' de-allocato, viene è riunito alla sua altra metà (buddy - compare), se disponibile

Fibonacci heap simile, ma i blocchi hanno dimensioni numeri di Fibonacci,

- maggiore scelta di dimensione, in numeri di Fibonacci crescono più lentamente delle potenze di 2
- minore frammentazione interna

Implementazione delle regole di scope.

Come recuperare dati non locali nello stack di attivazione?

- Scope statico:
  - catena statica
  - display
- Scope dinamico:
  - solo SdA
  - A-list
  - Tabella centrale dell'ambiente (CRT)

Scope statico: presentazione del problema

Consideriamo il programma:

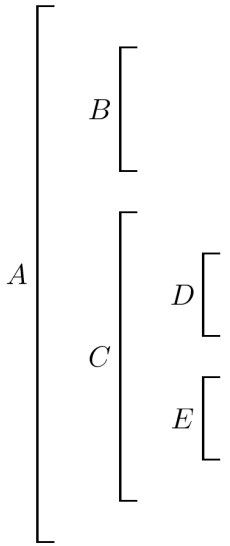
```
int x=10;
void incx () {
   x++;    }
void foo (){
   int x = 0;
   incx();   }
foo();
incx();
```

- `incx` viene chiamato due volte: indirettamente tramite `foo`, direttamente;
- `incx` accede sempre alla stessa variabile `x`
- `x` è memorizzato in un certo RdA
- problema: determinare di quanti RdA scendere nella pila,

Scope statico: legami validi

I legami validi sono quelli definiti:

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



- In D, i legami validi sono quelli definiti in D, C, A
- le altre dichiarazioni non sono visibili

Scope statico

A tempo di compilazione, per ogni variabile `x` in ogni procedura `P` so:

- 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, nella procedura `P` per accedere a `x` devo

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

Link statico

Per poter accedere ai RdA degli antenati:

- ogni RdA contiene un puntatore all'ultimo RdA del genitore.
- chiamato link statico:

Attraverso il link statici definiscono catena statica: la lista dei RdA degli antenati.

Nota: per ogni variabile `x` non locale ad una procedura `D`:

- il numero di link statici da seguire per trovare il RdA contenente `x` è determinabile a tempo di compilazione,
  resta costante, in ogni ricerca.
- la posizione di `x` nel RdA così trovato è fissa:
  - a tempo di compilazione si determina la posizione
  - non serve memorizzare il nome delle variabili negli RdA,

Riassumendo:

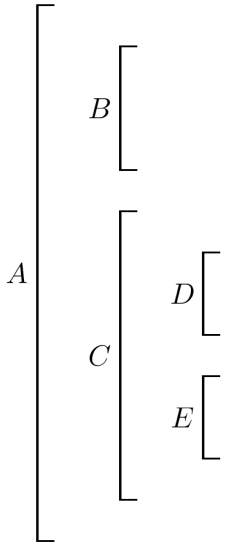
- link dinamico (procedura chiamante) dipende dalla sequenza di esecuzione del programma,

  definisce la catena dinamica,

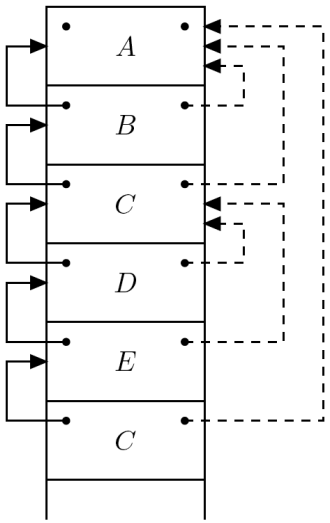
- link statico (procedura genitore) dipende dalla struttura delle dichiarazioni di procedure

  definisce la catena statica

Esempio: programma con struttura a blocchi



La sequenza di chiamate A, B, C, D, E, C, genera la pila:



Esempio esecuzione.

```
int x = 10;
void A(){
    x=x+1;}
void B(){
   int x = 5;
   void C (){
       x=x+2; A();
   }
   A(); C();
}
B();
```

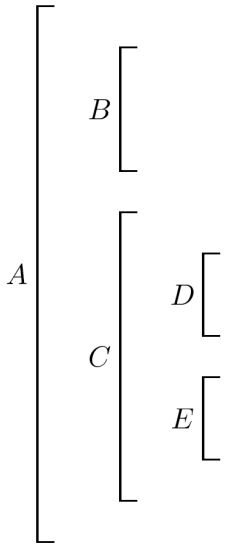
Come si crea il link statico?

La procedura chiamante `Ch` determinare il link statico della procedura chiamata `S`,

- lo passa ad `S`, che lo inserisce nel suo RdA.
- come `Ch` determina il link statico da passare a `S`?
  - quando `Ch` chiama `S` sa se la definizione di `S` è:
    - nelle dichiarazioni di `Ch` (profondità k=0) passa il Frame pointer come link statico
    - nelle dichiarazioni di un n-esimo avo di `Ch`, (profondità k = n) percorre la catena statica per n passi, per determinare il link statico da passare
    - in altre posizione `S` non sarebbe visibile

Esempi di profondita relativa k

Nel caso di un programma con struttura



La procedura C, può chiamare tutte la altre con profondità relativa K

(A,2) (B,1); (C,1); (D,0); (E,0)

k determina quando scendere statica per trovare il link statico da passare al chiamato

Tentativo di ridurre i costi: il display

L'accesso a variabili non locali, comporta la scansione della catena di link statici,

- costo proporzionale alla profondità.
- si può ridurre questo costo ad un singolo accesso usando il display:

Un unico array contenente il link ai RdA visibili al momento attuale

- i-esimo elemento dell'array: puntatore al RdA sottoprogramma
  - di livello di annidamento statico i,
    - programma principale: ann. statico 0
    - procedura definita nel programma principale: ann. stat. 1
    - ...
  - ultima istanza attivata (se, per ricorsione, ci sono più istanza)

Per accedere ad una variabile `x` con annidamento statico i , - l´i-esimo elemento nel display determina il RdA contenente `x`

Come si aggiorna il display

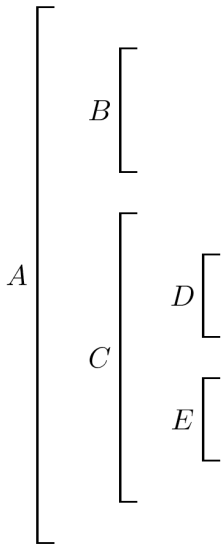
La procedura chiamata P, nel preambolo:

- sa la posizione nel display dove inserire il puntatore al suo RdA
- salva (nel suo RdA) il valore originario
- inserisce nel display il puntatore al suo RdA
- al termine della chiamata, P ripristina il valore originario

Algoritmo semplice, correttezza non banale:

- ogni procedura, all'ingresso aggiorno il display, in un unica posizione
  il display modificato corretto per l'ambiente della procedura
- alla sua uscita ripristina il valore originario
  il display torna ad essere corretto per il chiamante
- anche se chiamate una sequenza di chiamate di procedura
  - modifica il display in molte posizione,
  - al loro termine viene ripristinato il valore originario

Esempio



Es. Sequenze di chiamate A C D C E

Nella pratica

Nella statistica del codice rare lunghezze catena statica > 3,

display poco usato nelle implementazioni moderne.

Scope dinamico

Con scope dinamico l'associazione nome-oggetto denotabili dipende

- dal flusso del controllo a run-time
- dall'ordine con cui i sottoprogrammi sono chiamati.

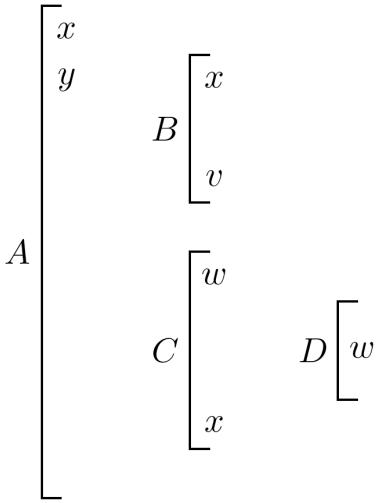
La regola generale è semplice:

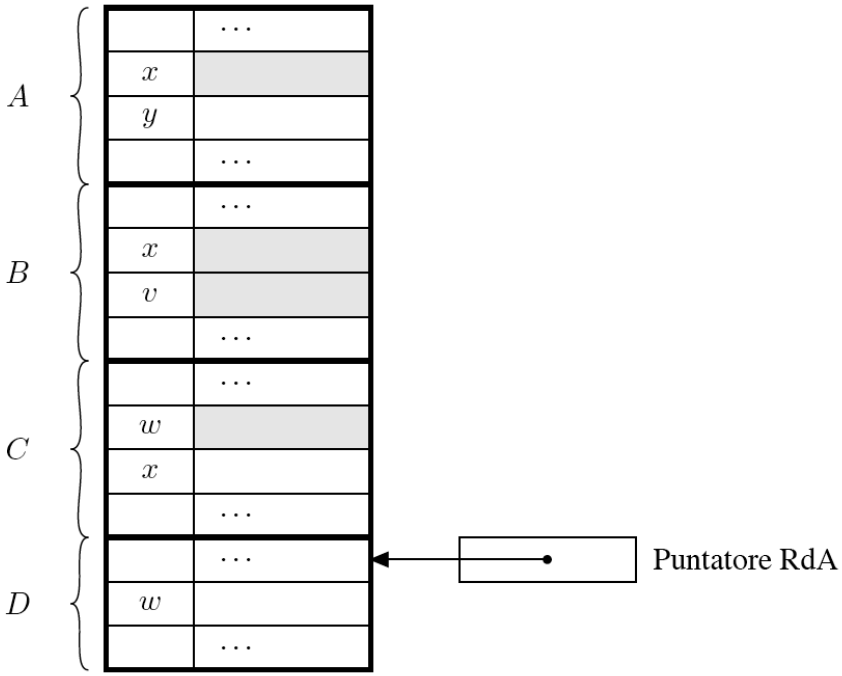
- il legame valido per il nome n
- è determinato dall'ultima dichiarazione del nome n eseguita.

Implementazione ovvia

- memorizzare i nomi negli RdA
  - a differenza dello scope statico dove non è necessario memorizzarli
- ricerca per nome scendendo nello stack di attivazione.

Esempio - Programma con chiamate A,B,C,D

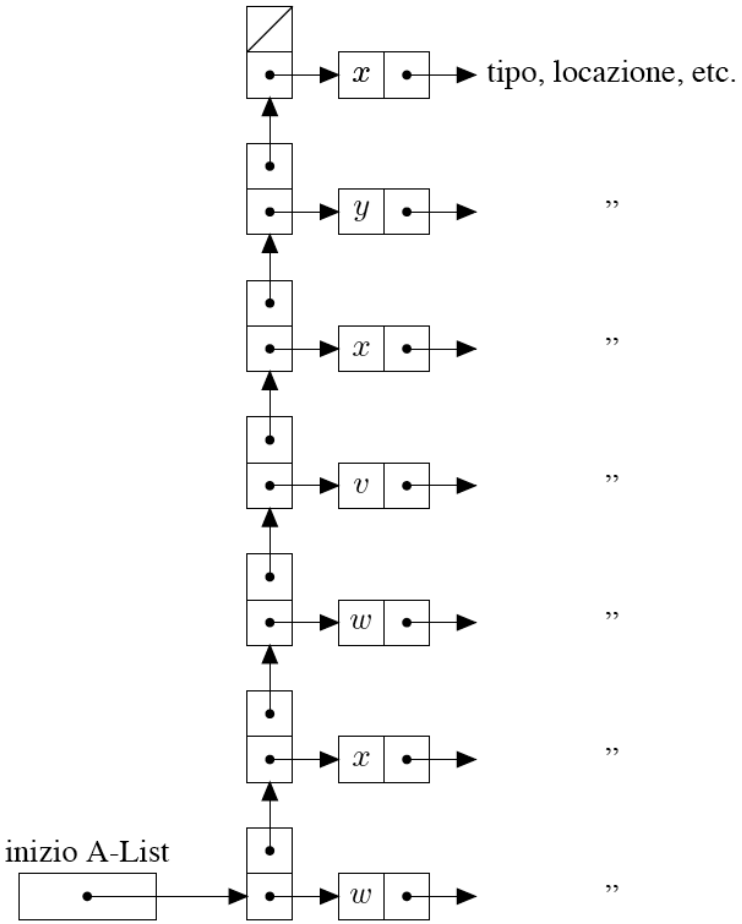




Variante: Association List: A-list

Le associazioni sono raggruppate in una struttura apposita,

- una lista di legami validi
- aggiornata con lo SdA



Costi delle A-list

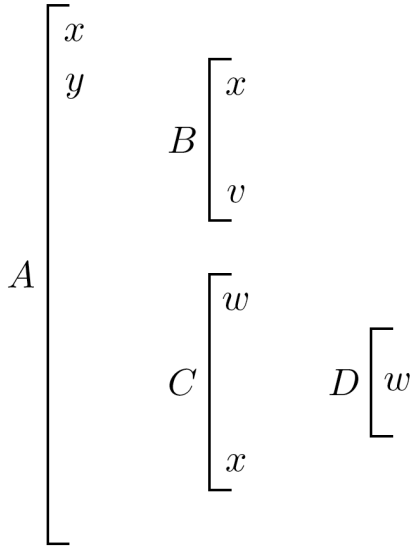
- Molto semplice da implementare
- Occupazione memoria:
  - nomi presenti esplicitamente
- Costo di gestione
  - ingresso/uscita da blocco
    - inserzione/rimozione di blocchi sulla pila
- Tempo di accesso
  - sempre lineare nella profondità della A-list

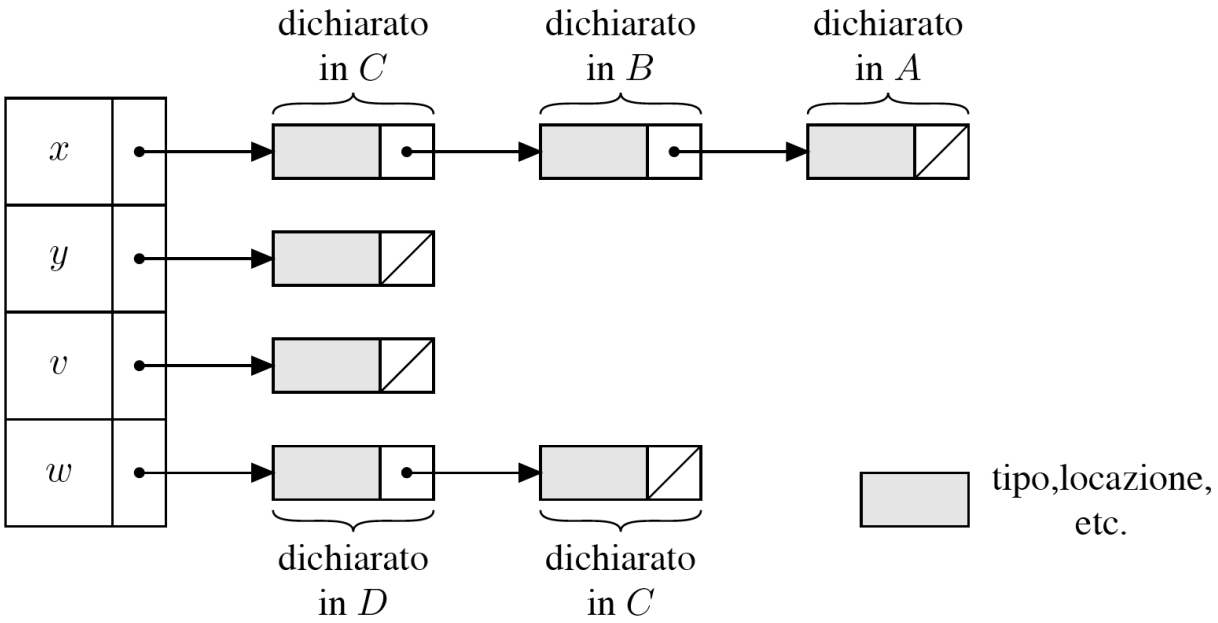
Possiamo ridurre il tempo d'accesso medio, aumentando il tempo di ingresso/uscita da blocco...

Tabella centrale dei riferimenti, CRT

- Evita le lunghe scansioni della A-list
- Una tabella mantiene tutti i nomi distinti del programma
  - se nomi noti staticamente accesso in tempo costante
  - altrimenti, accesso hash
- Ad ogni nome è associata la lista delle associazioni di quel nome:
  - la più recente è la prima
  - le altre, disattivate e per uso futuro, seguono
- accesso agli identificatori, in tempo costante

Esempio (con chiamate A,B,C,D)





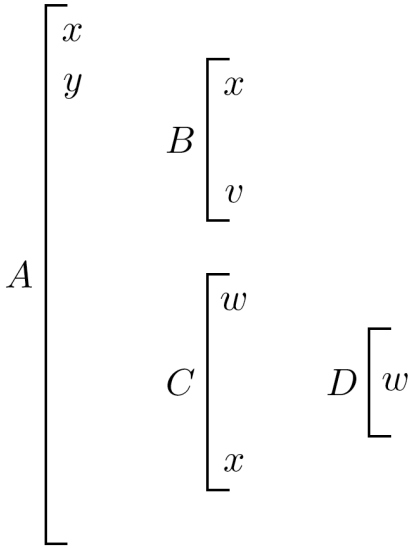
Pila nascosta

Seconda possibile implementazione

- Tabella attuale,
- Pila dei legami sospesi,

  una singola pila in cui inserisco i legami nascosti,
  da aggiornare in ingresso uscita dai blocchi.

Esempio (con chiamate A,B,C,D)



A

x	1	α_1
y	1	α_2
v	0	-
w	0	-

AB

x	1	β_1
y	1	α_2
v	1	β_2
w	0	-

x	α_1

ABC

x	1	γ_1
y	1	α_2
v	0	β_2
w	1	γ_2

x	α_1
x	β_1

ABCD

x	1	γ_1
y	1	α_2
v	0	β_2
w	1	δ_1

x	α_1
x	β_1
w	δ_1

Costi della CRT

- Gestione più complessa di A-list

Costo di gestione

- ingresso/uscita da blocco
  - manipolazione delle liste dei nomi dichiarati nel blocco

Tempo di accesso

- costante (due accessi indiretti)

Confronto con A-List

- si riduce il tempo d'accesso medio,
- si aumenta il tempo di ingresso/uscita dal blocco

Ottimizzazione:

- posso inserire in CRT solo i nomi locali usati esternamente alla procedura,
- ai nomi locali, si accede direttamente tramite RdA.

Esercizi

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().

```
void A()
 { void B()
    {  void C()
        {   }
    }
   void D()
    {  }
 }
```

Esercizi

Mostrare l'evoluzione dello stack di attivazione del seguente codice nei diversi meccanismi di scope (statico,dinamico), ed di implementazione (catena statica, display, A-List, CRT)

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

Esercizi

Mostrare l'evoluzione dello stack di attivazione del seguente codice nei diversi meccanismi di scope, ed di implementazione

```
int x = 1, y = x;
void A (){
    void B {
        int y = 2, z = 3;
        void C(){
            x++; y++, z++; }
        void D () {
            int x = 4, z = 5;
                C(); z++; }
        D(); C(); E()}
    void E (){
        x++; y++:}
B(); E();}
A();
```

Strutture di controllo

Espressioni, assegnazione, costrutti per il controllo di flusso, ricorsione

Strutturare il controllo

- Codice macchina: sequenza di istruzioni elementari, istruzioni di salto
- Linguaggi di programmazione: si vuole astrarre su controllo
- definizioni più:
  - strutturate
  - compatte
  - leggibili

Strutture per il controllo del flusso

- Espressioni
  - Notazioni
  - Meccanismi di valutazione
- Comandi
  - Assegnamento
- Sequenzializzazione di comandi
- Test, condizionali
- Comandi iterativi
- Ricorsione

che presenteremo ora

Altri meccanismi di controllo

- Chiamate di funzioni
- Gestione delle eccezioni
- Esecuzione concorrente
- Scelta non deterministica

presi in considerazione nel resto del corso.

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

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

Espressioni

Espressioni contenenti: identificatori, letterali, operatori (+, - . . .), funzioni valutate dalla macchina producono:

- un valore
- un possibile effetto collaterale
- possono divergere

Notazione

Principali differenze:

- Posizione dell'operatore: infissa, prefissa, postfissa
- Uso delle parentesi

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)`

Notazione

Parentesi:

- Scheme, Lisp: (Cambridge polish)
  parentesi necessarie per forzare la valutazione, non possono essere aggiunte arbitrariamente
- ML, Haskell, C,
  parentesi usate per definire un ordine di valutazione,

Zucchero sintattico:

scritture alternativa di un espressione (comando) per migliorare la leggibilità

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

Notazione polacca

Esistono notazioni che non necessitano parentesi

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

Ottenute tramite una visita anticipata, o differita, dell'albero sintattico

Le parentesi possono essere omesse solo se l'arità delle funzioni è prefissata

Esempi di arità variabile:

- Scheme: `(+ 1 2 3)`
- Erlang: funzioni diverse, con stesso nome, distinte per l'arità

Funzioni di arità arbitraria, parentesi indispensabili.

Notazione polacca

Poco leggibili e poco usate nei linguaggi di programmazione: Forth, calcolatrici tascabili

- polacca diretta, giustificazione: nella notazione a funzione argomenti `f(x, y)` possiamo omettere `( , )` se conosciamo l'arità di ogni funzione
- polacca inversa: descrive la valutazione di un'espressione con lo stack degli operandi
  presente in java bytecode, e in altri linguaggi intermedi: CLI
  `2 + 3 * 5` diventa:

```
2 3 5 * +
```

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

Sintassi delle espressioni: notazione infissa

I linguaggi di programmazione tendono a usare le notazioni della scrittura matematica:

- notazione infissa
- regole di precedenza tra gli operatori per risparmiare parentesi ma non sempre ovvio il risultato della valutazione:
  - `a + b * c ** d ** e / f` ??
  - `A < B and C < D` ??
    in Pascal Errore (se A,B,C,D non sono tutti booleani)

Regole di precedenza

Ogni linguaggio di programmazione fissa le sue regole di precedenze tra operatori

- di solito operatori aritmetici precedenza su quelli di confronto che hanno precedenza su quelli logici (non in Pascal)
- Numerose regole e 15 livelli di precedenze in C e suoi derivati (C++, Java, C#)
- 3 livelli di precedenza in Pascal
- APL, Smalltalk: tutti gli operatori hanno eguale precedenza: si devono usare le parentesi
- Haskell permette di definire nuove funzioni con notazione infissa, e specificarne precedenza e associatività `infixr 8 ^`

Tabella delle precedenze

Fortran	Pascal	C	Ada
		`++`, `--` (post-inc., dec.)	
`**`	`not`	`++`, `--` (pre-inc., dec.), `+`, `-` (unary), `&`, `*` (address, contents of), `!`, `~` (logical, bit-wise not)	`abs` (absolute value), `not`, `**`
`*`, `/`	`*`, `/`, `div`, `mod`, `and`	`*` (binary), `/`, `%` (modulo division)	`*`, `/`, `mod`, `rem`
`+`, `-` (unary and binary)	`+`, `-` (unary and binary), `or`	`+`, `-` (binary)	`+`, `-` (unary)
		`<<`, `>>` (left and right bit shift)	`+`, `-` (binary), `&` (concatenation)
`.eq.`, `.ne.`, `.lt.`, `.le.`, `.gt.`, `.ge.` (comparisons)	`<`, `<=`, `>`, `>=`, `=`, `<>`, `IN`	`<`, `<=`, `>`, `>=` (inequality tests)	`=`, `/=`, `<`, `<=`, `>`, `>=`
`.not.`		`==`, `!=` (equality tests)	
		`&` (bit-wise and)	
		`^` (bit-wise exclusive or)	
		`|` (bit-wise inclusive or)	
`.and.`		`&&` (logical and)	`and`, `or`, `xor` (logical operators)
`.or.`		`||` (logical or)	
`.eqv.`, `.neqv.` (logical comparisons)		`?:` (if...then...else)	
		`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=`, `|=` (assignment)	
		`,` (sequencing)	

Regole di associatività

Oltre al livello di precedenze, bisogna specificare in che ordine eseguire le operazioni di uno stesso livello

- Normalmente a sinistra `15 + 4 - 3` `(15 + 4) - 3`
- In alcuni casi a destra: `5 ** 2 ** 3` `5 ** (2 ** 3)`

Non sempre ovvie: in APL, tutto associa a destra, ad esempio,

```
15 + 4 - 3
```

è interpretato come

```
15 + (4 - 3)
```

Ricapitolando

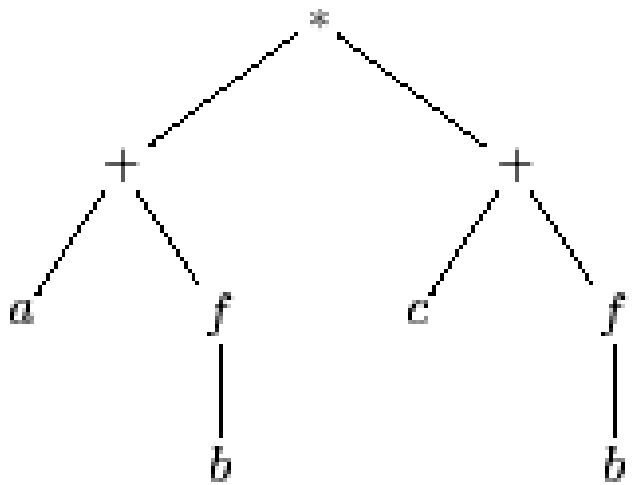
- regole di precedenza e associatività
- poco uniformi tra i vari linguaggi
- in alcuni casi piuttosto complesse

Nella pratica: se non si conoscono bene le regole, o si è insicuri, meglio inserire parentesi.

Rappresentazione ad albero

- La rappresentazione naturale di un espressione è un albero.
- Le espressioni vengono linearizzate per necessita di scrittura.

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



- la rappresentazione ad albero generata dall'analizzatore sintattico, per poter poi lavorare sull'espressione
- a partire dall'albero sintattico il compilatore produce il codice oggetto
  oppure l'interprete valuta l'espressione.

L'ordine di valutazione delle sottoespressioni

Le regole di precedenza, parentesi, o rappresentazione ad albero:

- definiscono precedenza e associatività,
- non definiscono un ordine temporale di valutazione delle sottoespressioni

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

Ordine importante per:

- effetti collaterali
- ottimizzazione

Effetti collaterali

La valutazione di un espressione restituisce un valore ma modifica lo stato del programma

Esempio tipico: la valutazione di un espressione

- porta a chiamate di funzioni
- le funzioni modificano la memoria

Nell'esempio:

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

il risultato della valutazione da sinistra a destra può essere diverso di quello da destra a sinistra

Ordine di valutazione

- In Java è specificato chiaramente l'ordine (da sinistra a destra)
- C non specifica l'ordine di valutazione, compilatori diversi si comportano in modo diverso.

```
int x=1;
printf("%d \n", (x++) * (++x));
x=1;
printf("%d \n", (++x) * (x++));
```

- L'ordine di valutazione ha influenza sul tempo di esecuzione, specie nei processori attuali (computazione parallela, accesso lento alla memoria)
  - C preferisce l'efficienza alla chiarezza, affidabilità
  - Java il contrario.

Ottimizzazione e ordine di valutazione.

Alcuni compilatori posso modificare l'ordine di valutazione per ragioni di efficienza;

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

può essere riarrangiato in

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

ed eseguito come

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

in alcuni casi, queste modifiche portano a modifiche nel risultato finale.

Evitare le ambiguità dovute all'ordine di valutazione

- In alcuni linguaggi non sono ammesse funzioni con effetti collaterali nelle espressioni (Haskell)
- altri linguaggi specificano l'ordine di valutazione (Java)
- in altri linguaggi, per evitare che il risultato dipenda da scelte del compilatore, forzando un ordine di valutazione,
  posso spezzare l'espressione

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

riscritta come

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

Effetti collaterali

Svantaggi: senza effetti collaterali la valutazione delle espressioni diventa:

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

Gli effetti collaterali possono essere utili:

- gestire strutture dati di grandi dimensioni,
  funzioni che operano su matrici
- definire funzione che generano numeri casuali `rand()`

Linguaggi funzionali puri

In linguaggi funzionali puri, la computazioni si riduce a:

- la solo valutazione di espressioni
- senza effetti collaterali

Aritmetica finita

- Numeri interi: limitati
- Numeri floating point: valori limitati e precisione limitata

Conseguenze: errori di overflow, errori di arrontondamento ma anche

le usuali identità matematiche non valgono per l'aritmetica dei calcolatori

```
a + (b + c) != (a + b) + c
```

- interi: la prima espressione genera errore di overflow la seconda no
  - `a = -2; b = maxint; c = 2;`
- floating point: l'errore nelle due valutazioni è differente
  - `a = 10**15; b = -10**15; c = 10**(-15);`

Valutazione eager - lazy. Operandi non definiti

Non sempre tutte le sottoespressioni sono valutate,

esempio tipico, espressioni `if then else`.

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

presuppone una valutazione lazy: si valutano solo gli operandi strettamente necessari.

Valutazione corto circuito

Alcuni operatori booleani (and, or) ammettono una la valutazione lazy,

- detta corto-circuito:
- se la valutazione del primo argomento è sufficiente a determinare il risultato non valuto il secondo
- ordine di valutazione fondamentale per determinare il risultato, di solito da sinistra a destra

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

se `a` uguale a `0`

- con valutazione corto circuito restituisce `true`
- valutazione eager genera errore
- anche una valutazione corto circuito da destra a sinistra genera errore

Valutazione corto-circuito

Restituisce immediatamente il risultato

- 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, hanno due 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.

Valutazione corto-circuito

Stesso codice (ricerca valore 3 in una lista)

si comporta in maniera diversa a seconda del linguaggio

- C, valutazione corto circuito: corretto

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

- Pascal, valutazione eager: genera errore

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

`p -> next` equivalente a `*p.next`

Comandi

Parti del codice la cui valutazione tipicamente:

- non restituisce un valore
- ha un effetto collaterale (modifica dello stato)

I comandi

- sono tipici del paradigma imperativo
- non sono presenti (o pochissimo usati) nei linguaggi funzionali e logici
- in alcuni casi restituiscono un valore (es. = in C)

Assegnamento

Comando base dei linguaggi imperativi

Inserisce in una locazione, cella, un valore ottenuto valutando un espressione.

```
X = 2
Y = X + 1
```

Notare il diverso ruolo svolto da `X` ne i due assegnamenti:

- nel primo, X denota una locazione, è un l-value
- nel secondo, X denota il contenuto della locazione precedente, è un r-value

Assegnamento

In generale

`exp1 := exp2`

- valuto `exp1` per ottenere un l_value (locazione)
- valuto `exp2` per ottenere un r_value, valore memorizzabile
- inserisco il valore nella locazione

l-value può essere definito da un espressione complessa

- esempio (in C)
  `(f(a)+3)->b[c] = 2`
  - f(a) puntatore ad un elemento in un array di puntatori a strutture `A`
  - la struttura `A` ha un campo `b` che un array
  - inserisco 2 nel campo c-esimo dell'array

Diversi significati del termine

La parte sinistra di un assegnazione è tipicamente una variabile.

Tralasciando l'uso matematico del termine "variabile", in informatica, a seconda dei contesti, esistono più significati:

- linguaggi imperativi: identificatore a cui è associata una locazione, dove a troviamo il valore modificabile.
- linguaggi funzionali (Lisp, ML, Haskell, Smalltalk): un identificatore a cui è associato un valore, non modificabile.

  coincidente con la nozione di costante per linguaggi imperativi.
- Linguaggi logici: una variabile rappresenta un valore indefinito, la computazione cerca le corrette istanziazioni delle variabili, quelle che rendono vero un certo predicato

Modello a valore

Diversi modi per:

- implementare le variabili
- implementare l'assegnamento
- definire cosa denotano le variabili

Nel modello a valore:

- alle variabili l'ambiente (il compilatore) associa una locazione di memoria
  - il valore contenuto nella locazione, denota il valore associato,
  - ma alcuni tipi di dato, il valore associato può essere a su volta una locazione

    es. tipi array in C

```
X[Y] = 3
```

L'assegnazione modifica il valere associato

Modello a riferimento:

- l'ambiente associa ad una variabile una locazione di memoria
- nella locazione troviamo un riferimento (una seconda locazione)
- contenente il valore

Per accedere al valore:

- devo dereferenziare la variabile
- dereferenzazione implicita

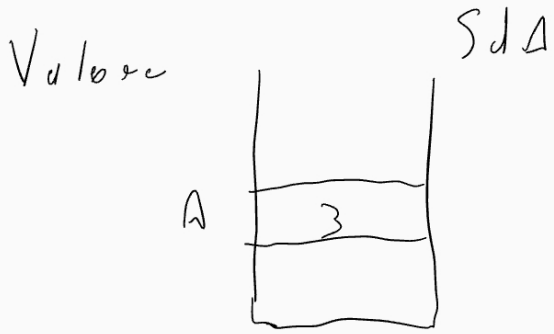
L'assegnazione modifica:

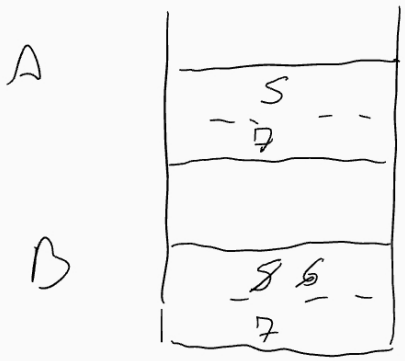
- il riferimento
- non il contenuto dello store

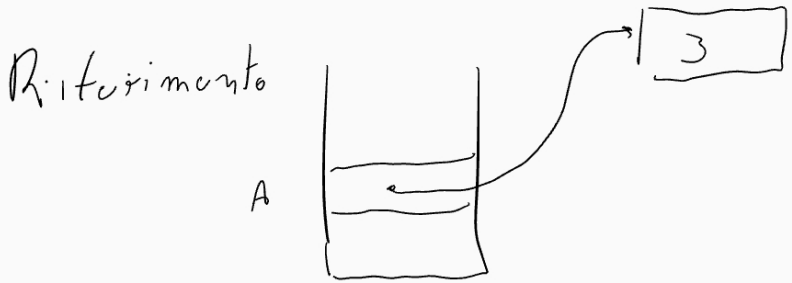
Dopo l'assegnamento `X = Y`, `X` e `Y` fanno riferimento alla stessa locazione di memoria, contenente un valore condiviso

- ogni variabile è, in certo senso, un puntatore
- usata con una sintassi diversa dai puntatori

Differenze dal punto di vista implementativo





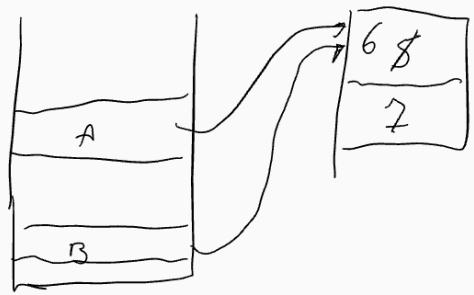


A = 3

A = (5, 7)

B = A

B[0] = 6



In alcuni linguaggi i due modelli si mischiano

A seconda del tipo della variabile

Java:

- tipi primitivi (interi, booleani ecc.) :
  - modello a variabile,
  - assegnamento copia un valore nella memoria
- tipi riferimento (tipi classe, array):
  - modello a riferimento
  - assegnamento crea un condivisione dell'oggetto.

Python:

Due categorie di tipi:

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

Assegnamento:

- immutabili: viene creata una nuova istanza dell'oggetto, non si modifica la memoria, ma analogo effetto.
- mutabili: viene condivisa, eventualmente modificata, l'istanza esistente

Esempio in Python

```
tuple1 = (1,2,3) # tuples are immutable
list1 = [1,2,3] # lists are mutable

tuple2 = tuple1
list2 = list1

tuple2 += (9,9,9)
list2 += [9,9,9]

print 'tuple1 = ', tuple1 # outputs (1, 2, 3)
print 'tuple2 = ', tuple2 # outputs (1, 2, 3, 9, 9, 9)
print 'list1 = ', list1 # outputs [1, 2, 3, 9, 9, 9]
print 'list2 = ', list2 # outputs [1, 2, 3, 9, 9, 9]
```

Vantaggi - svantaggi modello a riferimento

Vantaggi:

- non duplico strutture dati complesse
- tutte le variabili sono puntatori
  - utile nelle funzioni polimorfe
    stessa funzione agisce su una varietà di tipi di dato
    esempio: funzione che ordina un vettore

Svantaggi:

- si creano aliasing che oscurano il comportamento del programma
- accesso ai dati indiretto

Linguaggi funzionali (non puri) (ML)

Distinguo locazioni da contenuto:

- concettualmente più chiaro ma più prolisso
- accedo al contenuto esplicitamente

```
val x = ref 2      (* x denota una locazione contenente 2*)
val x2 = x         (* x e x2 denotano la stessa locazione  *)
val x3 = !x                (* x3 denota 2 *)
val _ = x := (!x) + 7      (* il contenuto di x, x2 è ora 9, *)
                           (* x3 denota sempre 2 *)
```

Ambiente e memoria

Nei linguaggi imperativi distinguiamo tra:

- Ambiente: Nomi –> Valori Denotabili
  - definito, modificato dalle dichiarazioni
- Memoria: Locazioni –> Valori Memorizzabili
  - modificato dalle istruzioni di assegnamento

Distinguiamo tra tre classi di valori:

- Valori Denotabili (quelli a cui si può associare un nome)
- Valori Memorizzabili (si possono inserire nello store esplicitamente con assegnamento)
- Valori Esprimibili (risultato della valutazione di una espressione)

Valori denotabili, demorizzabili, esprimibili

Le tre classi si sovrappongono ma non coincidono.

- procedure: denotabili, a volte esprimibili, quasi mai memorizzabili,
- locazioni: denotabili, esprimibili, memorizzabili con l'uso esplicito dei puntatori.

Linguaggi imperativi, i valori denotabili includono le locazione:

- variabili nomi che denotano locazioni.

Linguaggi funzionali puri:

- non esistono valori memorizzabili
- le locazioni non sono dentabili o esprimibili

Linguaggi funzionali:

- le funzioni sono valori esprimibili
- Java, C#, Python, Ruby permettono la programmazione funzionale

Operazioni di assegnamento

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

Realizzate in maniera standard pongono i seguenti problemi:

- scarsa leggibilità,
- efficienza: doppio computazione dell'indice `index(i)`, doppia accesso alla locazione,
- side-effect: se `index(i)` causa un effetto collaterale, questo viene ripetuto.

Si definiscono degli operatori di assegnamento, più sintetici

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

Operazioni di assegnamento

In C, Java . . . una pletora operatori di assegnamento, incremento/decremento.

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

- somma, sottrazione, moltiplicazione, divisione, resto, bit-wise and, bit-wise or

Incremento, decremento di una unità

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

Nel caso la variabile incrementata sia un puntatore C ad un array di oggetti,

- l'incremento viene moltiplicato per la dimensione degli oggetti puntati.

Espressioni e comandi

Sintatticamente si distingue tra comandi e espressioni

- Comandi: è importante l'effetto collaterale
- Espressioni: è importante il valore restituito.

In alcuni linguaggi la distinzione comando, espressione risulta sfumata:

- i due aspetti, effetto collaterale e risultato coesistono,
- dove è previsto un comando posso inserire un'espressione e viceversa.

C, Java, C#:

Comandi separati da espressioni:

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

ma

- espressioni possono comparire dove ci si aspetta un comando
- assegnamento (=) permesso nelle espressioni
  - l'assegnamento restituisce il valore assegnato, posso scrivere:
    - `a = b = 5` interpretato come `a = (b = 5)`
    - `if (a == b) { ... }` naturalmente, ma anche
    - `if (a = b) { ... }` che può generare errore di tipo

```
(a==b) ? x=1 else x=0;        \\ legito
(a==b) ? {x=1} else {x=0};    \\ genera errore
```

un singolo comando può essere visto come un espressione, un blocco no.

pre e post incremento

Il comando di incremento. ++ può essere visto come un espressione, con effetti collaterali,

- distinguo tra due versioni delle espressione di incremento, pre e post incremento
  - `++x`: pre-incremento, esegue `x = x+1` restituisce il valore incrementato
  - `x++`: post-incremento, esegue `x = x+1` restituisce il valore originario

  eseguono la stessa assegnazione ma restituiscono un diverso valore
- similmente esistono
  - `(x--)` `(--x)`

Algol68, tutto è un espressione.

- In Algol68: expression oriented
  - non c'è nozione separata di comando
  - ogni procedura restituisce un valore

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

Scelta opposta - Pascal:

- comandi separati da espressioni
- un comando non può comparire dove è richiesta un'espressione e viceversa

Esercizi

Mostrare l'evoluzione dello store nei seguenti comandi:

- valutazione delle espressioni da sinistra a destra,
- nell'assegnazione, il r-value valutato prima del l-value,
- indice base del vettore: 0.

```
int V[5] = { 1, 2, 3, 4, 5};
int i = 3;
V[--i] += i;
V[i--] = i + i++;
(V[i++])++;
```

```
i = 0;
i = V[i++]  = V[i++]  = (V[i++])++;
```

Comandi per il controllo sequenza

Comandi per il controllo sequenza esplicito

- ;
- blocchi
- goto

Comandi condizionali

- if
- case

Comandi iterativi

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

Comando sequenziale

Composizione sequenziale

```
C1 ; C2 ;
```

- è il costrutto di base dei linguaggi imperativi
- sintassi due possibile scelte:
  - ; separatore di comandi, non serve inserirlo nell'ultimo comando
  - ; terminatore di comandi, devo inserirlo anche nell'ultimo comando
    C e derivati usano questa sintassi

Algol 68, C: quando un comando composto è visto come espressione il valore è quello dell'ultimo comando

Blocco

Sintassi

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

Trasformo una sequenza di comandi in un singolo comando, raggruppo una sequenza di comandi

Posso usarlo per introdurre variabili locali

GOTO - istruzioni di salto

```
if a < b  goto 10
...
10: ...
```

- costrutto base in assembly
- permette una notevole flessibilità
- ma rende programmi poco leggibili, e nasconde gli errori
- interazione complessa con chiamate di funzioni e stack di attivazione

Accesso dibattito negli anni 60/70 sulla utilità del goto[1]

Alla fine considerato dannoso, contrario ai principi della programmazione strutturata

[1] E. Dijkstra. Go To statements considered Harmul. Communications of the ACM, 11(3):147-148. 1968.

sostituibilità del GOTO

Teoria

- teorema di Boehm-Jacopini
  - GOTO sostituibile da costrutti cicli while - test,

Pratica:

- la rimozione del GOTO non porta a una grossa perdita di flessibilità - espressività
- istruzioni di salto giustificabili e utile solo in particolari contesti, con costrutti appositi:
  - uscita alterativa da un loop: `break`
  - ritorno da sottoprogramma: `return`
  - gestione eccezioni: `raise exeption`

Nei linguaggi che prediligono la sicurezza, chiarezza (Java) il Goto non e' presente.

Programmazione strutturata

Metologia introdotta negli anni 70, per gestire la complessità del software

- Progettazione gerarchica, top-down
- Modularizzare il codice
- Uso di nomi significativi
- Uso estensivo dei commenti
- Tipi di dati strutturati
- Uso dei costrutti strutturati per il controllo
  - ogni costrutto, pezzo di codice, un unico punto di ingresso e di uscita
  - le singole parti della procedura modularizzate,
  - diagrammi di flusso non necessari.

Comando condizionale

```
if (B) {C_1} ;
if (B) {C_1} else {C_2} ;
```

- Introdotto in Algol 60
- Varie regole per evitare ambiguità in presenza di if annidati:

```
if b1 then if b2 then c1 else c2
```

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

```
- Pascal, C: else associa con il then non chiuso più vicino
- Algol 68, Fortran 77: parole chiave `endif` o `fi` marcano l
```

Rami multipli espliciti con comando `else if`

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

Espressione condizionale in Scheme.

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

- Valuta test-expr.
- Se il risultato un valore diverso da #f,
  - allora viene valutata `then-expr`
  - altrimenti si valuta `else-expr`

Esempi:

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

Comando condizionale in Scheme.

In alternativa:

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

Argomenti (in numero arbitrario

- coppie `[ guardia_booleana  valore_restituito ]`

Case

Estensione del `if then else` a tipi non booleani

```
case exp of                    {* exp: espressione a valori discret
|   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
```

- Molte versioni nei vari linguaggi
- Possibilità di definire range `case 0 ... 9: C2`
  - non presente in: Pascal, C (vecchie versioni)
  - presente C, Visual Basic ammettono range:
- ramo di default : C, Modula, Ada, Fortran,
  - senza default, e con nessuna opzione valida: non si esegue nulla.

Pattern matching

- ML, Haskell: possibilità usare pattern complessi all'interno di case

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

- casi non mutuamente esclusivi, si sceglie il primo,
- casi non esaustivi, si genera errore
- istanziazione di variabili, meccanismo piuttosto sofisticato

Sintassi di C, C++ e Java

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

Ogni sotto-comando termina `break`,

- devo uscire esplicitamente dal caso con `brake`
- altrimenti si continua col comando successivo,
  posso evitare di scrivere un comando due volte
- facile causa di errori
  in C# devo esplicitare l'azione finale: `break, continue`

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

Estensioni con range

```
switch (arr[i])
  {
  case 1 ... 6:
      printf("%d in range 1 to 6\n", arr[i]);
      break;
  case 19 ... 20:
      printf("%d in range 19 to 20\n", arr[i]);
      break;
  default:
      printf("%d not in range\n", arr[i]);
      break;
```

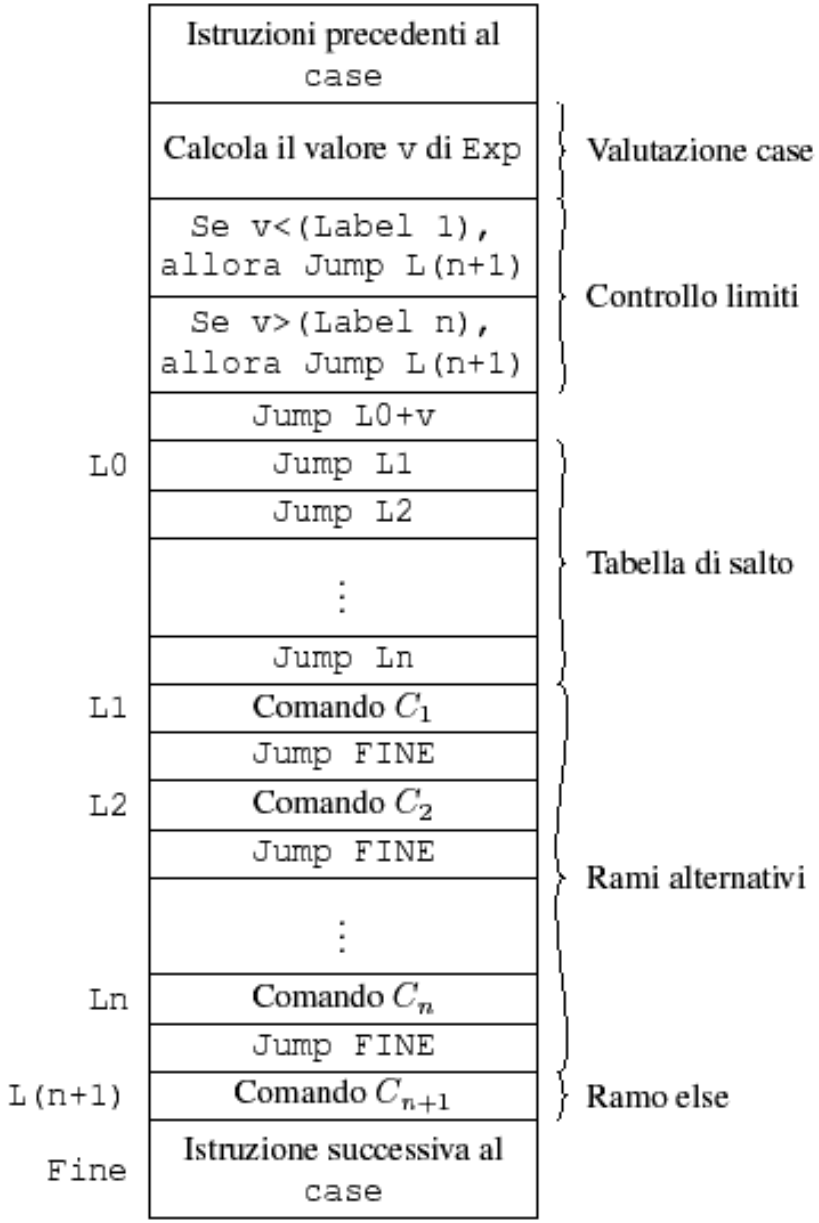
Compilazione del case

Più efficiente di if multiplo se compilato in modo astuto . . .

```
case  exp of
    |    label_1 : C_1
    |    label_2 : C_2
    ...
    |    label_n : C_n
    else C_n+1
```

- con il valore di `exp` accedo a
- una tabella di istruzioni di salto
- che porta al codice macchina del ramo corrispondente al valore

struttura del codice generato:



Versione per range ampi

- Lo schema precedente funziona bene,
  - tempo di esecuzione costante e non lineare sul numero di possibilità
  - occupazione di memoria limitata
  - se i range di valori sono limitati
- Con range ampi, troppa occupazione di memoria
  devo ripetere la stessa istruzione di salto per ogni valore nel range
- È possibile ridurre l'occupazione di memoria con
  - ricerca binaria
  - tabella hash
- codice assembly più complesso, tempo di esecuzione: logaritmico o costante

Iterazione

- Iterazione e ricorsione sono i due meccanismi che permettono di ottenere tutte le funzioni computabili
  - formalismi di calcolo Turing completi.

  Senza iterazione: nessuna istruzione ripetuta, tutto termina in un numero limitato di passi.
- Iterazione
  - indeterminata: cicli controllati logicamente

    (while, repeat, . . .)
  - determinata cicli controllati numericamente

    (do, for, foreach . . .) con numero di ripetizioni del ciclo determinate al momento dell'inizio del ciclo

Iterazione indeterminata

```
while condizione do  comando
```

Sintassi più usata Java, ...

```
while (counter > 1)
{ factorial *= counter--;
}
```

in altri linguaggi: Pascal,

```
while Counter > 0 do
begin
  Factorial := Factorial * Counter;
  Counter := Counter - 1
end;
```

- Introdotta in Algol-W,

Versione post-test, ripetuto almeno una volta

- tipicamente C, C++, Java;

```
do {
     factorial *= counter--; // Multiply, then decrement.
   } while (counter > 0);
```

- Ruby

```
begin
    factorial *= counter
    counter -= 1
end while counter > 1
```

- Pascal

```
repeat
    Factorial := Factorial * Counter;
    Counter -:=  1
until Counter = 0
```

Equivalenza

Facile sostituire un tipo di ciclo con l'altro

```
do {
    do_work();
} while (condition);
```

equivalente a:

```
do_work();
while (condition) {
    do_work();
}
```

A seconda dei casi, una versione risulta più sintetica dell'altra

Iterazione indeterminata

- Indeterminata perché il numero di iterazioni non è noto a priori
- l'iterazione indeterminata permette di ottenere tutte le funzioni calcolabili,
- facile da tradurre in codice assembly

Iterazione determinata

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

- al momento dell'inizio dell'esecuzione del ciclo, è determinato il numero di ripetizioni del ciclo
  - all'interno del loop, non si possono modificare: `indice`,
  - `inizio, fine, passo` valutati e salvati a inizio esecuzione
- il potere espressivo è minore rispetto all'iterazione indeterminata: non si possono esprimere computazioni che non terminano
- da preferire perché:
  - garantisce la terminazione,
  - ha una scrittura più leggibile e compatta
- in C, e suoi derivati, il for non è un costrutto di iterazione determinata, posso modificare l'indice

Semantica del For

```
FOR indice := inizio TO fine BY passo DO
        ....
END
```

nell'ipotesi di passo positivo:

1. valuta le espressioni inizio e fine e salva i valori ottenuti
2. inizializza `indice` con il valore di `inizio`;
3. se `indice > fine` termina l'esecuzione del for
   altrimenti:
   - si esegue corpo
   - si incrementa `indice` del valore di `passo`;
   - si torna a (3).

Diverse realizzazioni

```
FOR indice := inizio TO fine BY passo DO
        ....
END
```

I vari linguaggi differiscono nei seguenti aspetti:

- Possibilità di modificare indice, valore finale, passo nel loop (se si, non si tratta di vera iterazione determinata)
- 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

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

Dove `statement` è spesso un blocco

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

Altri linguaggi

Python:

```
for counter in range(1, 6):   # range(1, 6) gives values from 1
  # statements
```

Ruby:

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

Foreach

Ripeto il ciclo su tutti gli elementi di un oggetto enumerabile: array, lista

- Presente in vari linguaggi sotto diverse forme
- Limitato potere espressivo, ma utile per
  - chiarezza
  - compattezza
  - prevedibilità

Java dalla versione 5

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

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

Foreach

Vengono separati,

- algoritmo di scansione della struttura:
  - implicito nel `foreach`,
  - generato automaticamente nel compilatore
- operazioni da svolgere sul singolo elemento
  - definito esplicitamente nel codice

Può essere svolto su un tipo di dato strutturato che metta a disposizione funzioni implicite per determinare

- primo elemento
- passo ad elemento successivo
- test di terminazioni

Esempi

- liste, array, insiemi
- alberi

Foreach altri esempi

Python:

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

- Ciclo for per Python: un caso particolare di questo meccanismo.

Ruby

```
pets = ['cat', 'dog', 'fish']
pets.each do |f|
  f.print
end

for f in pets
    f.print
end
```

Foreach altri esempi

JavaScript

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

Ricorsione

- Modo alternativo all'iterazione per ottenere il potere espressivo delle MdT
- scelta obbligata nei linguaggi puramente funzionali
- Intuizione: una funzione (procedura) è ricorsiva se definita in termini si se stessa.
- Riflette la natura induttiva di alcune funzioni.

```
fattoriale (0) = 1.
fattoriale (n) = n*fattoriale(n-1)
```

diventa

```
int fatt (int n){
  if (n <= 1)
     return 1;
  else
     return n * fatt ( n-1 );
```

Definizioni induttive (intermezzo)

Numeri naturali 0, 1, 2, 3, . . . Minimo insieme X che soddisfa i due assiomi seguenti (Peano):

- `0` è in X;
- Se `n` è in X allora `succ(n)` è X;

_Principio di induzione. Una proprietà P(n) è vera su tutti i numeri naturali se

- P(0) è vera;
- Per ogni n, se P(n) è vera allora è vera anche P(n + 1).

Definizioni induttive (ricorsive). Se g: (Nat x A) -> A totale allora esiste una unica funzione totale f : Nat -> A tale che

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

Fattoriale segue lo schema di sopra.

Definizione induttive

- Garanzia di buona definizione (definisco univocamente una funzione totale)
- Schema piuttosto rigido:
  - la definizione induttiva della divisione `div(n, m)` è non ovvia
  - si può generalizzare: well founded induction, per avere
    - schema più flessibile
    - garanzia di buona definizione

Ricorsione e definizioni induttive

- Funzione ricorsiva F analoga alla definizione induttiva di F:

  il valore di F su n è definito in termini dei valori di F su m<n
- Tuttavia nei programmi sono possibili definizioni non “corrette”:
- la seguente scrittura non definisce alcuna funzione

```
foo(0) = 1
foo(n) = foo(n+1) - 1
```

- invece i seguenti programmi sono possibili

```
int foo (int n){
   if (n == 0)
       return 1;
   else
       return foo(n+1) - 1
}
```

Ricorsione e iterazione

La ricorsione è possibile in ogni linguaggio che permetta

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

Ogni programma ricorsivo può essere tradotto in uno equivalente iterativo

e viceversa,

Ricorsione e iterazione

Confronto:

- ricorsione più naturale su strutture dati ricorsive (alberi), quando la natura del problema è ricorsiva
  iterazione più efficiente su matrici e vettori.
- ricorsione scelta obbligata nei linguaggi funzionali
  iterazione scelta preferita nei linguaggi imperativi

In caso di implementazioni naif ricorsione molto meno efficiente di iterazione tuttavia

- optimizing compiler può produrre codice efficiente
- tail-recursion

Ricorsione in coda (tail recursion)

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 se stessa

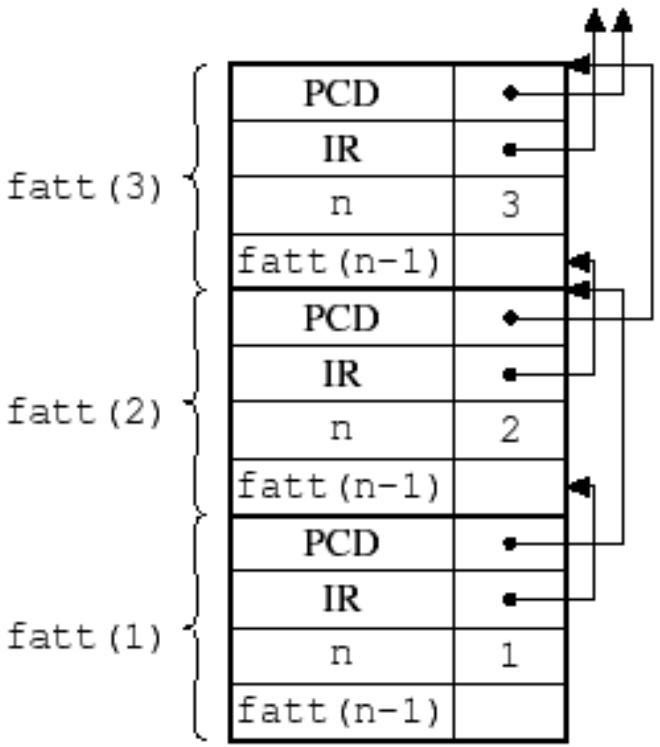
```
function tail_rec (n: integer, m): integer
begin ... ; return tail_rec(n-1, m1) end

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

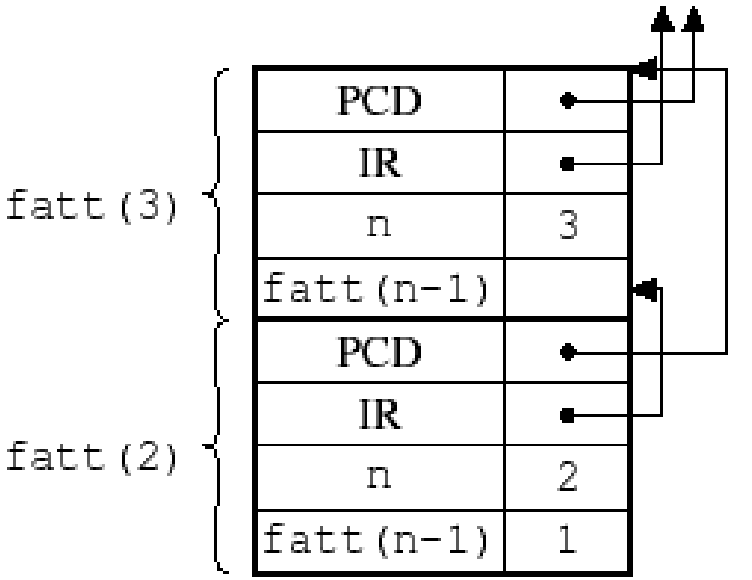
- Non serve allocazione dinamica della memoria con pila: basta un unico RdA,
  - dopo la chiamata ricorsiva, il chiamante non deve fare nulla, attende il risultato, e lo passa al suo rispettivo chiamante
  - record di attivazione del chiamante inutile, spazio riutilizzato dal chiamato

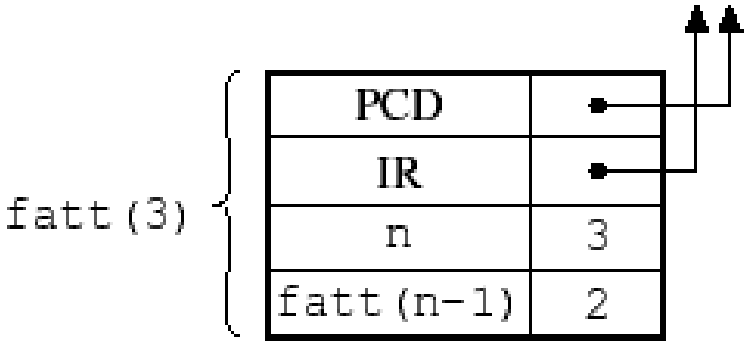
Più efficiente, esempio: il caso del fattoriale

```
int fatt (int n){
  if (n <= 1)
     return 1;
  else
     return n * fatt ( n-1 ); }
```



Esempio: il caso del fattoriale





Una versione tail-recursive del fattoriale

```
int fattrc (int n, int res){
  if (n <= 1)
     return res;
  else
     return fattrc ( n - 1, n * res); }
int fatt (int n){
  fattrc (n, i)}
```

- Abbiamo aggiunto un parametro che rappresenta il valore da passare al "il resto della computazione"
- `fattrc (n, res)` calcola il valore `res * n!`

Basta un unico RdA

- dopo ogni chiamata il RdA della funzione chiamante può essere eliminato
- il suo spazio usato per il RdA della chiamata

Un altro esempio: numeri di Fibonacci

Definizione:

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

in Scheme diventa

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

Complessità in tempo e spazio esponenziale in n

- ad ogni chiamata due nuove chiamate
- più precisamente il numero della chiamate ha una crescita `alla` Fibonacci.

Una versione efficiente per Fibonacci

`fibHelper(n, a, b)` una funzione che nell'ipotesi:

- a = Fib(i-1)
- b = Fib(i)

```
fibHelper(n, a, b)  =  Fib(i + n)
```

in Scheme:

```
(define (fibHelper n a b)
  (if (= n 0)
      b
      (fibHelper (- n 1) b (+ a b))))
(define (fib n)
  (if (= n 0)
      1
      (fibHelper (- n 1) 1 1)))
```

Analisi

Invariante:

- se `a` e `b` sono l'(m-1)-esimo e l'm-esimo elementi nella serie di Fibonacci,
- allora `(fibHelper n a b)` è (m+n)-esimo elemento nella serie

Complessità:

- in tempo, lineare in `n`
- in spazio, costante (un solo RdA)

Chiave di lettura della ricorsione di coda

- simulo l'esecuzione di un ciclo (`while`) dentro un linguaggio funzionale:
  - per ogni funzione `f` che in un linguaggio imperativo avrei implementato tramite un ciclo
  - definisco una funzione `f-helper` avente parametri extra
  - questi parametri extra svolgono il ruolo delle variabili modificabili nel ciclo
  - `f-helper` chiama se stessa aggiornando i parametri extra, come avviene in un ciclo
  - `f` chiama `f-helper` inizializzando i parametri extra (come la funzione imperativa)
- Simulo uno `stato` in maniera locale e controllata, senza introdurre uno stato globale

Uso della ricorsione di coda

- Ottimizzazione: solo le funzioni ricorsive più critiche (per la velocità d'esecuzione globale del programma) vengono riscritte usando la ricorsione di coda

Continuation passing style

- Stile di programmazione funzionale
  - genera programmi tail-recursive
  - può migliorare la complessità computazionale
  - introduce un controllo esplicito sull'ordine di esecuzione
- Trasformo ogni funzione, aggiungendo ai suoi argomenti, un argomento `k`, continuazione
  - `k` rappresenta il resto del programma,
    - prende il valore, generato dalla funzione, restituisce un risultato del programma
  - la computazione viene ridotta ad eseguire operazioni semplici e passare il risultato ad una continuazione.

Continuation passing style

```
(define (pyth x y)
 (sqrt (+ (* x x) (* y y))))

(define (*& x y k)
  (k (* x y)))

(define (pyth& x y k)
 (*& x x (lambda (x2)
           (*& y y (lambda (y2)
                     (+& x2 y2 (lambda (x2py2)
                                 (sqrt& x2py2 k))))))))
```

Fattoriale:

```
(define (fattoriale n)
 (if (= n 0)
     1
     (* n (factorial (- n 1)))))
```

diventa:

```
(define (factorial& n k)
 (=& n 0 (lambda (b)
           (if b
               (k 1)
               (-& n 1 (lambda (nm1)
                         (factorial& nm1 (lambda (f)
                                           (*& n f k)))))))))
```

Astrarre sul controllo

Procedure, passaggio dei parametri, eccezioni

Argomenti

- Procedure e funzioni: astrazione sul controllo
- Modalità di passaggio dei parametri
- Funzioni di ordine superiore
  - funzioni come parametro
  - funzioni come risultato
- Gestori delle eccezioni

Astrazione

Meccanismo per gestire la complessità:

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

Permette di:

- evidenziare tratti comuni in strutture diverse (matematica)
- gestire la complessità:
  - spezzare un sistema complesso in sottosistemi,
  - descritti astrattamente, senza entrare nel dettaglio

In informatica:

- astrazione sul controllo (istruzioni)
- astrazione sui dati

Astrazione sul controllo

Definizione di procedura-funzione principale meccanismo di astrazione

```
float log (float x) {
    double z;
    /* CORPO DELLA FUNZIONE */
    return expr;
}
```

Possiamo:

- usare `log`
  - conoscendo le sue specifiche
  - senza conoscerne nei dettagli l'implementazione
- specificare `log`
  - senza conoscere l'implementazione
- implementare `log` (scriverne il codice)
  - rispettando le specifiche
  - senza conoscere il contesto in cui verrà usato

Parametri

Terminologia:

- dichiarazione/definizione

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

`n` paramento formale

- uso/chiamata

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

`y+3` parametro attuale

Distinguiamo tra:

- funzioni: restituiscono un valore
- procedure: non restituiscono nulla,
- in C e derivati: procedure viste come funzioni
  che restituisco un valore di tipo `void` (avente un unico elemento)
  distinzione sfumata

Come una funzione (chiamata) comunica col chiamante

- Valore restituito

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

- Passaggio di parametri
  varie modalità di passaggio
- Modifiche ambiente non locale
  - in questo caso il meccanismo di astrazione meno efficace
  - chi usa la funzione deve conoscere anche quali modifiche sono svolte
  - nei linguaggi funzionali questa possibilità è non presente, più astratti

Modalità di passaggio dei parametri

Come avviene l'interazione chiamante – chiamato attraverso i parametri

Distinguo tra:

- parametri d'ingresso: main -> proc
- parametri d'uscita: main <- proc
- parametri d'ingresso – uscita : main <-> proc

Varie modalità di passaggio con diversa:

- semantica
- implementazione
- costo

Modalità di passaggio dei parametri

Due modi principali:

- per valore (call by value):
  - il parametro formale è una variabile locale
  - il valore del parametro attuale è assegnato al formale
  - parametro in ingresso: main -> proc
    le modifiche al formale non passano all'attuale
  - attuale: r-value, una qualsiasi espressione
- per riferimento (call by reference)
  - è passato un riferimento (indirizzo) all'attuale;
  - i riferimenti al formale sono riferimenti all'attuale (aliasing)
  - parametro di ingresso-uscita: main <-> proc
    modifiche al formale passano all'attuale
  - attuale: l-value, deve rappresentare una locazione

Passaggio per valore

```
void foo (int x){x = x+1;}
...
y = 1;
foo(y+1);
foo(y);
```

- Il formale `x` è una var locale (nel RdA)
- alla chiamata, l'attuale `y+1` è valutato e assegnato a `x`
  - viene fatta un operazione di assegnamento `x = y+1`
- nessun legame tra `x` nel corpo di foo e y nel chiamante
  - se `x` ha tipo semplice (modello a valore)

Passaggio per valore

- la sua semantica definita in termini della semantica dell'assegnamento
- può essere costoso per dati di grande dimensione, se l'operazione di assegnazione lo è
- Pascal (default), Scheme, C (unico modo);
- Implementazione:
  - nel RdA alloco una variabile associata al parametro formale
  - al momento della chiamata, il parametro attuale viene valutato e nel RdA inserito il suo valore

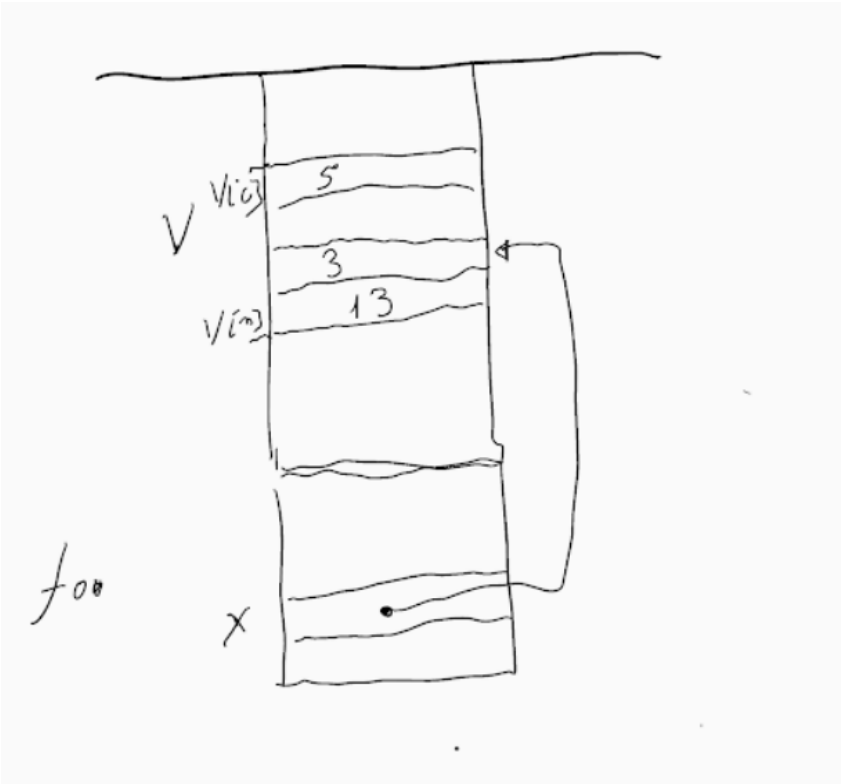
Passaggio per riferimento (per variabile)

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

- viene passato un riferimento (indirizzo, puntatore)
- l'attuale deve essere un `l-value` (un riferimento)
- il formale, `x`, diventa un alias dell'attuale ,`y`
  - considerato a basso livello, non presente nei linguaggi più recenti
- trasmissione bidirezionale tra chiamante e chiamato

Implementazione passaggio per riferimento

- nello stack di attivazione inserisco un puntatore
  - efficiente nel passaggio
- ma un diverso accesso ai dati nel chiamato
  - mascherato nell'uso del codice
  - il parametro formale è memorizzato come puntatore,
  - ma trattato come una variabile nel codice,



Nei linguaggi di programmazione

- raramente meccanismo di default: Fortran
- in alcuni linguaggi offerto come opzione:
  C++, PHP, Visual Basic .NET, C#, REALbasic, Pascal

  foo (ref int x)
  foo (var x : int)
- in altri simulabile:
  - C, ML, Rust simulato passando un puntatore, riferimento, indirizzo di memoria (call by address)
  - Java simulato usando i tipi riferimento, o il call by sharing

Simulazione in C

Tramite puntatori:

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

Call-by-sharing

Nei linguaggi con modello a riferimento: Python, Java, Ruby, JavaScript

il passaggio, nominalmente per valore, crea una condivisione (sharing) dell'argomento tra chiamato e chiamante

- argomento `x` tipo riferimento (assegnazione usa il modello a riferimento): strutturato, array, class,
- le modifiche nella funzione del parametro formale si ripercuotono sul parametro attuale
- formale riferimenti ad una singola copia dello stesso oggetto.

Call-by-sharing: esempi in Python

```
def f(a_list):
    a_list.append(1)

m = []
f(m)
print(m)
```

stampa `[1]`, ma

```
def f(a_list):
    a_list = [1]

m = []
f(m)
print(m)
```

stampa `[]`, nella procedura un nuovo vettore viene creato e assegnato a `a_list`.

Call by sharing: esempi in Java:

Le classi sono un tipo riferimento, l'assegnamento avviene copiando il riferimento (puntatore)

```
class A {
  int v;
  }
A y = new A(3)
...
void foo (A x){ x.v = x.v + 1;}
...

foo (y);
```

Lo stesso esempio con la classe `Integer` avrebbe un comportamento diverso

`x = x + 1;` crea un nuovo intero e lo assegna a `x`

Call by sharing

Il nome “call-by-sharing” non standard
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
- eventuali modifiche nel chiamato si possono ripercuotere sul chiamante

Riassumendo

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
  - copia condivisa e oggetti modificati, comportamento call-by-reference
  - copia condivisa e oggetti non modificabili (l'assegnazione crea una nuova istanza), comportamento call-by-value

Value, reference, sharing

Passaggio per valore:

- semantica semplice: si crea una copia locale
- implementazione abbastanza semplice,
- costoso il passaggio per dati di grande dimensione
- efficiente il riferimento al parametro formale
- necessità di altri meccanismi per comunicare main <- proc

Passaggio per riferimento:

- semantica complessa: si crea aliasing
- implementazione semplice
- efficiente il passaggio
- un po' più costoso il riferimento al parametro formale (indiretto)

Call-by-sharing

- a seconda dei casi, simile a call-by-value o a call-by-reference

Passaggio per costante (o read-only)

Il passaggio per valore garantisce la pragmatica: main -> proc
a spese dell'efficienza

- dati grandi sono copiati anche quando non sono modificati

Passaggio read-only (Modula-3; C, C++: `const`)

- il formale tratto come una costante
- nella procedura non è permessa la sua modifica:
  - no all'assegnamento;
  - no al passaggio ad altre procedure che possono modificarlo (per riferimento)
- controllo statico del compilatore:
  - vincolo blando in C
  - forte in Java

Passaggio per costante

Implementazione a discrezione del compilatore:

- parametri “piccoli” passati per valore
- parametri “grandi” passati per riferimento

In C:

```
int foo(const char *a1, const char *a2){
    /* le stringhe a1 a2 non possono essere modificate */
    }
```

In Java: final

```
    void foo (final A a){
        //qui a non può essere modificato
         }
```

Passaggio per risultato

Duale del passaggio per valore: main <- proc

```
void foo (result int x) {x = 8;}
...
y = 1;
foo(y);
foo(V[y+1]);
```

- l'attuale deve essere un l-value, `y` o `V[y+1]`
- al ritorno da `foo`, il valore di `x` è assegnato all'attuale `y`o `V[y+1]`
- nessun legame tra `y` iniziale del chiamante e `x` nel corpo di `foo`
  - non è possibile trasmettere dati alla procedura mediante il parametro
- pragmatica: usato quando una funzione deve restituire più di un singolo risultato
- Ada: out

Passaggio per valore/risultato

Insieme valore+risultato. Pragmatica: main <-> proc evitando aliasing

```
void foo (value-result int x)
    { x = x+1; }
...
y = 8;
foo(y);
```

- l'attuale deve essere un `l-value`
- il formale x è una var locale (sulla pila)
- alla chiamata, il valore dell'attuale è assegnato al formale
- al ritorno, il valore del formale è assegnato all'attuale
- nessun legame tra `x` nel corpo di foo e `y` nel chiamante
- costoso per dati grandi: copia
- Ada: in out (ma solo per dati piccoli; per dati grandi passa riferimento)

Passaggio per nome

L'espressione del parametro attuale viene passata, non valutata, alla procedura: Regola di copia:

- non si forza la valutazione del parametro attuale al momento della chiamata
- una chiamata alla procedura `P` produce l'effetto di
  - eseguire il corpo di P valutando,
    ogni volta che il parametro formale viene usato nel corpo
    l'espressione attuale

```
int sel (name int x, y) {
    return (x==0 ? 0 : y);}
...
z = sel(w, z/w);
```

Passaggio per nome

Introdotto Algol-W è il default,

Può simulare il passaggio per riferimento:

```
int y;
void foo (name int x) {x= x + 1; }
...
y = 1;
foo(y);
```

Se il parametro formale appare a sinistra dell'assegnamento,
il compilatore controlla che il parametro attuale sia un l-value.

- vediamo un caso più delicato. . .

Cattura delle variabili

```
int y = 1;
void fie (name int x){
    int y = 2;
    x = x + y;
}
fie(y);
```

- conflitto (e “cattura”) di variabili
- in quale ambiente valutare il parametro formale `x` nel corpo procedura?
- nell’ambiente della chiamata `fie(y)`
- Si realizza una “macro espansione”, in modo semanticamente corretto con una macro espansione può esserci in un fenomeno di cattura delle variabili

Valutazione multipla

Se un parametro formale appare più volte nel codice, il parametro attuale viene valutato più volte.

```
int V[] = {0, 1, 2, 3};
int y = 1;
void fie (name int x){
    int y = 2;
    x = x + y;
}
fie( V[y++] );
```

Implementazione: chiusura

Al momento della chiamata:

- oltre all'espressione, del parametro attuale devo fornire
- il suo ambiente di valutazione.

La coppia `<exp,env>` prende il nomo di _chiusura.

Come passare la chiusura?

- `exp` : un puntatore al testo di exp
- `env` : un puntatore di catena statica (sullo stack) al record di attivazione del blocco di chiamata

Implementazioni:

- complesso da implementare.
- Algol 60 e W: implementazioni storiche
- non utilizzato nei linguaggi imperativi attuali,
- linguaggi funzionali lazy: Haskell, ma in una versione più efficiente
- call-by-need, valuto l'argomento al più una volta
  - con l'assenza dei side-effect, valutazioni multiple restituisco sempre lo stesso valore
    call-by-need e call-by-name stesso comportamento
    call-by-need più efficente
- vedremo come le chiusure servono anche per passare funzioni come argomento

Simulazione del passaggio per nome:

- Parametro per nome = funzione fittizia, senza argomenti, in corpo viene valutato nell'ambiente del chiamante (un thunk, nel gergo Algol)

Esempio in Scheme:

```
(define (doublePlusOne e)
    (+ (e) (e) 1))

(define x 2)

(doublePlusOne (lambda () (+ x 3)))
```

thunk: meccanismo generale per simulare la valutazione lazy (per nome) in un linguaggi con valutazione eager (per valore).

Parametri di default

In alcuni linguaggi

Ad, C++, C#, Fortran, Python

possibili procedure con valori di default per alcuni argomenti

- è possibile fornire meno argomenti
- nel qual caso la procedura usa gli argomenti di default

Esempio in Python

```
def printData(firstname, lastname ='Mark', subject ='Math'):
     print(firstname, lastname, 'studies ', suject)

# 1 positional argument
printData('John')
# 2 positional arguments
printData('John', 'Gates')
printData('John', 'Physics')

# Output:
John Mark studies Math
John Gates studies Math
John Physics studies Math
```

Parametri con nome:

L'accoppiamento parametri attuali - parametri formali viene determinato dalla posizione,

in alternativa, in alcuni linguaggi, possibile specificare esplicitamente il ruolo svolto dai parametri attuali:

```
def printData(firstname, lastname ='Mark', subject ='Math'):
     print(firstname, lastname, 'studies ', suject)

# Mixed passing is possible
printData(firstname='John', subject='Physics')
printData('John', subject='Physics')

# Output:
John Mark studies Physics
John Mark studies Physics
```

Funzioni di ordine superiore

Alcuni linguaggi permettono di:

- passare funzioni come argomenti di altre funzioni (procedure)
  - caso relativamente semplice
  - presente in diversi linguaggi imperativi
  - funzioni come oggetti di secondo livello
- restituire funzioni come risultato di funzioni
  - caso più complesso
  - presente nelle versioni più recenti di linguaggi imperativi
  - tipico dei linguaggi funzionali
  - funzioni come oggetti di primo livello

Funzioni come argomento, semantica

Caso standard:

- il chiamante `main` passa una funzione `f`
- funzione `f` che verrà valuta, eventualmente più volte, nel chiamato `g`

Problema principale:

- in quale ambiente non locale viene valuta `f`
- politiche di scope: (statico, dinamico)
- ma non solo, politiche di binding: (deep, shallow)

Ambiente esterno del paramento funzione: esempio

```
int x = 1;
int f (int y){ return x + y; }
int g (int h (int i)){
    int x = 2;
    return h(3) + x;}
...
    int x = 4;
    int z = g(f);
```

Tre momenti della funzione `f`

- definizione di `f`: `int f (int y)`
- passaggio di `f` come argomento: `int z = g(f)`
- chiamata di `f`, tramite il nome `h`: `h(3) + x`

Tre possibili possibile alternative per l'ambiente esterno di `f`

Politiche di binding

Politiche di binding: quale ambiente non locale si applica al momento dell'esecuzione di `f` (quando chiamata via `h` dentro `g`)?

- scope statico: uso l'ambiente della definizione (come sempre)
- scope dinamico: due alternative:
  - ambiente al momento della creazione del legame h->f, ossia della chiamata di `g` con parametro `f`
    - deep binding
  - ambiente al momento della chiamata di `f`, in `g`, via `h`?
    - shallow binding

Politiche di binding: esempio

```
int x = 1;
int f (int y){ return x + y; }
int g (int h (int i)){
    int x = 2;
    return h(3) + x;}
...
    int x = 4;
    int z = g(f);
```

- tre dichiarazioni di `x`
- quando `f` sarà chiamata (tramite `h`)
  - quale `x` (non locale) sarà usata?
- in scope statico, la `x=1` esterna
- in scope dinamico,
  - deep binding: le `x=4` del blocco di chiamata
  - shallow binding: la `x=2` interna

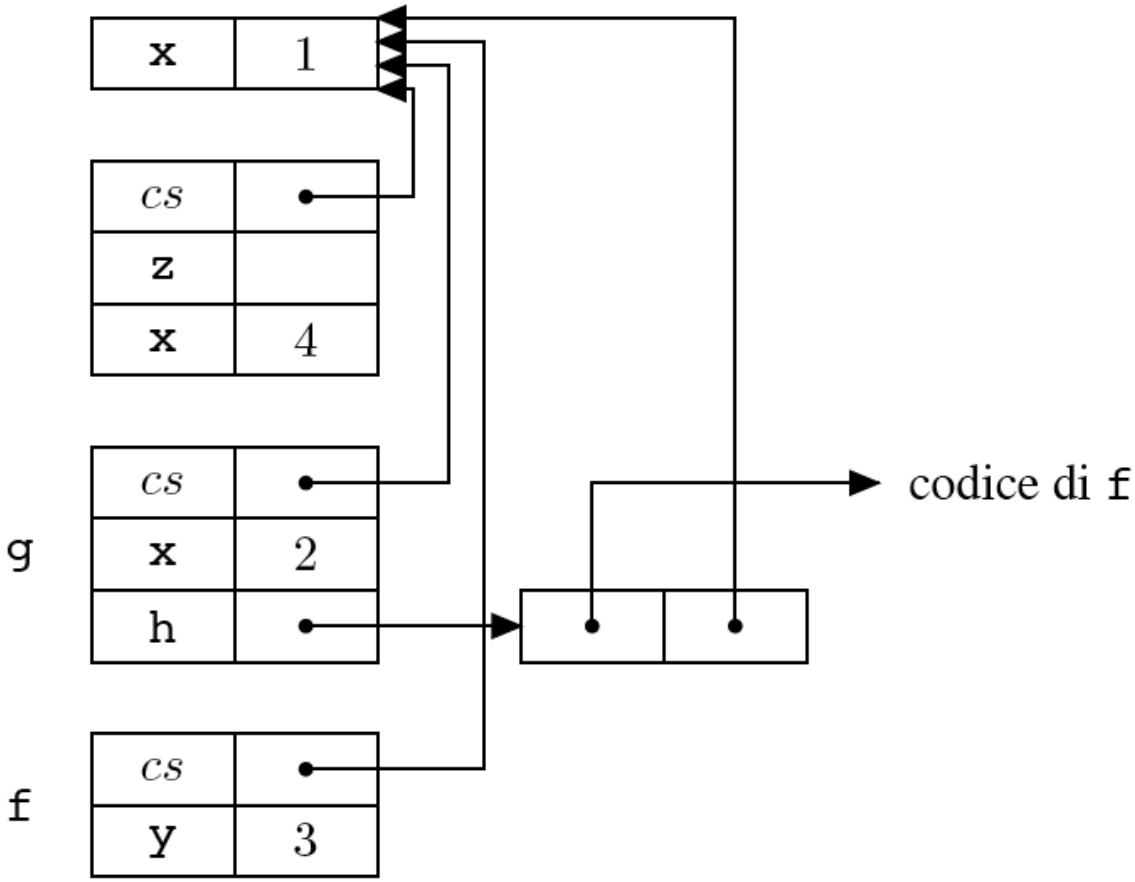
Implementazione uso delle chiusure

Analoga alla chiamata per nome:

alla chiamata di una procedura `g` con parametro attuale una procedura `f`

- si inserisce nel RdA di `g`, in corrispondenza ad `f`, una closure `<code, env>`
  - un riferimento al codice della procedura `f`
  - un riferimento al suo ambiente non locale di `h`.
- Alla chiamata della procedura `f` dentro `g`
  - si alloca (come sempre) il record di attivazione
  - usa come puntatore alla catena statica il riferimento (ambiente) fornito dalla chiusura `env`.

Deep binding e chiusure, scope statico



Scope dinamico: implementazione

Shallow binding

- non necessita chiusure
  - alla parametro funzione si associa solo il codice
  - per accedere a x, risali la pila
  - uso delle strutture dati solite (A-list, CRT)

Deep binding

- al parametro funzione si associa la chiusura
- per “ricordarsi” il RdA della chiamata
- parte della pila va saltata
  - per le procedure chiamate via parametro devo usare un link statico

Deep vs shallow binding con scope statico

A prima vista deep o shallow non fa differenza con scope statico

- è la regola di scope statico a stabilire quale non locale usare

Non è così: vi possono essere dinamicamente

- più istanze del blocco che dichiara la procedura passata come parametro,
  - la procedura viene dichiarata più volte
- la politica di binding cambia la dichiarazione da prendere in considerazione,
- accade in presenza di ricorsione

Per coerenza, viene sempre usato deep binding

- implementato con chiusure

Esempio

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

- due chiamate ricorsive di `foo`
- con due diversi valori di x
- `fie` dichiarata due volte, con due x non locali diversi
- quale dichiarazione considerare?

Funzioni come argomento in C

```
void mapToInterval ( void (*f)(int), int max ) {
  for ( int ctr = 0 ; ctr < max ; ctr++ ) {
    (*f)(ctr);
} }
void print ( int x ) {
  printf("%d\n", x); }
void main () {
...
mapToInterval(print, 100)  /* notare print senza parentesi */
```

L'asterisco nel parametro formale, `*f` necessario

Implementazione semplificata:

- in C non ci sono funzioni annidate,
- non servono chiusure,
- basta un puntatore alla funzione,
  - che verrà valutata nel ambiente globale.

Funzioni come risultato, caso semplice:

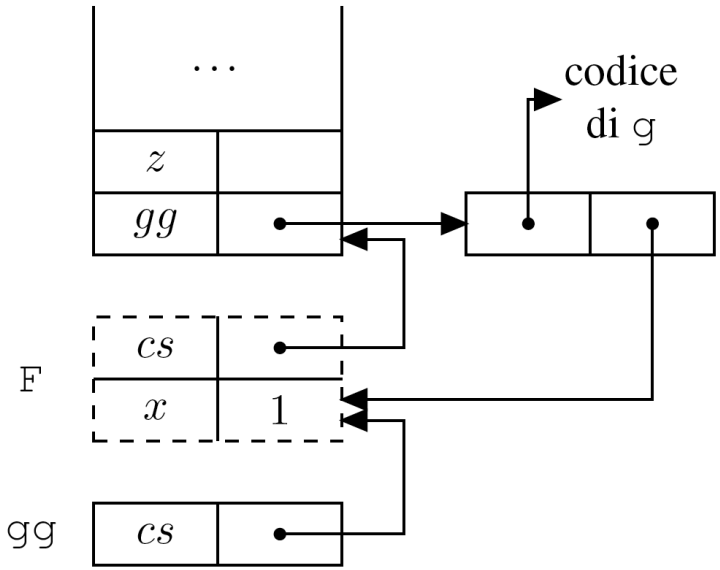
Si restituisce codice e ambiente non locale

```
int x = 1;
void -> int F () {
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

- La proc F ritorna una chiusura.

Funzioni come risultato, caso complesso

```
void -> int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
```



Implementazione delle funzioni come risultato

Linguaggi funzionali:

- Uso delle chiusure, ma . . .
- i record di attivazione possono dover persistere indefinitamente
  - perdita proprietà dello stack (LIFO)
- non si una stack di attivazione
  - record di attivazione sullo heap
  - le catene statica e dinamica collegano i record
  - invoca il garbage collector quando necessario

Linguaggi imperativi, con funzioni di ordine superiore:

- gestione sofisticata dei RdA

Implementazione funzioni di ordine superiore

Caso più semplice:

- Funzioni come argomento
  - si passa una chiusura

Caso più complesso:

- Funzione ritornata da una chiamata di procedura
  - occorre mantenere l'ambiente della funzione restituita:
  - disciplina a pila per i RdA non funziona più.

Eccezioni: “uscita strutturata”

Meccanismo per gestire eventi eccezionali:

- errori,
- situazioni non previste,
- ma anche terminare in anticipo la computazione, perché si sa già il risultato

Primi linguaggi gestivano attraverso istruzioni di salto: `GOTO`

- esempio di uso del `GOTO` non sostituibile dai meccanismi soliti (cicli, procedure)
- con `GOTO` difficile implementare correttamente l’uscita da una procedura
- per una gestione corretta si introduce un costrutto apposito.

Funzionamento:

- si definisce un insieme di eccezioni
- il programma può sollevare l'eccezione raise an exception
  - la normale computazione viene interrotta
  - si cerca
- un gestore l'eccezione sollevata,
  - codice da eseguire solo nel caso venga sollevata quell'eccezione
  - inserito in un blocco protetto.
- La ricerca del gestore comporta
  - terminazione comandi correnti
  - uscita da cicli
  - uscita da procedure
    - RdA devo poter essere deallocati.

Riassumendo

Tre costrutti:

- definizione delle eccezioni
- sollevamento dell'eccezione
- definizioni di blocchi protetti, con relativi gestori delle eccezioni.

Esempio Java

- la funzione `average` calcola la media di un vettore;
- se il vettore è vuoto, solleva eccezione
- le eccezioni sono sottoclassi di una classe `Throwable`
- devo dichiarare, `throws` se una funzione può generare un eccezione

```
class EmptyExcp extends Throwable {int x=0;};

int average(int[] V) throws EmptyExcp(){
   if (length(V)==0) throw new EmptyExcp();
   else {int s=0; for (int i=0, i<length(V), i++) s=s+V[i];}
   return s/length(V);
};
...
try{...
    average(W);
    ...
}
catch (EmptyExcp e) {write('Array vuoto'); }
```

Esempio Jave: gestione di un'eccezione

- il gestore è legato al blocco di codice protetto
- l'esecuzione del gestore rimpiazza la parte di blocco che doveva essere ancora eseguita
- possibile assegnare il parametro `x` in `EmptyExcp`
- comunico dei valori attraverso l'eccezione

```
class EmptyExcp extends Throwable {int x=0;};

int average(int[] V) throws EmptyExcp(){
   if (length(V)==0) throw new EmptyExcp();
   else {int s=0; for (int i=0, i<length(V), i++) s=s+V[i];}
   return s/length(V);
};
...
try{...
    average(W);
    ...
}
catch (EmptyExcp e) {write('Array␣vuoto'); }
```

Propagare un'eccezione

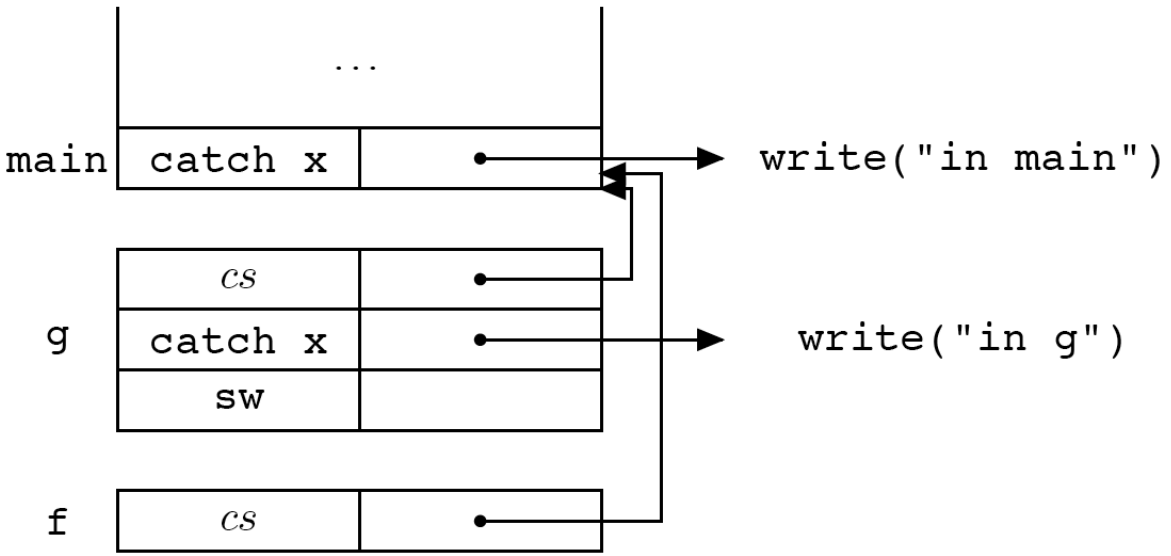
Se l'eccezione non è gestita nella procedura corrente:

- la routine termina, l'eccezione è ri-sollevata 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 default (ferma un programma con un messaggio di errore)

Eccezione si propaga lungo la catena dinamica

```
{
void f() throws X {
   throw new X();
   }

void g (int sw) throws X {
   if (sw == 0) {f();}
   try {f();} catch (X e) {write("in g");}
   }
...
```



Eccezione si propaga lungo la catena dinamica

Conseguenza:

- a tempo di esecuzione non posso determinare il gestore di un eccezione generato dentro una funzione (`f`)

Uso delle eccezioni per aumentare l'efficienza

Programma ricorsivo Scheme che

- moltiplica tutte i nodi di un albero binario,
- ogni nodo contiene un intero.

Soluzione efficiente:

- in scheme non serve dichiarare eccezione prima di usarle
- definisco una funzione ausiliaria di visita dell'albero
  genera l'eccezione `found-zero` se trova uno zero.
- 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,
- evito di dover continuare la visita quando so già il risultato finale.

Con CPS si posso ottenere comportamenti simili

Codice

```
(define-struct node (left right))

(define tree (make-node (make-node 10 9) (make-node 0 (make-no

(define (fold-product-aux nd)
  (cond
    [(number? nd)
     (if (equal? nd 0) (raise "zero-found") nd)]
    [(node? nd)
     (* (fold-product-aux (node-left nd))
        (fold-product-aux (node-right nd)))]))

(define (fold-product nd)
  (with-handlers
    ([(lambda (e) (equal? e "zero-found")) (lambda (e) 0)])
    (fold-product-aux nd)))
```

Implementare le eccezioni

Semplice:

- all'inizio di un blocco protetto (try):
  - alloco un nuove RdA con la lista dei gestori
- quando un'eccezione è sollevata:
  - cerco il gestore nel RdA corrente,
  - se non trova scendo nella lista dei RdA alla ricerca di un gestore

Inefficiente nel caso – il più frequente – che non si verifichi eccezione:

- ogni ingresso in un blocco protetto richiede lavoro sullo stack.

Implementare le eccezioni

Una soluzione migliore è quella di una mappa dei blocchi

- RdA solo in corrispondenza delle procedure
- nel RdA, si ha una mappa dei blocchi protetti:
  - indirizzo di inizio - fine di ogni blocco
  - quali eccezioni gestisce
- alla chiamate dell'eccezione si consulta la mappa per decidere quale gestore invocare

Strutturare i dati

I tipi di dato

Tipo di dato

Definizioni:

- Collezione di valori:
  - (omogenei ed effettivamente presentati),
  - dotata di un insieme di operazioni per manipolare tali valori
  - il compilatore associa ad ogni tipo, una sua rappresentazione a basso livello

A cosa servono i tipi?

Livello di progetto: organizzazione concettuale dei dati

- divido i dati in diverse categorie
- ad ogni categoria associo un tipo
- tipi come “commenti formali” su identificatori, espressioni

Livello di programma: identificano e prevengono errori

- controllo che i dati siano usate coerentemente al loro tipo
- controllo automatico
- costituiscono un “controllo dimensionale”:
  - 3+“Pippo” può essere sbagliato

Livello di implementazione: determinano l’allocazione dei dati in memoria

Il sistema di tipi di un linguaggio:

- tipi predefiniti
- meccanismi per definire nuovi tipi
- meccanismi relativi al controllo di tipi:
  - equivalenza
  - compatibilità
  - inferenza

Sistemi di tipo statici o dinamici

Quando avviene il controllo di tipo, type checking :

- statico: a tempo di compilazione,
  gran parte dei controlli in C, Java
- dinamico: durante l'esecuzione del codice
  Python, JavaScript, Scheme

Separazione non netta,

- quasi tutti i linguaggi fanno dei controlli dinamici
- alcuni controlli di tipo (es. dimensione degli array), possono avvenire solo a tempo di esecuzione.

Statici: vantaggi e svantaggi

- vengono anticipati gli errori
- meno carico di lavoro, molti dei controlli di tipo a tempo di esecuzione possono essere omessi,
- a volte più prolissi, bisogna introdurre informazioni di tipo nel codice,
  (ma si documenta meglio il codice)
  alcuni linguaggi (Python, ML) usano type inference per rendere le dichiarazioni di tipo facoltative
- meno flessibili, per prevenire possibili errori di tipo, si impedisce codice perfettamente lecito,

```
(define (f g) (cons (g 7) (g #t)))
(define pair_of_pairs (f (lambda (x) (cons x x))))
```

- a costo di un certa complessità, programmi equivalenti, possono avere tipo statico

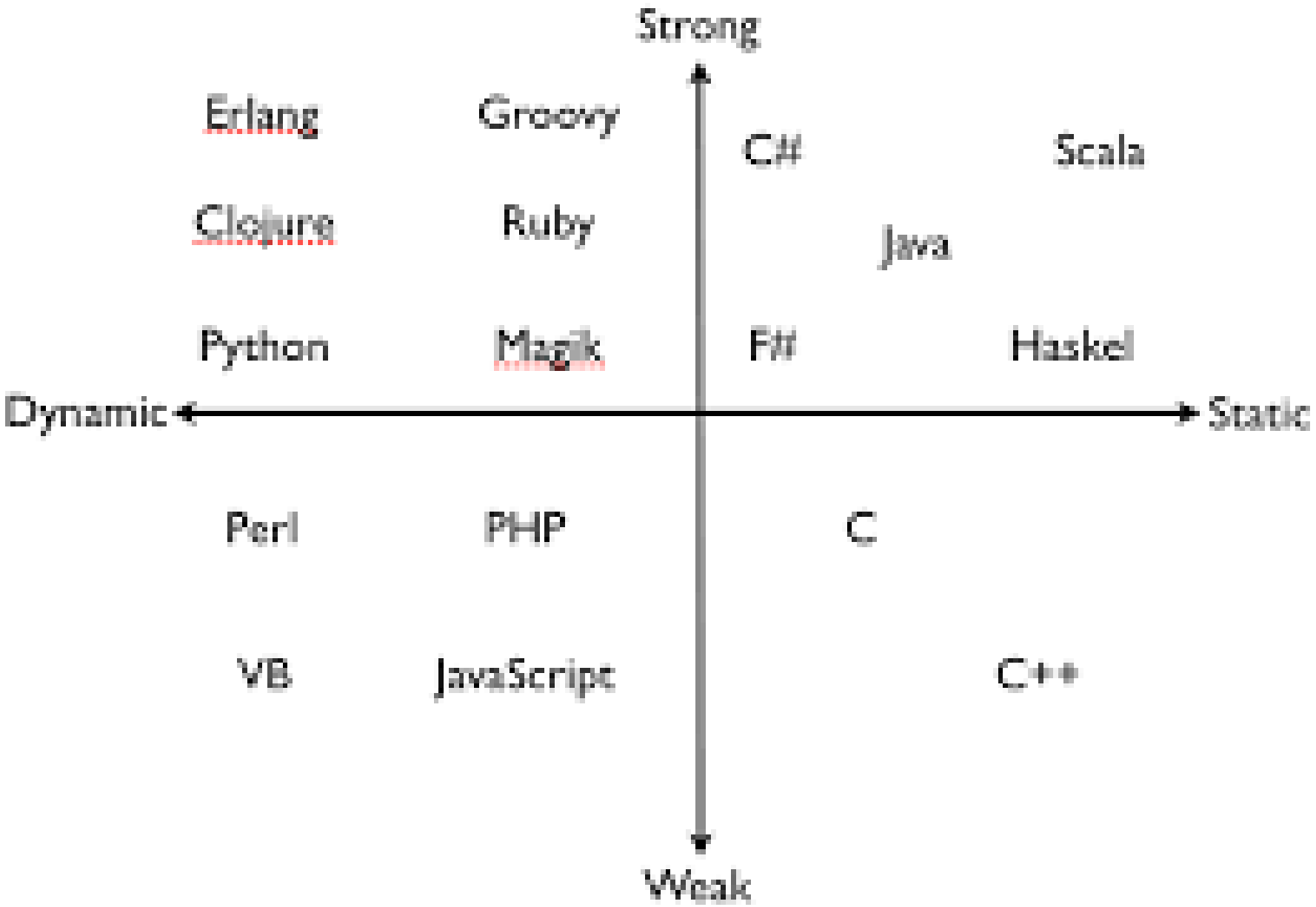
Dinamici: svantaggi e svantaggi

- bisogna eseguire il codice per trovare l'errore
  (i test vanno comunque fatti anche per i s.t. statici)
- informazioni sul tipo da inserire nei dati e controlli di tipo da fare a tempo di esecuzione
  (test non pesanti, ottimizzazione possibile)
- più concisi (meno definizioni di tipo nel codice)
- più flessibili

Strong - weak type system

- Strong type system impediscono, rendono difficile, che errori di tipo non vengano rilevati
  (type safe)
- Weak type system permettono una maggiore flessibilità a costo della sicurezza,
  - errori di tipo
    es.: uso una sequenza di 4 caratteri come un numero intero possono essere non rilevati (ne a tempo di esecuzione, ne a tempo di compilazione)

Concetti indipendenti:



Una catalogazione dei tipi e dei loro valori:

Possiamo distinguere in valori:

- denotabili: rappresentabili mediante identificatori
- esprimibili: rappresentabili mediante espressioni (diverse da identificatori - dichiarazioni)
- memorizzabili: che possono essere salvati in memoria, assegnati ad una variabile

Valori delle stesso tipo ricadono tutti nella stessa categoria,

Linguaggi diversi fanno scelte diverse

- se un specifico tipo sia denotabile, esprimibile memorizzabile:
- (es. funzioni)

Tipi predefiniti, scalari

Scalari: i valori possono essere ordinati, messi in scala

Booleani

val:	true, false (Java), (Scheme: `#t, #f`)
op:	or, and, not, condizionali
repr:	un byte
note:	C non ha un tipo Boolean

Caratteri

val:	`a,A,b,B, ..., è,é,ë, ; , ‘,`
op:	uguaglianza; code/decode; dipendenti dal linguaggio
repr:	un byte (ASCII - C), due byte (UNICODE - Java), lunghezza variabile (UTF8 - opzione in Java)

Tipi scalari

Interi

val: 0,1,-1,2,-2,. . . ,maxint

op: +, -, *, mod, div, . . . , più un buon numero di funzioni definite in varie librerie.

repr:
- alcuni byte (1, 2, 4, 8, 16), complemento a due
  - in alcuni linguaggi, più tipi interi per specificare il numero di byte della rappresentazione (`byte` o `char`, `short`, `int`, `long`, `long long`)
  - C contiene anche la versione unsigned: binario puro
  - Scheme e altri linguaggi, dispongono di interi di dimensione illimitata.

Tipi scalari

Floating point

val: valori razionali in un certo intervallo
op: +, -, *, /, . . . , più un buon numero di funzioni definite in varie librerie.
repr: alcuni byte (4, 8, 16?); (`float`, `double`, `Quad` o `float128`), notazione standard IEEE-754, virgola mobile

- Nota: alcuni linguaggi non specificano la lunghezza delle rappresentazione usata per interi e floating-point, questo può creare problemi nella portabilità tra compilatori diversi.

Tipi numerici

Tipi numerici non supportati dall'hardware,
non presenti in tutti i linguaggi:

- Complessi
  - repr: coppia reali -presenti Scheme, Ada, definiti tramite librerie in altri linguaggi.
- Fixed point
  - val: razionali ma non in nozione esponenziale
  - repr: alcuni byte (2 o 4); complemento a due o BCD (Binary Code Decimal), virgola fissa
  - presenti in Ada.
- Razionali
  - rappresentazione esatta dei numeri frazionari
  - repr: coppie di interi
  - presenti in Scheme

Tipi numerici in Scheme

Scheme considera 5 insiemi di numeri

```
number
complex
real
rational
integer
```

ciascuno sovrainsieme dei sottostanti

- rappresentazione interna nascosta, lasciata all'implementazione
  per gli interi normalmente rappresentazione a lunghezza arbitraria
- si distingue tra `exact` (integer, rational)
  e `inexact` numbers (real, complex).

Tipi Scalari

Void

- val: un solo valore
- op: nessuna operazione
- repr: nessun bit

permette di trattare procedure, comandi
come caso particolare di funzioni, espressioni

```
void f (...) {...}
```

la procedura `f` vista come funzione,
deve restituire un valore (e non nessuno)

il valore restituito da `f`, di tipo void, è sempre lo stesso,
non è possibile e non serve specificarlo nell'istruzione `return`

Una prima classificazione

Tipi ordinali (o discreti):

- booleani, interi, caratteri
- ogni elemento possiede un succ e un prec (eccetto primo/ultimo)
- altri tipi ordinali:
  - enumerazioni
  - intervalli (subrange)
- uso
  - vi si può “iterare sopra”
  - indici di array

Tipi scalari non ordinali

- floating-point, complessi

Enumerazioni

- introdotti in Pascal
-

```
type Giorni = (Lun,Mar,Mer,Giov,Ven,Sab,Dom);
```

- programmi di più leggibili,
  - non codifico il giorno della settimana con un intero, uso una rappresentazione leggibile
- valori ordinati: `Mar < Ven`
- iterazione sui valori: `for i:= Lun to Sab do ...`
- op: `succ, pred`, confronto <=
- rappresentati con un byte (`char, byte`)

Enumerazioni

In C, Java:

```
enum giorni = {Lun,Mar,Mer,Giov,Ven,Sab,Dom};
```

In C la definizione precedente è equivalente a

```
typedef int giorni;
const giorni Lun=0, Mar=1, Mer=2, ...,Dom=6;
```

- Java distingue tra `Mar` e `1`,
  valori non compatibili di tipi distinti
- C non distingue

Intervalli (subrange)

- introdotti in Pascal, non presenti in in C, Scheme o Java
  in Java implementabili come sottoggetti
- i valori sono un intervallo dei valori di un tipo ordinale (il tipo base dell'intervallo)
- Esempi:

```
type MenoDiDieci = 0..9;
type GiorniLav = Lun..Ven;
```

- rappresentati come il tipo base
- perché usare un tipo intervallo invece del suo tipo base:
  - documentazione "controllabile" (con type checking dinamico)
  - potenzialmente è possibile risparmiare memoria,

Tipi composti, o strutturati, o non scalari

Record o struct

- collezione di campi (field), ciascuno di un (diverso) tipo
- un campo è selezionato col suo nome

Record varianti o union

- record dove solo alcuni campi (mutuamente esclusivi) sono attivi ad un dato istante

Array

- funzione da un tipo indice (scalare) ad un altro tipo
- array di caratteri sono chiamati stringhe; operazioni speciali

Tipi composti, o strutturati, o non scalari

Insieme

- sottoinsieme di un tipo base

Puntatore

- riferimento (reference) ad un oggetto di un altro tipo

Funzioni, procedure, metodi, oggetti.

Record o struct

- raggruppare dati di tipo eterogeneo
- C, C++, CommonLisp, Algol68: struct (structure)
- Java: non ha tipi record, sussunti dalle classi record: classe senza metodi,
- Esempio, in C:

```
struct studente {
char nome[20];
int matricola; };
```

- Selezione di campo:

```
studente s;
s.nome = "Mario"
s.matricola=343536;
```

- record possono essere annidati

Struct in Java

Le definizioni precedenti sono simulate in Java come:

```
class Studente {
public String nome;
public int  matricola;
}

Studente s = new Studente();
s.nome = "Mario";
s.matricola = 343536;
```

Record o struct

- memorizzabili, denotabili, ma non sempre esprimibili
  - non è generalmente possibile scrivere un espressione, diversa da un identificatore, che definisca un record
  - C lo può fare, ma solo nell'inizializzazione di variabile record,
  - uguaglianza generalmente non definita (eccezione: Ada)
  - struct sono valori esprimibili in Scheme

Struct in Scheme

Posso implementare record come liste, e definendo delle funzioni che

- costruiscono record
- accedono i campi
- testano il tipo di record

```
(define  (book title authors)  (list 'book title authors))
(define  (book-title b) (car (cdr b)))
(define  (book-? b) (eq? (car b) 'book))
```

usate come

```
(define bazaar
  (book
   "The Cathedral and the Bazaar"
   "Eric S. Raymond" ))
(define titolo_bazar (book-title bazaar)

(book? bazar)
```

Costruttori automatici

Racket, MIT-Scheme:

```
(define-structure book title authors)
(define bazaar
  (make-book
   "The Cathedral and the Bazaar"
   "Eric S. Raymond" ))
(define titolo_bazar (book-title bazaar)

(book? bazar)
```

Costruttori automatici

Racket

```
(struct  book (title authors))

(define bazaar
  (book
   "The Cathedral and the Bazaar"
   "Eric S. Raymond" ))
(define titolo_bazar (book-title bazaar)

(book? bazar)
```

Costruttori automatici

L'esempio studente diventa

```
(define-struct studente (nome matricola))

(define s (make-studente "Mario" 343536))

(studente-nome s)
```

- la definizione `(define-struct studente (nome matricola))` porta alla creazione delle opportune funzione di
  - creazione `make-studente`
  - selezione campo `studente-nome`
  - test `studente?`

Record: memory layout

- memorizzazione sequenziale dei campi
- allineamento alla parola (4, 8 byte)
  - spreco di memoria
- packed record
  - disallineamento
  - accesso più complesso (in assembly)
- riordino dei campi può permettere
  - risparmio di spazio
  - mantengo le parole di memoria allineate

Record varianti

In un record variante alcuni campi sono alternativi tra loro:

solo alcuni di essi attivi in un dato istante

```
type studente = record
nome : packed array [1..6] of char;
matricola: integer;
case fuoricorso : Boolean of
    true: (ultimoanno: 2000..maxint);
    false:(anno:(primo,secondo,terzo);
           inpari:Boolean;)  end;
var s: studente;
s.fuoricorso := true;
s.ultimoanno:= 2001;
```

- I due campi (le due varianti) `ultimoanno` e `anno` possono condividere la stessa locazione di memoria
- Il tipo del tag fuoricorso può essere un qualsiasi tipo ordinale

Record varianti: memory layout

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

Due versioni di tipo stud:

Record varianti

Possibili in molti linguaggi C: union + struct

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

- Pascal (Modula, Ada) unisce unioni e record con eleganza
  - in Pascal: `s.anno`
  - in C: `s.campivarianti.studInCorso.anno`
    nessuna correlazione esplicita tra campo fuoricorso e campi 'varianti'

Union, Variant e type safety

I tipi unione permettono facilmente di aggirare i controlli di tipo

```
union Data {
    int i;
    float f;
    char str[20];
} data;
float y;
...
data.str = "abcd";
y = data.f;
```

- codifica ASCII di “abcd”, trattata come numero reale, senza alcun avviso da parte del sistema

Union, Variant e type safety

Variant del Pascal con il campo case, agevolano la scrittura di codice di controllo ma

- nessuna garanzia
- nessun controllo forzato
- vincoli di tipo facilmente aggirabili

Java non usa i tipi unione.

Motivazione dei tipi unione,

- risparmio di spazio, attualmente meno importante
- alcuni tipi naturalmente descritti da tipi unione
  - lista: lista vuota o elemento seguito da lista
  - albero binario: foglia o (nodo, sottoalbero sinistro, sottoalbero destro)

Esistono versioni sicure dei tipi unione

Algol 68, Haskell, ML:

- esistono tipi equivalenti ai tipi unione
  più alterative possibili
- ogni alternativa viene etichettata
- analizzo i tipi unione attraverso un costrutto `case`,
  - per ogni possibile caso, definisco il codice che lo gestisce.

```
case data in
(int i) : a = i + 1
(float j) : x = sqrt(j)
```

Lauree magistrali in informatica

È tradizione presentare agli studenti del terz'anno una descrizione delle lauree magistrali

Quest'anno la presentazione avviene online, domani, venerdì 16 ore 12

Agostino Dovier, coordinatore corso di laurea

Una presentazione, per l'anno passato, è disponibile in rete:

www.dmif.uniud.it/2020/03/nuove-magistrali-informatica

Struct (e quasi Union) in Java con le sottoclassi

```
enum AnnoCorso {PRIMO, SECONDO, TERZO};
class Studente {
   public String  nome;
   public int matricola;
   }
class StudenteInCorso extends Student {
    public AnnoCorso anno;
}
class StudenteFuoriCorso extends Student {
    public int ultimoAnno;
} ....
      StudenteInCorso mario = new StudenteInCorso();
      mario.nome = "mario";
      mario.matricola = 456;
      mario.anno = AnnoCorso.TERZO;
```

ma non posso trasformare `mario` in un `StudenteFuoriCorso`

Array

Collezioni di dati omogenei:

- funzione da un tipo indice al tipo degli elementi, rappresentata in modo estensionale
- indice: tipo ordinale, discreto
- elemento: “qualsiasi tipo memorizzabile” (raramente un tipo funzionale)

Dichiarazioni

```
int vet[30];      // tipo indice: tra 0 e 29, C
int[] vet;        // Java, notazione preferita
int vet[];        // Java, notazione alternativa
var vet : array [0..29] of integer;  // Pascal
```

Array multidimensionali

Due possibili definizioni:

- array con più indici
- array con elementi altri array

In Pascal le due definizioni sono equivalenti

```
var mat : array [0..29,'a'..'z'] of real;
var mat : array [0..29] of array ['a'..'z'] of real;
```

Possibili entrambe ma non sono equivalenti in Ada:

- la seconda permette slicing

C e Java non posso dichiarare array con più indici.

```
int mat[30][26]    // C
int[][] mat        // Java
```

Array: operazioni

Principale operazione permessa:

- selezione di un elemento: `vet[3]` `mat[10,'c']` `mat[10][12]`
  - l'elemento può essere letto o modificato

Alcuni linguaggi permettono slicing:

- selezione di parti contigue di un array
- Esempio: in Ada, con

```
mat : array(1..10) of array (1..10) of real;
```

`mat(3)` indica la terza riga della matrice quadrata `mat`

- possibile anche in C `mat[3]`
- selezioni più sofisticate, non necessariamente intere righe, Python

```
y=(1,2,3,4)
x = y[:-1]
```

Calcolo vettoriale

- operazioni vettoriali, operazione aritmetiche estese agli array
- usate in: calcolo scientifico, grafica, criptografia
- presenti anche in assembly.
- Fortran90: A+B somma gli elementi di A e B (dello stesso tipo)

Memorizzazione degli array

Elementi memorizzati in locazioni contigue.

Per array multidimensionali, due alternative:

- ordine di riga:
  `V[0,0] V[0,1] ...V[0,9] V[1,0] ...`
  maggiormente usato;
  le righe, sub-array di un array, memorizzate in locazioni contigue
- ordine di colonna:
  `V[0,0] V[1,0] V[2,0]... V[13,0];V[0,1];...`

Ordine rilevante per efficienza in sistemi con cache, per algoritmi di scansione del vettore

- vettore memorizzato per righe, scansione per colonne:
  - esamino punti distanti in memoria
  - genero un numero più alto di cache miss

Array: calcolo indirizzi

Calcolo locazione corrispondente a `A[i,j,k]` (per riga),

- `i` indice piano, `j` riga, `k` colonna
- gli indici partano da `0`
- `A : array[l1, l2, l3]` of `elemType`
- `elemType A[l1][l2][l3]`
  - `S3`: dimensione di (un elemento di) `elem_type`
  - `S2` = `l3 * S3` dimensione di una riga
  - `S1` = `l2 * S2` dimensione di un piano
- locazione di `A[i,j,k]` è:

```
a              indirizzo di inizio di A
+ i*S1      = ind di inizio del piano di A[i,j,k], sottoma
+ j*S2      = ind di inizio della riga di A[i,j,k], sottov
+ k*S3      = ind di A[i,j,k]
```

Array: shape

- Forma (o shape): numero di dimensioni e intervalli dell'indice per ogni dimensione

Può essere determinata in vari istanti:

- forma statica (definita a tempo di compilazione)
- forma fissata al momento dell'esecuzione della dichiarazione, chiamata della procedura e definizione delle variabili locali (per vettori locali a una procedura)
- forma dinamica, le dimensioni variano durante l'esecuzione (Python, JavaScript)

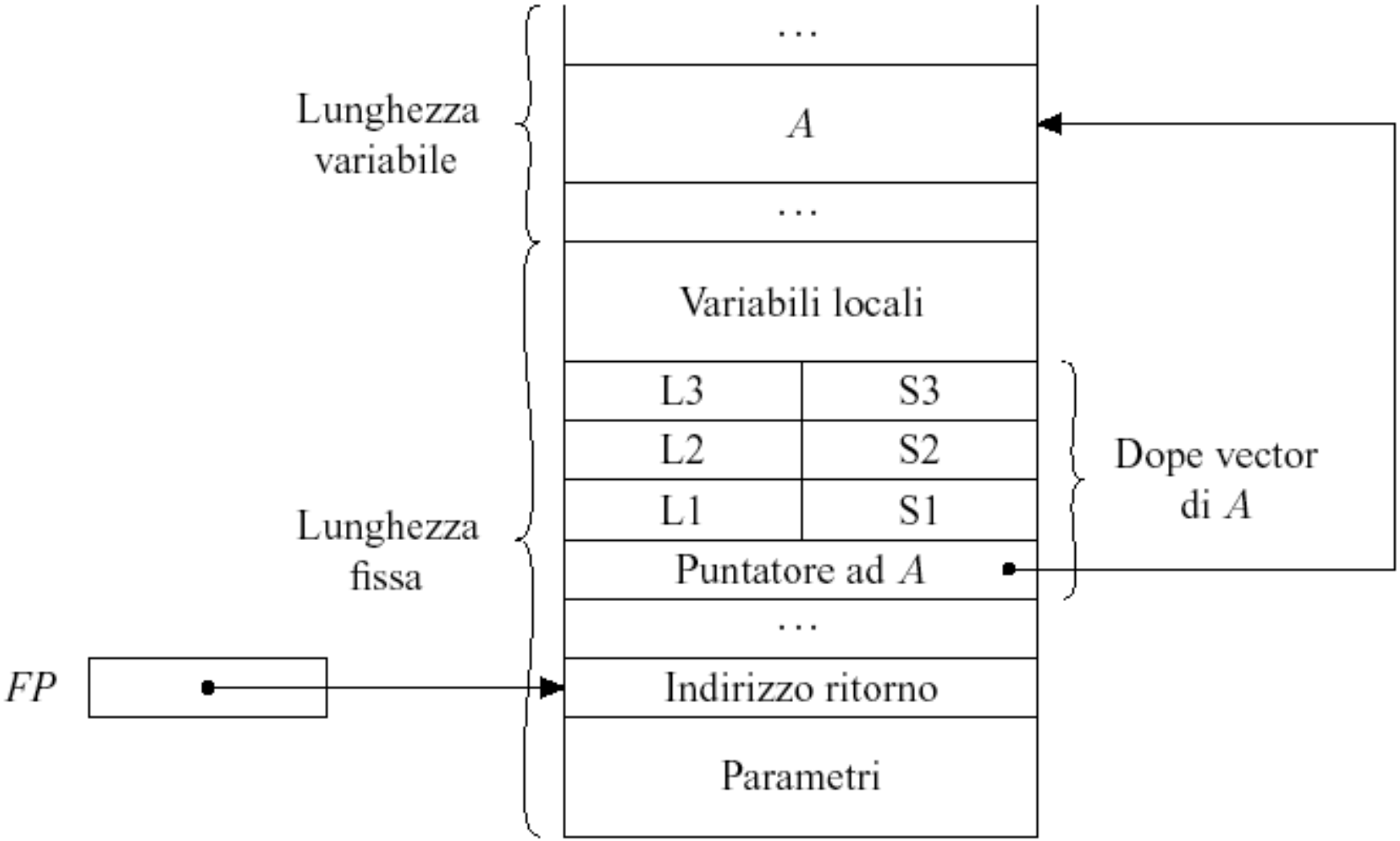
Dope vector (descrittore del vettore)

Come si accede ad un vettore?

- Con forma statica, informazioni sulla forma dell'array è mantenuta dal compilatore,
- Se la forma non statica, info mantenuta in un descrittore dell'array detto dope vector che contiene:
  - puntatore all'inizio dell'array (nella parte variabile)
  - numero dimensioni
  - limite inferiore (se a run-time vogliamo controllare anche l'out-of-bound anche limite superiore)
  - occupazione per ogni dimensione (valore Si)
- Il dope vector è memorizzato nella parte fissa del RdA
- Per accedere al vettore, si calcola l'indirizzo a run-time, usando il dope vector

Esempio: RdA con dope vector

```
A : array[l1..u1, l2..u2, l3..u3] of elem_type
```



Memorizzazione dell'array

Dove memorizzare un array dipende dalla forma:

- Statica: RdA
- Fissata al momento della dichiarazione: dope vector nella parte iniziale, e vettore in fondo al RdA
- Forma dinamica: heap, solo il dope vector nel RdA

Per Java:

- gli array sono oggetti,
- memorizzati nella heap
- la definizione di una variabile vettore non alloca spazio
- spazio allocato con una chiamata a `new`

```
int[] vettore = new int[4];
```

- array multidimensionali: memorizzati come vettori di puntatori

Array e sicurezza

Controllo degli indici di un array

- controllo necessariamente dinamico
- svolto solo da alcuni linguaggi
- importante per:
  - type safety: accedo a zone arbitrarie della memoria con tipo sbagliato
  - sicurezza del codice: buffer overflow,
    - istruzioni di scrittura in un vettore con indici fuori range,
    - possono scrive in zone arbitrarie del RdA,
    - cambiando gli indirizzo di ritorno della procedura,
    - si può di eseguire codice arbitrario, es. codice scritto nel vettore

Insiemi

Disponibili in alcuni linguaggi,

- `set of char S`
  `set of [0..99] I`
- operazioni insiemistiche
- implementabili mediante vettori, compatti, di booleani
  - definiscono la funzione caratteristica
  - le operazione logiche, point-wise, implementano le corrispondenti operazioni aritmetiche
- per insiemi con un universo di grosse dimensioni, conveniente implementare tramite tabelle hash,
- se non presenti nel linguaggi, implementati tramite librerie, classi.

Puntatori

identificano: l-value, locazioni di memoria

- Valori : riferimenti (l-valori); costante null (o nil)
  - tutti i puntatori strutturalmente uguali, tipizzati in base all'oggetto puntato
- Tipi: nel tipo del puntatore specifico il tipo dell'oggetto puntato

```
int  *i  \\ int* i
char *a  \\ char* a
```

- Operazioni:
  - creazione di un oggetto puntato
    - funzioni di libreria che alloca e restituisce l'indirizzo (es., malloc)
  - dereferenziazione
    - accesso al dato "puntato": *p
  - test di uguaglianza

Puntatori, scelte nei diversi linguaggi

L'uso dei puntatori non ha molto senso in linguaggi con modello a riferimento

- ogni variabile contiene un riferimento in memoria al dato associato (puntatore)

In alcuni linguaggi, puntatori fanno riferimento al solo heap

- non in C

```
int i = 5;
p = &i;
```

Puntatori e array in C

- Array e puntatori sono intercambiabili in C

```
int n;
int *a;        // puntatore a interi
int b[10];     // array di 10 interi
...
a = b;          // a punta all'elemento iniziale di b
n = a[3];       // n ha il valore del terzo elemento di b
n = *(a+3);     // idem
n = b[3];       // idem
n = *(b+3);     // idem
```

- Aliasing: `a[3]=a[3]+1;`
  modificherà anche `b[3]` stessa locazione di memoria.

Aritmetica sui puntatori

- In C, sui puntatori sono possibili alcune operazione aritmetiche, con significato diverso
  - `a+3`, incrementa il valore di `a` di `12` (`3*` dimensione intero)
  - in generale gli incrementi sono moltiplicati per la dimensione dell'elemento del vettore
- Nessuna garanzia di correttezza, posso accedere a zona arbitrarie di memoria
- Array multidimensionali, sono equivalenti le espressioni:

```
b[i][j]
(*(b+i))[j]
*(b[i]+j)
*(*(b+i)+j)
```

Puntatori e tipi di dato ricorsivi

Uso principale dei puntori:

- implementare strutture dati ricorsive
  - liste
  - code
  - alberi binari
  - alberi

```
type int_list = union{ struct val {int info;
                                   int_list next;};
                       void null}
```

Spesso implementate con record e puntatori.

```
typedef struct listElem element;
struct listElem {
    int info;
    element *next;
}
```

Dangling reference

- Problema con l'uso dei puntatori: si fa riferimento ad una zona delle memoria che contiene dati di un tipo non compatibile, arbitrari.
- Possibili cause
  - aritmetica sui puntatori
  - deallocazione dello spazio sul heap `free(p)`
  - deallocazione dei RdA (vedremo esempi)
- Soluzioni
  - restringere l'uso dei puntatori
  - nessuna deallocazione esplicita e meccanismi di garbage collection
  - oggetti puntati solo nell'heap
  - introdurre dei meccanismi di controllo nella macchina astratta

Dangling reference

A che zona della memoria fanno riferimento i puntatori.

- per sicurezza l'heap.
- in C non necessariamente.

Copiare un vettore in un altro, come riferimento, può portare a riferimenti pendenti

```
int *ref
int vett1[2];
void proc (){
   int i = 5;
   int vett2[2];
   ref = &i;
   vett1 = vett2;
   }
```

dopo una chiamata a `proc`, `vett1` e `ref` puntano a zone deallocate della memoria.

Inferenza di tipo

- Le dichiarazione definiscono un tipo per ogni identificatore di variabile o costante
- Il compilatore associa un tipo agli altri oggetti del codice:
  - espressioni e sottoespressioni
  - blocchi di comandi
  - funzioni o procedure
- Algoritmi di type inference più o meno sofisticati:
  - In Python, ML, Haskell non serve definire il tipo delle variabili, dedotti dal conteso
  - tipo per le funzioni presente o meno

Equivalenza e compatibilità tra tipi

Due relazioni tra tipi:

- Equivalenza: 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)
  i due tipi sono perfettamente intercambiabili

- Compatibilità: `T` è compatibile con `S` quando

  valori di `T` possono essere usati in contesti dove ci si attende valori `S`
  ma non necessariamente il viceversa.

Equivalenza tra tipi: per nome

Due tipi sono equivalenti se sono lo stesso nome, identificatore di tipo,

- essere espressioni di tipo uguali non è sufficiente
- esempio, con equivalenza per nome dopo

```
x : array [0..4] of integer;
y : array [0..4] of integer
```

- le variabili `x` e `y` hanno tipi differenti
- Usata in Pascal, Ada, Java

Equivalenza per nome lasca (loose) o stretta

Data la seguente dichiarazione, A e B sono tipi equivalenti?

```
type A =  ... ;
type B = A;  {* alias  *}
```

- equivalenza per nome stretta: no
- equivalenza per nome lasca: si
  (Pascal, Modula-2)
  - spiegazione: una dichiarazione di un tipo alias di tipo (`B = A`) non genera un nuovo tipo (`B`) ma solo un nuovo nome per `A`

Passaggio dei parametri

- nel passaggio dei paramatri si può richiedere l'equivalenza di tipo tra parametro formale e attuale
- codice come:

```
int[4] sequenza = {0, 1, 2, 3};
void ordina (int[4] vett);
...
ordina(sequenza)
```

lecito in C ma non in Pascal, Ada,
si usano equivalenze di tipo diverse

Con equivalenza per nome devo scrivere:

```
typedef int[4]  vettore
vettore sequenza = {0, 1, 2, 3};
void ordina (vettore vett);
...
ordina(sequenza)
```

Equivalenza strutturale

Due tipi sono equivalenti se hanno la stessa struttura:

> *Definizione. L'equivalenza strutturale tra tipi è la (minima) relazione di equivalenza tale che:*
>
> - *se un tipo* `T` *è definito come* `type T = espressione` *allora* `T` *è equivalente a* `espressione`
> - *due tipi, costruiti applicando lo stesso costruttore a tipi equivalenti, sono equivalenti*

Equivalenza controllata per riscrittura, definizione alternativa:

- Se da due tipi complessi `A` e `B`, riscritti nelle loro componenti elementari, espandendo le definizioni, generano la stessa espressione di tipo, allora `A` e `B` sono equivalenti strutturalmente

```
typedef struct{int i; float f} coppia;
typedef struct{int j; coppia c} A;
typedef struct{int j; struct{int i; float f} c} B;
```

Diverse interpretazione dell'equivalenza strutturale

- Vanno considerate equivalenti?

```
typedef struct{int i; float f} coppia1;
typedef struct{float f; int i} coppia2;
```

Generalmente no, si per ML o Haskell.

o

```
typedef int[0..9] vettore1
typedef int[1..10] vettore2
```

Generalmente no, sì per Ada e Fortran.

Casi di equivalenze strutturali accidentali

- Equivalenza strutturale: a basso livello,
  non rispetta l'astrazione che il programmatore inserisce col nome.

```
type student = {
    name: string,
    address: string
}
type school = {
    name: string,
    address: string
}
type age = float;
type weight = float;
```

- Con l'equivalenza strutturale, possiamo assegnare un valore `school` ad una variabile `student` o un valore `age` ad una variabile `weight`.

Nei linguaggi,

- spesso si usa in miscuglio delle due equivalenze
  - a seconda del costruttore di tipo usato, si usa o meno l'equivalenza strutturale
- linguaggi recenti tendono a preferire l'equivalenza per nome

Esempi di equivalenza tra tipo

- C: equivalenza per nomi sui tipi `struct` equivalenza strutturale sul resto (array)
- equivalenza per nomi forte in Ada:

```
x1, x2: array (1 .. 10) of boolean;
```

`x1` e `x2` hanno tipi diversi

- Equivalenza strutturale in ML, Haskell:

```
type t1 = { a: int, b: real };
type t2 = { b: real, a: int };
```

`t1` e `t2` sono tipi equivalenti

Compatibilità

Un tipo `T` è compatibile con un tipo `S` quando oggetti di tipo `T` possono essere usati in contesti dove ci si attende valori di tipo `S`

- Esempio: `int` compatibile con `float`

```
int n = 5
float r = 5.2;
r = r + n;
```

- Oggetto: `n`
- Contesto: `r = r + _ ;`

Esempi di compatibilità

Quali tipi siano compatibili dipende dal linguaggio:

Esempi di casi di compatibilità, via via più laschi
`T` è compatibile con `S` se:

- `T` e `S` sono equivalenti;
- I valori di `T` sono un sottoinsieme dei valori di `S` (tipi intervallo);
- tutte le operazioni sui valori di `S` sono possibili anche sui valori di `T` (“estensione” di record, sottoclasse);
- i valori di `T` corrispondono in modo canonico ad alcuni valori di `S` (int e float), (int e long);
- I valori di `T` possono essere fatti corrispondere ad alcuni valori di `S` (float e int con troncamento) (long e int);

Conversione di tipo

Se `T` compatibile con `S` avviene una conversione di tipo.
un espressione di tipo `T` diventa di tipo `S`

Questa conversione di tipo, nel caso di tipi compatibile è una

- Conversione implicita coercizione, (coercion): il compilatore, inserisce la conversione, nessuna traccia nel testo del programma

Vedremo anche meccanismi di:

- Conversione esplicita, o cast, quando la conversione è implementata da una funzione, inserita esplicitamente nel testo programma.

Coercizione di tipo

L'implementazione deve fare qualcosa.

Tre possibilità, i tipi sono diversi ma:

- con stessi valori e stessa rappresentazione.

  esempio: tipi strutturalmente uguali, nomi diversi

  - il compilatore controlla i tipi, non genera codice di conversione
- valori diversi, ma stessa rappresentazione sui valori comuni.

  Esempio: intervalli e interi

  - codice per controllo di tipo dinamico,
    non sempre inserito
- valori e rappresentazione diversi. Esempio: interi e floating-point, oppure `int` e `long`

  - codice per la conversione

Cast

Il programmatore inserisce esplicite funzione che operano conversioni di tipo

- sintassi ad hoc per specificare che un valore di un tipo deve essere convertito in un altro tipo.

```
s = (S) t
r = (float) n;
n = (int) r;
```

- anche qui, a seconda dei casi, necessario codice macchina di conversione,
- si può sempre inserire esplicitamente un cast laddove esiste una compatibilità (utile per documentazione)
- linguaggi moderni tendono a favorire i cast rispetto coercizioni (comportamento più prevedibile)
- non ogni conversione esplicita consentita
  - solo i casi in cui linguaggio dispone funzione conversione.

Polimorfismo

Una stesse espressione può assumere tipi diversi:

- distinguiamo tra varie forme di polimorfismo:
- polimorfismo ad hoc, o overloading
- polimorfismo universale:
  - polimorfismo parametrico (esplicito e implicito)
  - polimorfismo di sottotipo

Polimorfismo ad hoc: overloading

Uno stesso simbolo denota significati diversi:

```
3 + 5
4.5 + 5.3
4.5 + 5
```

- Il compilatore traduce + in modi diversi
- spesso risolto a tempo di compilazione, quando dipende dal contesto
  - dopo l'inferenza dei tipi
- Java supporta questa forma di polimorfismo
  - nelle sottoclassi ridefinisco i metodi definiti nella classe principale,
  - in base al tipo dell'oggetto si decide il metodo da applicare.

Overloading

In molti linguaggi posso definire una stessa funzione (metodo)

- più volte
- con tipi diversi

Il linguaggio sceglie quale definizione usare in base a:

- numero di argomenti (Erlang)
- tipo, e numero, degli argomenti (C++, Java)
- tipo del risultato (Ada)

Polimorfismo parametrico

Un valore funzione ha polimorfismo universale parametrico quando

- ha un'infinità di tipi diversi,
- ottenuti per istanziazione da un unico schema di tipo generale.

Una funzione polimorfa è costituita da un unico codice che si applica uniformemente a tutte le istanze del suo tipo generale

```
identity(x) = x; :  <T> -> <T>          ∀ T. T -> T

reverse(v) = ....; :  <T>[] -> void      ∀ T. T[] -> void
```

`T` è una variabile di tipo

Polimorfismo in Scheme.

Grazie ai tipi dinamici, Scheme è un linguaggio naturalmente polimorfo

- Funzioni come:
  - (map f list) applica la funzione f a tutti gli elementi di una lista

    ```
    (T -> S) -> (list T) -> (list S)
    ```

  - (select test list) selezione gli elementi di list su cui test ritorna true

    ```
    (T -> Bool) -> (list T) -> (list T)
    ```

  sono naturalmente definibili.

Problema, definire un sistema di tipi statico che permetta il polimorfismo

Polimorfismo parametrico esplicito

In C++: function template (simile ai generics di Java)

- una funzione swap che scambia due interi

```
void swap (int& x, int& y){
int tmp = x; x=y; y=tmp;}
```

- una template swap che scambia due dati qualunque

```
template <typename T>          //T è una sorta di parametro
void swap (T& x, T& y){
T tmp = x; x=y; y=tmp;}
```

- istanziazione automatica

```
int i,j;     swap(i,j); //T diventa int a link-time
float r,s;   swap(r,s); //T diventa float a link time
String v,w;  swap(v,w); //T diventa String a link time
```

Generics in Java

```
    public static < E > void printArray( E[] inputArray ) {
        // Display array elements
        for(E element : inputArray) {
            System.out.printf("%s ", element);
        }
    }
    public static void main(String args[]) {
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        printArray(intArray);   // pass an Integer array
        printArray(charArray);   // pass a Character array
    }
}
```

l'uso di `Integer` e `Character`, al posto di `int`, `char` permette di non dover replicare il codice.

Implementazione

- Istanziazione automatica (C++)
  più istanze del codice generico
  una per ogni particolare tipo su cui viene chiamato
- Un unica istanza del codice generico (Java, ML, . . .) stesso codice macchina funziona su più tipi diversi
  possibile se
  - tutte le variabili memorizzate alla stesso modo
  - modello per riferimento (ogni variabile è un puntatore al dato)
  - accesso più lento ma codice universale

Polimorfismo parametrico in ML (implicito)

La funzione swap in ML:

```
swap(x,y) = let val tmp = !x in
              x = !y; y = tmp end;
val swap = fn : 'a ref * 'a ref -> unit
```

- ML non necessita definizione di tipo:
  - inferenza automatica di tipo a tempo di compilazione
  - si determina, per ogni funzione, il tipo più generale che la descrive
- ML, come Java, non necessita la replicazione del codice
  - si accede ai valori tramite riferimenti (indirizzi di memoria)
  - il codice manipola riferimenti, indipendenti dal tipo di oggetto puntato.

Generalizzazione del polimorfismo parametrico

Nella sua formulazione iniziale il polimorfismo parametrico ha limitazioni

Es.

```
void quickSort( E[] inputArray)
```

- non posso applicare `quickSort` a un vettore di elementi non confrontabili
- devo chiedere l'esistenza di un operazione confronto
- diverse soluzioni
  - passo la funzione confronto come parametro

  ```
  void quickSort( E[] inputArray, (E*E)->Bool compare)
  ```

  prolisso, devo esplicitamente passare tutte le funzioni ausiliarie
  - linguaggi ad oggetti: chiedo che `E` contenga un metodo confronto (istanza di una Interface)
  - linguaggi funzionali, chiedo che sugli elementi di `E` possa essere applicato una funzione confronto (<) (istanza di una Type Class)

Polimorfismo di sottotipo

- Usato nei linguaggi ad oggetti:
- può assumere diverse forme a seconda del linguaggio di programmazione
- si basa su una relazione di sottotipo: `T < S` (`T` sottotipo di `S`)
  - un oggetto di tipo `T`, ha tutte le proprietà (campi, metodi) di un oggetto di tipo `S`
  - `T` compatibile con `S`
- in ogni contesto, funzione, che accetta un oggetto di tipo `T` posso inserire, passare, un oggetto di tipo `S`

Polimorfismo di sottotipo e polimorfismo parametrico

- Per una maggiore espressività posso combinare polimorfismo di sottotipo con quello parametrico
- esempio: funzione `select` dato un vettore di oggetti restituisce l'oggetto massimo
  - con solo polimorfismo di sottotipo:
    `select: D[] -> D`
    posso applicare `select` ad vettore sottotipo di `D` ma all'elemento restituito viene assegnato tipo `D`
  - combinando i polimorfismi:
    `select: ∀ T < D. <T>[ ] -> <T>`
    descrivo meglio il comportamento di select,
    informazioni in più sul tipo risultato

In Java

```
public <T extends D> T select (T[] vector) {

}

public <T implements D> T select (T[] vector) {

}
```

Java usa anche la nozione di `interface` e `implements` come meccanismo
per estendere il polimorfismo
(`implements` è una relazione più generale di subclass `extends`)

Polimorfismo

Problema complesso, definire un sistema di tipi:

- generale: permette di dare tipo a molti programmi
- sicuro: individua gli errori
- type checking efficiente
- semplice: comprensibile dal programmatore

Conseguenze:

- Vasta letteratura
- Tante implementazioni diverse
- In evoluzione

Astrazione sui dati

Tipi di dati astratti

Varie forme di astrazione

Astrazione nozione ricorrente in questo corso

- ignorare i dettagli per
  - mettere in luce gli aspetti significativi
  - gestire la complessità
- Astrazione sul controllo
  - Nasconde la realizzazione nel corpo di procedure
- Astrazione sui dati
  - Nasconde decisioni sulla rappresentazione delle strutture dati e sull'implementazione delle operazioni
  - Nasconde rappresentazione ma anche codice
  - Esempio: una coda FIFO realizzata mediante
    - una lista
    - un vettore

Le parti di un astrazione

- Componente: nome dell'oggetto che vogliamo astrarre,
  - es.: struttura dati, funzione, modulo
- Interfaccia
  - modi per interagire, dall'esterno, con il componente
- Specifica (formale o informale)
  - descrizione del comportamento atteso,
  - proprietà osservabili attraverso l'interfaccia
- Implementazione
  - organizzazione dei dati, codice delle funzioni
  - definiti all'interno del componente,
  - dovrebbero essere invisibili all'esterno
    - dall'interfaccia non posso accedere all'implementazione
  - protetta da una capsula che la isola

Esempio di astrazione sui dati: coda di priorità

- Componente
  - Coda a 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 max priorità e la coda degli elementi rimanenti

Esempio di astrazione sui dati: coda di priorità

- Implementazione
  varie alternative possibili
  - un albero binario di ricerca
  - un vettore parzialmente ordinato

Secondo esempio: gli Interi

- Componente
  - tipo di dato integer
- Interfaccia
  - costanti, operazioni aritmetiche
- Specifica
  - le leggi dell'aritmetica
  - range di numeri rappresentabili
- Implementazioni
  - hardware: complemento a 2, complemento a 1, lunghezza fissa
  - software: interi di dimensione arbitraria
- Tipo di dato astratto?
  - C: no, posso accedere all'implementazione (attraverso le operazioni booleane)
  - Scheme: si, sugli interi solo operazioni aritmetiche

Tipi di dato astratti (Abstract Data Type)

- Le idee precedenti le posso applicare in molti ambiti
  - es. algoritmo di ordinamento,
    - specifico l'interfaccia oggetti su cui opera
    - nascondo l'implementazione
- applicate alle strutture dati definiscono gli ADT
- idee nate negli anni '70
- riassumendo:
  - Separa l'interfaccia dall'implementazione
  - un ulteriore esempio:
    - Tipo di dato `Set` con funzioni: `empty`, `insert`, `union`, `is_member?`, ...
    - `Set` implementato come: vettore, lista concatenata ...
- Usa meccanismi di visibilità, controlli di tipo per garantire la separazione

ADT per una pila di interi

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int P[100];
                        int n;
                        int top;
   }
   signature
      Int_Stack crea_pila();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack crea_pila(){
         Int_Stack s = new Int_Stack();
         s.n = 0;
         s.top = 0;
         return s;
      }
      Int_Stack push(Int_Stack s, int k){
         if (s.n == 100) errore;
         s.n = s.n + 1;
         s.P[s.top] = k;
         s.top = s.top + 1;
         return s;
      }
      int top(Int_Stack s){
         return s.P[s.top];
      }
      Int_Stack pop(Int_Stack s){
         s.n = s.n - 1;
         s.top = s.top - 1;
```

Un'altra implementazione per una pila di interi

```
abstype Int_Stack{
   type Int_Stack = struct{
                        int info;
                        Int_stack next;
   }
   signature
      Int_Stack crea_pila();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack crea_pila(){
         return null;
      }
      Int_Stack push(Int_Stack s, int k){
         Int_Stack tmp = new Int_Stack(); // nuovo elemento
         tmp.info = k;
         tmp.next = s;                    // concatenalo
         return tmp;
      }
      int top(Int_Stack s){
         return s.info;
      }
      Int_Stack pop(Int_Stack s){
         return s.next;
      }
```

Principio di incapsulamento

- Indipendenza dalla rappresentazione
  - due implementazioni corrette di un tipo (astratto)
    non sono distinguibili dal contesto (restante parte del programma)
- Le implementazioni sono modificabili senza con ciò interferire col contesto
  - i contesti non ha alcun modo per accedere all'implementazione
  - le interfaccia si simulano

Un linguaggio fornisce gli ADT se possiede meccanismi per realizzare questo "nascondimento" dell'informazione (information hiding)

Classi e oggetti.

Linguaggi a oggetti forniscono gli ADT:

- Il principio dell'incapsulamento (information hiding) è uno delle idea (ma non l'unica) alla base della programmazione OO.
- Attraverso le classi posso implementare un ADT.
  - variabili di istanza (rappresentazione interna) vengono tutte dichiarate private,
  - solo i metodi che implementano l'interfaccia sono pubblici,
  - rispetto ad altre implementazione degli ADT una diversa sintassi per chiamare i metodi
    - `push(pila, elemento)`
    - `pila.push(elemento)`

Moduli

- Costrutto generale per lo information hiding
  - disponibile in linguaggi imperativi, funzionali, ecc.
  - uso banale: evitare conflitti di nomi
- Supporta la programmazione in the large.
  - moduli distribuiti su file diversi
  - compilazione separata
- interfaccia, più sofisticata
- dentro il modulo:
  - definisco in nomi visibili all'esterno, alcuni in maniera ristretta
    - variabili in solo lettura
    - tipi esportati in maniera opaca, usabile 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

Moduli

- In molti linguaggi, con nomi diversi:
  - module: in Haskell, Modula, uno dei primi linguaggi ad usarli, . . .
  - package: in Ada, Java, Perl, . . .
  - structure: in ML, . . .

Moduli in Haskell

Definizione di modulo

```
module BinarySearchTree (Bst (Null, Bst), insert, empty, in,
  where
data Bst = ...
insert = ...
empty  = ...
...
```

- la lista `(Bst (Null, Bst), insert, empty)` definisce in nomi esportati
- definizione che non appaiono nella lista non visibili all'esterno
- scelta migliore se voglio implementare un ADT
  - `Bst(Null)` costruttore `Bst` non visibile all'esterno

Moduli in Haskell

Per importare

```
module Main (main)
  where
import  BinarySearchTree (Bst (Null), insert, in)
```

- le dichiarazione di `import` vanno messe in testa
- posso selezionare i nomi da importare

Paradigma a oggetti

Programmazione orientata agli oggetti

Programmazione orientata agli oggetti.

Paradigma di programmazione

- programmazione imperativa, ma non solo (Scala, OCaml)
- concetto di oggetto (campi e metodi)
- usato nei linguaggi di programmazione più diffusi (Java, Python, C++)
- declinato in vari modi:
  - puro: tutti i dati sono oggetti (Smalltalk, Ruby)
  - oggetti come meccanismo ausiliario (Ada)

Caratteristiche tipiche

Si estende il meccanismo dei tipi di dati astratti che permette

- information hiding e incapsulamento
  - nascondo la rappresentazione interna di un dato
  - accedo 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

Esempio

Definisco il tipo di dato contatore

```
abstype Counter {
    type Counter = int;
    signaturue
        void reset (Counter c);
        int get (Counter c);
        void inc (Counter c);
    operartions
        void reset (Counter c){
            c = 0
        }
        void get (Counter c){
            return c
        }
        ...
```

Estensione del tipo

Per estendere `Counter` devo riscrivere il codice,
nessuna correlazione di tipo tra `NewCounter` e `Counter`

```
abstype NewCounter {
    type NewCounter = struct{
                          int count
                          int resets = 0;
    signaturue
        void reset (NewCounter c);
        int get (NewCounter c);
        void inc (NewCounter c);
    operartions
        void reset (NewCounter c){
            c.count = 0;
            c.resets = c.resets + 1;
        }
        void get (NewCounter c){
            return c.counts
```

Obiettivi, linguaggio OO

- Estendere un tipo di dato astratto (`NewCounter`),
  potendo ereditare (parte del codice) (di `Counter`)
- Poter utilizzare il tipo di dato esteso (`NewCounter`)
  in contesti che accettano il tipo di dato originale (`Counter`);
  compatibilità tra tipi
- Selezione dinamica dei metodi:
  procedure che usino in maniera uniforme `Counter` e `NewCounter`
  devo decidere, a tempo di esecuzione,
  quale implementazione del metodo `reset` utilizzare.

Oggetti

- Un oggetto è una capsula (record, struct) che contiene:
  - `campi`: dati,
  - `metodi`: procedure associate ai dati, formalmente memorizzate insieme ad essi
- Un programma orientato agli oggetti:
  - invocare un metodo su un oggetti
  - un oggetto risponde, eseguendo la procedura associata,

Classi

Definiscono la struttura degli oggetti

Una classe definisce un insieme di oggetti specificando:

- un insieme di campi con relativo tipo
- un insieme di metodi con relativo codice
- visibilità dei campi e metodi
- costruttori degli oggetti

Oggetti creati dinamicamente per istanziazione di una classe, mediante costruttori

```
class Counter {
    private int count;
    public  void reset (){
            count = 0
        }
    public  void get (){
            return count;
```

Classi

Diversi aspetti nella definizione di una classe:

- information hiding e incapsulamento
  definisco cos'è visibile all'esterno e cosa no
  `private`, `public`, `protected`
- astrazione sui dati e sul controllo
  assegno un nome alla classe, ai campi, ai metodi
- estensibilità e riuso del codice
  nel caso definisca delle sottoclassi,

Linguaggi prototype based

- I linguaggi OO si distinguono in:
  - Class based: più comuni
  - Prototype based (object-based):
    - oggetti come record con metodi.
    - non sono necessari, pattern predefiniti
    - presente sono nei linguaggi con sistema di tipo dinamico
- JavaScript il principale esponente dei prototype based
  - nessuna parentela con Java,
  - linguaggi di default per introdurre codice in pagine html,

JavaScript

- Oggetti come record (oggetti, e record, tipi di dati esprimibili)

```
var person = {firstName:"John", lastName:"Doe", age:50,
    name = function () {
    return this.firstName + " " + this.lastName;
    }
};
```

- oggetti estensibili: posso aggiungere nuovi dati o metodi

```
person.eyeColor = blue;
```

- linguaggio con tipi dinamici, controllo dei tipi a run-time
- nessuna distinzione tra campi e metodi,
  - formalmente ogni oggetto contiene i propri campi e metodi
  - che possono essere ridefiniti
  - implementazione: metodi descritti da puntatori a codice condiviso.

JavaScript: prototype

- 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);
```

- Possibilità di estendere i costruttori

```
Person.prototype.altName = function() {
    return this.lastName + " " + this.firstName;
```

JavaScript: costruttori predefiniti

- Costruttori predefiniti, creano oggetti standard con un ricco insieme di metodi

```
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
```

JavaScript: meccanismo delle closure

```
function counter() {
    var count = 0;
    return function() {
        return ++count;
    };
}
var closure = counter();
closure(); // restituisce 1
closure(); // restituisce 2
```

- `counter` costruisce una funzione e la restituisce
- `closure` fa riferimento all'ambiente di `counter` in cui è stata definita
- `count` definita in questo ambiente viene preservata (e incrementata)
- information hiding, la variabile `count` accessibile solo attraverso closure

Information hiding non disponibile in forma esplicita

Identificatori `this` e `self`

All'interno dei metodi di un oggetto spesso si fa riferimento ad altri metodi o campi dell'oggetto stesso

questo può essere fatto in maniera:

- esplicita, con le parole chiave `this` o `self`
  `this.field`, `self.field` (a seconda del linguaggio)
  per affermare che faccio riferimento ad un campo interno
- implicita: nel metodo appare solo `field`
  implicitamente si intende il campo `field` dell'oggetto corrente

In Java:

- riferimento implicito
- possibile usare la keyword `this`
  utile in caso di mascheramento del campo o del metodo

Modello riferimento-variabile

Normalmente oggetti usano il modello a riferimento

- variabile contiene un puntatore all'oggetto
- oggetti memorizzati nell'heap
- creazione esplicita degli oggetti mediante funzioni di costruzione, allocazione dello spazio
- meccanismi di garbage collection per riuso della memoria

Modello a valore

- variabile contiene i campi dell'oggetto
- può essere memorizzata nello stack
- creazione implicita degli oggetti (chiamata di procedure)
- usata in C++,
- accesso diretto ai dati e maggiore efficienza

Incapsulamento

Le classi garantiscono incapsulamento

- opportuni modificatori determinano se campi e metodi sono
  - pubblici, `public`
  - privati, `private`
  - protetti, `protected`
- la parte pubblica definisce l' "interfaccia" della classe
- la parte privata è usata nell'implementazione
- protetti definisce una parte privata, visibile nelle sottoclassi

Linguaggi diversi offrono funzionalità diverse

- `protected` può aver diversi significati
  - visibile nelle sottoclassi
  - visibile nel package

Static field, static method

Oltre alle etichette di visibilità: `public`...

i campi e metodi possono essere etichettati come `static`

- un campo `static`
  - variabile, in singola copia, condivisa tra tutti gli oggetti di una classe
  - le modifiche fatte da un oggetto sono visibili agli alti
- metodo `static`
  - può accedere solo alle variabili `static`
  - non accede alla variabile dei singoli oggetti (`instance`)
  - non usa l'identificatore `this`, nemmeno in maniera implicita

In altri ambiti:

- in una procedura una variabile locale `static`
  - variabile memorizzata nella parte statica della memoria
  - preserva il suo valore tra una chiamata e l'altra
  - nel caso di procedure ricorsive presente in singola copia

Costruttori

Nel definire una classe devo definire una serie di costruttori:

- inizializzano i campi del nuovo oggetto creato
- possibili più costruttori, selezione
  - per numero e tipo degli argomenti,
    se il nome del costruttore coincide con quello della classe (Java)
  - per nome
- nel caso sottoclassi:
  - eseguo prima il costruttore della super-classe
    eventualmente con chiamata esplicita
  - e poi quello della sottoclasse
- chiamata con sintassi ad hoc, `new`

Distruttori

Possibile definire distruttori

- metodi da eseguire prima della deallocazione di un oggetto

Recupero spazio di memoria heap

- esplicita
- garbage collector
  meno efficiente ma più sicura

Sottoclassi

Posso estendere classi con nuovi metodi o campi,

```
class NamedCounter extends Counter{
    private String name;
    public void set_name(String n){
        name = n;
    }
    public String get_name()
        return name
    }
}
```

`NamedCounter` è una sottoclasse (o classe derivata) di `Counter`:

- ogni istanza di `NamedCounter` risponde a tutti i metodi di `Counter`
- il tipo `NamedCounter` è compatibile con `Counter`

Ridefinizione di metodo (overriding)

```
class NewCounter extends Counter{
    private int num_reset = 0
    public void reset (){
        count = 0;
        count_reset++;
    }
    public int get_num_reset() {
        return num_reset;
    }
}
```

- `NewCounter` contemporaneamente:
  - estende l'interfaccia di `Counter` con nuovi campi
  - ridefinisce il metodo `reset`
  - il metodo `reset` ha due diverse definizioni in `Counter` e `NewCounter`

Ridefinizione dei campi porta al shadowing

Ruby, getter and setter

Ruby vuol essere un linguaggio ad oggetti puro

- tutti i dati sono oggetti
- posso accedere ai campi interni solo attraverso i metodi

```
class Temperatura
    def celsius
        return @celsius
    end
    def celsius=(temp)
        @celsius = temp
    end
    def farenheit
        return @celsius * 9/5 + 32
    end
    def farenheit=(temp)
        @celsius = (temp - 32) * 5/9
    end
```

Ruby, zucchero sintattico

Le uniche operazioni ammesse sono chiamate di metodi

si usa la sintassi standard attraverso zucchero sintattico

```
Temperatura.celsius = 0  # sta per Temperatura.celsius=(0)
y = Temperatura.farenheit
y + 5  # sta per per y.+(5)
```

Sottoclassi e meccanismi di hiding

Una sottoclasse può modificare la visibilità della super-classe

- C++ nascondere metodi pubblici nella super-classe

```
class derived : protected base { ... }
class derived2 : private base { ... }
```

- Eiffel modifica in maniera arbitraria
- Java, C# non modifica la visibilità della superclasse

Ereditarietà

Attraverso la definizione di sottoclassi introduco un meccanismo di ereditarietà per il riutilizzo e la condivisione del codice

- in un esempio precedente, qui riportato
  - `NamedCounter` eredita da `Counter` i metodi (e i campi) non ridefiniti
- l'ereditarietà permette condivisione del codice in maniere modificabile, estendibile:
  - ogni modifica all'implementazione di un metodo in una classe automaticamente disponibile a tutte le sottoclassi

```
class NamedCounter extends Counter{
    private String name;
    public void set_name(String n){
        name = n;
    }
    public String get_name()
        return name
    }
}
```

Ereditarietà singola e multipla

- Singola (Java)
  - ogni classe `extends` una sola super-classe
  - eredita al più da una sola super-classe immediata
- Multipla (C++)
  - una classe può ereditare da più di una super-classe immediata
- Ereditarietà multipla
  - più flessibile il riuso del codice
  - fonte di problemi
    - name clash: se lo stesso metodo definito nelle due super-classi, quale metodo ereditare?

Name-clash nell'ereditarietà multipla

```
class Top{
    int w;
    int f(){
        return w;
    }
}
class A extending Top{
    int x;
    int g(){
        return w+x;
    }
}
class B extending Top{
    int y;
    int f(){
        return w+y;
    }
    int k(){
        return y;
    }
}
class Bottom extending A,B{
    int z;
    int h(){
        return z;
    }
}
```

Name-clash nell'ereditarietà multipla

Nell'esempio precedente quale metodo `f` eredita `Bottom`

- quello ereditato nella classe `A`
- quello ridefinito della classe `B`

Possibile soluzioni:

- impedisco ogni conflitto tra metodi
- impongo che ogni conflitto venga risolto nel codice, specificando da quale classe ereditare il metodo (C++)
- definisco una criterio di scelta dei metodi presenti in più classi (es. quelle che appare per ultimo nel codice)

Non esiste una soluzione ottimale,

- per ciascuna delle precedenti si possono fare esempi in cui risulta innaturale.

Mixin

Soluzione sofisticata, alternative all'ereditarietà multipla,

- permette un riutilizzo del codice, evitando le ambiguità dell'ereditarietà multipla
- usa classi astratte
- permette di incorporare metodi da una classe senza diventarne sottoclasse,
- quindi metodi generici usabili su più classi
- realizzato
  - in varie forme nei diversi linguaggi
  - esempio: classi astratte con codice di default per i metodi

Polimorfismo di sottotipo

Tipico dei linguaggi OO

Un tipo `T` è sottotipo di `S` (`T <: S`) quando:

- `T` compatibile con `S`,
- nei contesti di tipo `S` posso inserire un oggetto `T`

Definendo una relazioni di sottotipo posso definire funzione polimorfe

se

- `T <: S`
- `f : S -> R`

allora automaticamente:

- `f : T -> R`
- posso assegnare a `f` tipi diversi
- a `f` possa applicare valori di diverso tipo

Sottoclassi e sottotipi

Spesso

- se la classe `T` e sottoclasse di `S`
- allora la classe `T` e sottotipo di `S`

Infatti

- se `T` possiede tutti i metodi di `S`
- posso usare oggetti `T` in tutti i contesti `S`

Eccezioni

- C++, Eiffel
  - perché nelle sottoclassi posso rendere privati metodi della pubblici della superclasse
  - sottotipo solo se questo non accade
- in C++

```
class B : public A {...  \\ B sottotipo di A
class C : A {...  \\ C non sottotipo di A ma eredita
```

- classi con binary methods
  - una classe `A` contiene un metodo che ha un parametro di tipo `A` esempio test di uguaglianza, un oggetto controlla se un altro oggetto è identico a lui
  - i binary methods sono incompatibili con la relazione di sottotipo

Ordine tra sottoclassi

- Sottoclasse relazione d'ordine:
  `A <: B`, `B <: C` implica `A <: C`
- In alcuni linguaggi esiste:
  `Object` classe più generale di tutti,
  tutte le classi sono sottoclassi di `Object`
  contiene metodi comuni a tutte le classi (creazione, uguaglianza)

Metodi astratti, classi astratte, interface

Utilizzati anche 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 è possibile creare oggetti di classe astratte
  - servono come modello per classi concrete (tutti i metodi istanziati)
- Interface: classi
  - con solo metodi
  - nessun campo

Vantaggi delle `interface`:

- definire la struttura di una classe senza specificare il codice
  - si definiscono tipo dei metodi, ma non il codice dei metodi
- semplici dichiarazioni di tipo
- non dovendo ereditare il codice, nessuna controindicazione ad implementare (essere sottotipo) di più di un interface

Esempio

```
abstract class A{
    public int f();
}

abstract class B{
    public int g();
}

class C extending A,B{
    private x = 0;
    public int f(){
        return x;
    }
    public int g(){
        return x+1;
    }
}
```

Classi astratte e relazione di sottotipo

Se una classe concreta `C` implementa un interface (classe astratta) `A`

- `C` sottotipo di `A`
- oggetti di `C` rispettano le specifiche di `A`

Un tipo può avere più di un sovratipo immediato

- `C` sottotipo sia di `A` che di `B`
- per le interface non esisto i problemi di name-clash dell'ereditarietà multipla
  - nessun codice,
  - tipi forzatamente identici
- la gerarchia dei sottotipi non è un albero,

Ereditarietà e sottotipo

- Sottotipo
  - meccanismo che permette di usare un oggetto in un altro contesto
  - è una relazione tra le interfacce di due classi
    interfaccia insieme dei campi e metodi pubblici di una classe
- Ereditarietà
  - meccanismo per riusare il codice dei metodi,
  - è una relazione tra le implementazioni di due classi.

Ereditarietà e sottotipo

Due concetti formalmente indipendenti

- ma spesso collegati nei linguaggi OO, le sottoclassi ho
  - l'ereditarietà dei metodi implica
  - ed una relazione di sottotipo.
- sia C++ che Java hanno costrutti che introducono contemporaneamente entrambe le relazioni tra due classi
  - in Java
    `extends` introduce sia ereditarietà e sottotipo
    `implements` solo sottotipo

Java ereditarietà singola estensione multipla

```
interface A{
    int f();
}
interface B{
    int g();
}
class C{
    int x;
    int h(){
        return x+2}
}
class D extends C implements A, B{
    int f(){ return x };
    int g(){ return h() };
```

Selezione dinamica dei metodi

- Un metodo `m` viene invocato su un parametro `count` di tipo oggetto :
  - nelle diverse invocazioni `count` può essere istanziato con sottotipi diversi
    - con diverse versioni del metodo `m`
  - come avviene la scelta della versione di `m` da invocare?
- Esempio

```
Counter V[100];
...
for (Counter count : V){count.reset()}
```

- `V` può contenere elementi di tipo `Counter` che `NewCounter`
- Selezione dinamica (Java)
  - a tempo d'esecuzione, in funzione del tipo dell'oggetto che riceve il messaggio
  - selezione determinata dal tipo dell'oggetto,

Selezione statica

- Selezione statica (C++),
  - dipende dal tipo del parametro
  - più efficiente
  - ma si introducono meccanismi per permettere la selezione dinamica metodi `virtual`

Polimorfismo di sottotipo

- La relazione di sottotipo utile per implementare polimorfismo di sottotipo

```
interface Printable{void print() {...}; };
void f (Printable x) {...};

// f : (T <: Printable) -> void
```

definisco una funzione che può essere chiamata su una qualsiasi istanziazione di `A`

Polimorfismo parametrico

- uso parametri di tipo, <T>, per definire classi e funzioni parametriche

```
class Tree<T>{ ...};
public static <T> int height (Tree<T> t) {...
    };
Tree<int> tint = new Tree<int>();
...
n = height<int> tint;
m = height tint;

// height: forall T . Tree(T) -> int
```

Combinazione dei due polimorfismi

Definisco funzioni parametriche con vincoli sul parametro:

```
interface Printable{ void print() }
public static <T extends Printable> void printAll (Tree<T> tTr

// printAll: forall T <: Printable . Tree(T) -> void
```

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 oggetto in un contesto è lecito se l’oggetto possiede tutti i metodi necessari
  - indipendentemente dal suo tipo
  - usato nei linguaggi con un sistema di tipo dinamico
  - controllo a tempo di esecuzione che se un oggetto riceve un messaggio `m`,
    `m` appartenga ai suoi metodi
- tipi di dati dinamici
  - maggiore espressività rispetto a tipi statici
  - minore efficienza, minor controllo degli errori, possibili errori di tipo nascosti

Paradigma funzionale

Linguaggio Haskell

Paradigmi imperativo e funzionale

Paradigma funzionale si contrappone
al paradigma imperativo dove:

- l'esecuzione programma comporta
  l'esecuzione di una sequenza di istruzioni
- istruzione base, modifica di una variabile, locazione di memoria

Paradigma imperativo usato nelle implementazioni hardware, linguaggi macchina

- macchina di von Neumann

Gran parte dei linguaggi ad alto livello restano vincolati al paradigma imperativo

- linguaggi di von Neumann
- linguaggi ad oggetti

Paradigma funzionale

Paradigma di programmazione:

- dichiarativo
  - il programma consiste nella definizione di una serie di funzioni e di costanti

Differenze rispetta all'imperativo;

- assenza di stato
  - non si accede direttamente alla memoria (store)
  - non esistono variabili modificabili
- diverso meccanismo di computazione
  - non si esegue una di sequenza di istruzione
  - si valuta un'espressione
    - dal punto di vista astratto: sequenza di riscritture
    - concreto: macchina a stack, attivo tanti RdA

Meccanismo di valutazione teorico - riscrittura

Data la definizione:

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

la valutazione di `fatt 3` porta a valutare nell'ordine:

```
if 3 == 0 then 1 else 3 * (fatt (3 - 1))
3 * (fatt 2)
3 * (if 2 == 0 then 1 else 2 * (fatt (2 - 1))
3 * 2 * (fatt 1)
...
```

passi di computazione:

- sostituisco una chiamata di funzione con il suo corpo istanziato
  - `fatt 2` diventa `if 2 == 0 then 1 else 2 * (fatt (2 - 1)`
- applico le funzioni base
  - `3 - 1` diventa `2`

Meccanismo di valutazione implementato

Uso degli ambienti, la valutazione di:

```
fatt 3
```

diventa la 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 per essere valutata, porta alla valutazione di

```
n == 0
```

nell'ambiente . . . ..

Meccanismo di valutazione implementato

Si crea un pila di espressioni da valutare, con relativi ambienti di valutazione.

Struttura complessa:

- macchina SECD di Landin (1964)
- base per le implementazioni attuali

Paradigma funzionale

- l'assenza di stato comporta l'assenza di cicli (`while`)
  - la valutazione di un espressione porta sempre allo stesso risultato, non ha senso ripeterla
  - cicli sostituiti da funzioni ricorsive, si valuta la stessa funzione con argomenti diversi,
- funzioni oggetti del primo ordine,
  - funzioni di ordine superiore
  - le funzioni sono argomento o risultato di altre funzioni
  - funzioni senza nome, `(lambda (x) (+ x 1))` espressioni che rappresentano una funzione
- polimorfismo
  - riuso del codice attraverso funzioni che si adattano a tipi di dato diversi
    - type checking dinamico: Scheme
    - type checking statico: Haskell, schemi di tipo, type checking sofisticato

Vantaggi del paradigma funzionale

- L'assenza di stato semplifica il ragionamento
  - non è possibile l'aliasing
- Un esempio di errore creato dall'aliasing, modello per riferimento, errore che era presente in una libreria Java)
  - Classe per l'accesso protetto alle risorse
  - accedo alla risorsa con il metodo `useTheResource`
  - il metodo controlla che solo gli utenti nella lista `allowedUsers` possano accedere
  - è possibile ricevere la lista `allowedUsers` per controllare la propria priorità.

Codice

```
class ProtectedResource {
    private Resource theResource = ...;
    private String[] allowedUsers = ...;
    public String[] getAllowedUsers() {
        return allowedUsers;
    }
    public String currentUser() { ...}
    public void useTheResource() {
        for(int i=0; i < allowedUsers.length; i++) {
        if(currentUser().equals(allowedUsers[i])) {
            ...// access allowed: use it
        return; }}
        throw new IllegalAccessException();}}
```

Patch

Far restituire da `getAllowedUsers()` una copia della lista interna

```
public String[] getAllowedUsers() {
    String[] copy = new String[allowedUsers.length];
    for(int i=0; i < allowedUsers.length; i++)
        copy[i] = allowedUsers[i];
    return copy;
}
```

In programmazione funzionale

- il problema non si pone
- accedo sempre ad una copia

Haskell (e Scheme)

Presentazione del paradigma attraverso un esempio concreto

- Haskell

Confronto con Scheme, funzionale ma con scelte diametralmente opposte

- Origini
  - Scheme: uno dei tanti dialetti Lisp (1958, Johh McCarthy)
    come Clojure, Common Lisp, Emacs Lisp, Standard Lisp, Racket
  - Haskell: ML (1973, David Milner, Edimburgo),
    come Standard ML, Caml, OCaml,

Haskell (e Scheme)

- Sintassi
  - Scheme: pochi costrutti, semplici (a scapito delle concisione del codice)
  - Haskell: non molti costrutti base, molto zucchero sintattico, programmi molto sintetici,
- Esempio, crivello di Eratostene in Haskell

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Haskell (e Scheme)

Sistema di tipi

- Scheme: controllo di tipo dinamico
- Haskell: controllo statico
  - inferenza di tipo
- entrambi fortemente tipati

Haskell (e Scheme)

Meccanismo di valutazione

- Scheme: linguaggio eager, call-by-value
  - gli argomenti di una funzione vengono valutati prima di essere passati nel corpo,
  - valuto tutto appena possibile
- Haskell: lazy, call-by-need (passaggio per nome)
  - gli argomenti di una funzione vengono passati cosi come stanno
  - valuto un argomento se proprio non posso farne a meno
  - posso gestire strutture dati infiniti, stream

Confronto eager – lazy

- esistono espressioni che convergono con la valutazione lazy, e divergano nella eager
- ma non il viceversa
- sui tipi semplici, se convergono, convergono sullo stesso valore
- valutazione eager più semplice da implementare
- valutazione lazy, implementata in maniere semplice,
  porta a valutare lo stesso argomento molto volte con perdite di efficienza

Valutazione lazy.

- Vantaggi
  - evito lavoro inutile `fst (2, (fact 1000))`
  - più programmi terminano `fst (2, (diverge 0))` `diverge  x = diverge x`
  - più programmi non generano errori `fst (2, error)`
- Svantaggi
  - potenziale ripetizione delle 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
    - side-effect modifico 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

Valori

In un linguaggio funzionale, soprattutto eager, è importante definire

- l'insieme dei valori le espressioni che si considerano essere completamente valutate
  - le costanti numeriche `1, 2, ...` sono valori,
    tutti le alte espressioni di tipo intero no
  - coppia `(e1, e2)` è un valore se e solo se le sue componenti, `e1`, `e2`, sono valori
  - le lambda astrazioni sono valori
    `(lambda (x) (+ 3 1))` funzione costante
    `(lambda () (+ 3 1))` funzione senza argomento, thunk
    - meccanismo per sospendere la valutazione
  - ...

Haskell (e Scheme)

- Purezza
  - Scheme: non e' un linguaggi funzionale puro, esistono effetti collaterali, posso modificare lo stato

```
(define b 1)
(define succ (lambda (x) (+ x b)))
(set! b 5)
```

  - Haskell puramente funzionale
    come (quasi) tutti i linguaggi lazy - la gestione dei side-effect nei linguaggi lazy più complessa,
    non è chiaro in quale istante un'espressione è valutata e quante volte
    problematico se la valutazione ha effetti collaterali

Haskell (e Scheme)

- in generale
  - in Haskell molti dei concetti di Scheme, ma sotto una veste diversa
  - Haskell più sofisticato, nuovi costrutti, meccanismi per definire programmi

Documentazione

- Sito haskell.org, dispone di tutte le informazioni
  - tutorial: A gentle introduction to Haskell 98 ,
    qualche discrepanza con la versione attuale Haskell2010,
    (sistema di tipi più ricco)
    non completamente retrocompatibile
  - download: ghc: Glasgow Haskell Compiler (linguaggio compilato)
    - ghci: un interprete Haskell, esecuzioni interattiva.

Valori e tipi

Tipi scalari:

- 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
```

sistemi di tipi più complesso rispetto a Haskell98, tutorial diverse "classi" di tipi numerici.

- Overloading, ma nessuna coercizione (conversione implicita)

```
(rem 7 4) :: Integral a => a  --- generico tipo intero
1 + 2  :: Num a => a
(1 + 2) + pi      --- accettato, a (1+2) associato un tipo
(rem 7 4) + pi    --- genera errore, Integral incompatibile
```

Sistema di inferenza dei tipi:

- non serve (quasi mai) dichiarare il tipo di un identificatore
- il compilatore (interprete) lo capisce dal contesto, in maniera piuttosto complessa.
  - algoritmo di inferenza di tipi
    cerca lo schema di tipo più generale
    associabile ad un espressione
- `ghci` fornisce il tipo di un'espressione `e` con `:type e`

```
Prelude> :type (rem 7 4)
(rem 7 4) :: Integral a => a
```

Tipi scalari

- Caratteri (Unicode set)

```
'a'   :: Char
```

- Boolean

```
True  False  :: Bool
```

Tipi composti

- enuple di tipi diversi

```
('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

```
fst (2,4)
snd (2,4)
fst (2,3,5)  --- genera errore di tipo
```

- le componenti di altre enuple selezionate per pattern matching

Liste

Sequenze di elementi dello stesso tipo:

```
[1,2,4] :: Num t => [t]
1,2,pi] :: Floating t => [t]
[1,2,'a']  --- type error
['a','b','d'] == "abd"  --- :: [Char]  stringhe identifico
```

- liste formate da due costruttori

```
[] :: [a]   --- lista vuota, nil di Scheme
(:) :: a -> [a] -> [a]  --- concatenazione, infisso, cons
1:2:4:[]   --- senza parentesi,  (:) associa a destra
(1:(2:(4:[])))   ----  posso aggiungere parentesi, a diffe
1:2:4:[]  == [1,2,4] --- zuccherare sintattico
'a':'b':'c':[]  == "abc"  --- zucchero sintattico
```

Liste

- sequenze aritmetiche, presente in Python, e altri linguaggi

```
[1..9]      --- definizioni compatte
[1,3..20]   --- schema più sofisticato
            --- definisce una progessione aritmetica
```

- distruttori: `head`, `tail`, equivalente di `car`, `cdr`, ma con nomi accettabili

```
head [2,4,6]
tail [2,4,6]
head (tail [2,4,6])
```

- posso costruire liste di liste, ma tipi omogenei

```
[[1,2],[1,2,3],[1]] :: [[Integer]]
[[1,2],[1,2,3], 1 ] --- genera errore di tipo
```

Tipi definiti dall'utente

Meccanismo piuttosto sofisticato per definire un ricco insieme di tipi

- Tipi Enumerazione

```
data Colore = Rosso | Verde | Marrone
y = Rosso
```

- Haskell case sensitive,
  - nome dei tipi, costruttori, costanti: iniziano per maiuscola
  - nome variabili (normali o di tipo), funzioni: iniziano per minuscola

```
data Colore = rosso | Verde | Marrone   --- genera errore
data colore = Rosso | Verde | Marrone   --- genera errore
```

- tipo Bool enumerazione predefinito (ma ridefinibile)

```
data Bool = True | False
data NBool = True | False
```

- dichiarazioni multiple: vietate dal compilatore ma accettate dall'interprete

Tipi: record

Tipi record, con costruttori

```
data IntPoint = IPt Integer Integer
IPt 3 6
```

- definisco un tipo con più componenti, alle singole componenti accedo in base alla posizione,
- mancano le etichette sulle componenti, ma posso introdurle con sintassi alternativa
- etichetto la struttura `IPt`

Tipi: union

Tipi unione, più alternative

```
data Point23Dim = Pt2 Integer Integer
                | Pt3 Integer Integer Integer
Pt2 1 2
Pt3 1 2 3
```

- definisco un elemento che può avere 2 o 3 componenti,
- più strutture alternative
- un costruttore per ogni alternativa

Tipi parametrici

- Tipi parametrici, un tipo che dipende da un altro tipo

```
data Point a = Pt a a
```

- `a` parametro, variabile di tipo
- `Pt` costruttore generico di tanti tipi diversi
- generic (parametrized) types in Java, implementano idee simile ma con tipi espliciti

```
Pt 2 3       :: Num a => Point a
Pt 'a' 'b'   :: Point Char
```

- `Point`: type constructor
- `Pt` : data constructor

Java Generic Classes

Tipi parametrici analoghi ai tipi generici in Java

```
class Point<T> {
   private T x, y;

   public void pt(T x, T y) {
      this.x = x;
      this.y = y; }

   public T getX() {
      return x; }

   public T getY() {
      return y; } }
```

Haskell non c'è un istanziazione esplicita dei tipi parametrici

Name set

Posso anche definire:

```
data Point a = Point a a
Point 2 3 :: Point Integer
```

- uso lo stesso nome per due oggetti in categorie sintattiche differenti,
- nameset diversi, il contesto dirime l'ambiguità

Enuple

Viste come caso particolare di tipo record parametrico

```
(3, 'a', ['b']) :: Num t => (t, Char, [Char])
```

- nota virgola per separare i campi,
- parentesi tonde, obbligatorie, con significato diverso dal solito

Equivalente alla definizione

```
data (_ ,_ ,_ ) a b c = (_, _, _ ) a b c
```

- più zucchero sintattico ad hoc.

Posso definire enuple di lunghezza arbitraria

- concettualmente: tanti costruttori
  - ognuno con un numero di argomenti differenti
  - ma tutti con le stesso nome

Tipi ricorsivi (parametrici)

Definizione ricorsive sono lecite

```
data Tree a   = Leaf a | Branch (Tree a) (Tree a)
data List a   = Nil    | Cons  a (List a)

Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)) :: Tree Integer
Cons 'a' (Cons 'z' (Cons '8' Nil))   :: List In
```

Definizione di un nuovo tip T

- vengono definiti un insieme di costruttori
- costruttori: funzioni che restituiscono elementi di T.
- di ogni costruttore viene definito il dominio, la lista degli argomenti
- definizioni parametriche rispetto ad uno o più tipi `a`, `b`
- definizioni ricorsive sono lecite

Tipi di dato ricorsivi in Scheme.

Usando le liste (arbitrarie) di Scheme posso definire diversi modi per rappresentare dati di tipi ricorsivi

- un approccio canonico

```
(define  (Leaf a)  (list 'LTree a))
(define  (Branch  tl tr) (list 'BTree tl tr))
(define  (leftSon t) (car (cdr t)))
(define  (BTree-? t) (eq? (car t) 'BTree))
```

- definisco
  - i costruttori `Leaf` `Branch`
  - i distruttori `leftSon` `rightSon`
  - funzioni di controllo `BTree`
- nessuna definizione di tipo o controllo di tipo
  - il programmatore usa uno schema canonico per definire gli oggetti
  - controlli a tempo di esecuzione

Tipi predefiniti come caso particolare

Tipi di dato predefiniti sono quasi tutti un caso particolare dei tipi ricorsivi:

- Boolean: tipo di data predefinito maniera perfetta
- Enuple, Liste: predefiniti ma usano una sintassi ad hoc
- `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

- Posso dare un nome ad espressioni di tipo

```
type String = [Char]   --- predefinito
type Person = (Name,Address)
type Name   = String
data Address = None | Addr String

type IntTree = Tree Integer
type BinaryFunction a = a -> a -> a
type BinaryIntFunction = BinaryFunction Integer
```

- Equivalenza strutturale tra i tipi:
  - `[String]` e `[[Char]]` sono espressioni di tipo equivalenti.
  - come `Person -> Name` e `(Name,Address) -> Name` e `(Name,Address) -> [Char]`

Equivalenze tra tipi

Nota: tipi ricorsivi con nomi diversi, con stessa struttura, non sono equivalenti

```
data Point1 = Point1 Integer Integer
data Point2 = Point2 Integer Integer
```

Variabili e binding

Posso associare valori a variabili

```
x = 5 + 3
```

- legame immutabile, identificatori (variabili, funzioni)
  non possono essere ridefinite
  se non in ambienti locali, mascheramento
  - o usando l'interprete: ogni interazione con l'interprete genera un nuovo ambiente locale
- legame lazy, per nome
  a `x` associo l'espressione `x + 3` senza valutarla
- l'ordine non ha importanza

```
x = y + 3
y = 0
```

Variabili e binding

- possibili definizioni mutamente ricorsive

```
x = y + 3
y = x - 3
z = z + 1
```

- il programma diverge solo quando viene forzata la valutazione di `x` o `y` o `z`
  legame lazy

Ambiente locale

Le dichiarazioni

```
x = 5
succ x = x +1
```

Definiscono legame immutabile,

- gli identificatori no posso essere ridefiniti,
- validi su tutto il programma.

Utile avere ambienti locali,

- variabili locali, funzioni ausiliare, per definire altre funzioni.

Due costruttori di ambienti locali

`let`

```
y = 5
let y = a + 3
    a = 5
in y + a
```

- `let` definisce un espressione con ambiente locale
- mutua ricorsione
- nell'ambiente locale posso ridefinire delle variabili globali

`where`

```
y = 2 + z
where z = x*x
```

- usata solo all'interno di definizioni
  funzioni, valori ausiliari necessarie per definire una funzione principale
- post-fisso, mutua ricorsione

Funzioni

Oggetti del primo ordine

- lambda astrazione, per costruire funzioni nameless

  ```
  \ x -> x + 1
  ```

  il simbolo \ ricorda la lettera greca lambda λ

- definizione di funzioni (associamo un identificatore ad una funzione)
  - due alternative

```
inc x = x + 1
inc = \ x -> x + 1 --- definizione equivalente
```

Pattern matching

- definizione per casi, si applica il primo caso lecito

```
fact 0 = 1
fact n = n * fact (n -1)

fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)

fibAux 1 = (0,1)
fibAux n = (snd pair, fst pair  + snd pair)
             where pair = fibAux(n-1)
```

Currying

- da Haskell Curry,
- funzioni su più argomenti sono viste come funzioni che restituiscono funzioni
- isomorfismo tra `(A * B) -> C` e `A -> (B -> C)`
- currying significa passare dalla prima alla seconda rappresentazione dello spazio di funzioni

Esempio: definisco una funzione somma come (tre definizioni equivalenti):

```
add   = \ x -> ( \ y -> (x + y ))
add   = \ x y -> x + y
add x y = x + y
```

Currying

il currying permette

- una gestione elegante di funzione a più argomenti, non serve costruire coppie di valori
- sintetizza la definizione di alcune funzioni

```
succ = add 1
succ = \x -> add 1 x
```

- per i nostalgici, la versione uncurryed è comunque possibili

```
uncurryedAdd  :: Num a => (a, a) -> a
uncurryedAdd (x, y) = x + y
dodici = uncurryedAdd (5,7)
```

- notare che le parentesi sono quelle dei tipi enupla,
- non quelle che racchiudono i parametri di una procedura

Regole sintattiche per evitare parentesi

- per applicare una funzione e sufficiente la giustapposizione
  - `f x` l'equivalente della notazione matematica `f(x)`
  - o nella sintassi di Scheme `(f x)`
    - può essere usata in Haskell,
      posso inserire un numero arbitrario di parentesi forzano solo l'ordine di valutazione
- l'applicazione associa a sinistra
  - `add x y` abbreviazione per `(add x) y`
  - posso anche scrivere `(add x y)`, come in Scheme,
    o `((add x) y)`
- nelle espressioni di tipo, la freccia `->`, associa a destra

  ```
  add :: Integer -> Integer -> Integer
  ```

  abbreviazione per

  ```
  add :: Integer -> (Integer -> Integer)
  ```

Funzioni definite per casi (pattern matching)

- Meccanismo comodo per definizione chiare e sintetiche di funzioni
- Zucchero sintattico, si può ridurre ad altri meccanismi base, costrutto `case`

```
head (x:xs) = x
tail (_:xs) = xs
```

- wildcard _, sostituisce le variabili nei pattern,
  - rende esplicito il non uso della variabile,
  - tutti i wildcard sono considerati variabili distinte

```
empty :: [a] -> Bool
empty (_:_) = False
empty _     = True
```

Esempi

```haskell
length :: [a] -> Integer
length [] = 0
length (_:xs) = 1 + length(xs)

map :: (a->b) -> [a] -> [b]
map _ []     = []
map f (x:xs)  = f x : map f xs

squares = map (\x -> x * x) [1..10]
```

Cosa contiene `map add [1..10]`?

Alcuni funzionali standard

- curry: trasforma una funzione binaria nella sua versione curryficata

```
curry  :: ((a, b) -> c) -> a -> b -> c
curry f x y  =  f (x, y)
curry f =  \ x y ->  f (x, y)
```

- operazione inversa

Alcuni funzionali standard

- curry: trasforma una funzione binaria nella sua versione curryficata

```
curry  :: ((a, b) -> c) -> a -> b -> c
curry f x y  =  f (x, y)
curry f =  \ x y ->  f (x, y)
```

- operazione inversa

```
uncurry  :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) =  f x y
uncurry f = \ (x, y)  f x y
```

- compose, composizione di due funzioni

Alcuni funzionali standard

- curry: trasforma una funzione binaria nella sua versione curryficata

```
curry  :: ((a, b) -> c) -> a -> b -> c
curry f x y  =  f (x, y)
curry f =  \ x y ->  f (x, y)
```

- operazione inversa

```
uncurry  :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) =  f x y
uncurry f = \ (x, y)  f x y
```

- compose, composizione di due funzioni

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f ( g x )
compose f g = \ x -> f ( g x )
```

predefinita `.` in notazione infissa `f . g`

- esercizio: comprendere i tipi, aggiungere parentesi per chiarirli

Meccanismo di valutazione, passaggio dei parametri

Valutazione lazy, call-by-need

- argomenti di funzioni e valori associati a variabili, non vengono valuti, al momento del passaggio dei parametri, definizione, si passa la loro closure (espressione, ambiente di valutazione)
- Valutazione avviene all'interno del corpo, se necessario.

La definizione

```
v = 1/0
```

- associa a `v` l'espressione `1/0`
- nessun errore fino a che si forzo la valutazione di `v`
- esempio scrivo `v` o `if v < 0 then 0 else 1` nell'interprete
  - `if v < 0 then 0 else 1` viene passato come argomento ad una funzione `show` che determina la stringa della sua rppresentazione
  - i singoli caratteri di `show (if v < 0 then 0 else 1)` vengono valutati e stampati

Dati lazy

- la valutazione differita, anche per i costruttori di dato, permette di definire dati potenzialmente infiniti,
- di volta in volta viene valutata solo la parte richiesta dal resto del codice
  per esempio, per applicare pattern matching

```
ones = 1 : ones :: Num a => [a]
```

- definizione ricorsiva,
- viene espansa sono quando necessario, per esempio esamino il secondo elemento di `ones`

```
head(tail(tail ones))
```

Costruzione della lista di tutti i naturali

```
numsFrom n = n : numsFrom (n + 1)
nums = numsFrom 0
```

Dati lazy

Costruzione della lista dei quadrati

```
squares = map (\x -> x^2) (numsFrom 0)
```

La sequenza di Fibonacci, vista come stream,

```
addList (x:xs)(y:ys) = (x + y) : addList xs ys

fib = 1:(addList fib (0 : fib))
```

- calcolata in maniera efficiente grazie alla meccanismo call-by-need, `fib` replicata ma valutata una volta sola
- corretto perché la somma della sequenza di Fibonacci con la sua traslata, di una posizione a destra, restituisce la sequenza di Fibonacci senza il primo elemento (correttezza non ovvia)

Estrarre parte finite da dati lazy

- funzione che seleziona la parte iniziale finita (funzione predefinita nel `Prelude`)

```
take 0  _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

- utile per interagire con l'interprete `ghci`
  - quando scrivo `exp` l'interprete forza la valutazione completa di `exp`
    - `exp` viene convertito in una stringa
      funzione `show`
    - ogni elemento della stringa viene stampato
  - se inserisco `ones` stampa la sequenza infinita di `1`
- funzione per selezionare l'iesimo elemento di uno stream

```
takeEl 0 (x:_)  = x
takeEl n (_:xs) = takeEl (n-1) xs
```

Stream in linguaggi funzionali eager

Attraverso l'uso di thunk posso rappresentare strutture dati infinite in linguaggi eager.

Gli esempi precedenti tradotti in Scheme come:

```
(define (numsFrom n) (lambda () (cons n (numsFrom (+ n 1)))))

(define nums (numsFrom 0))

(define (take n xs)
  (cond
    [(equal? n 0) null]
    [(equal? (xs) null) null]
    [true (cons (car (xs)) (take (- n 1) (cdr (xs))))]))

(take 20 nums)
```

Stream in Scheme

La funzione `map` sugli stream di Scheme diventa:

```
(define (lazymap f xs)
    (if (equal? (xs) null)
        xs
        (lambda () (cons (f (car (xs)))
                         (lazymap f (cdr (xs)))))))

(take 20 (lazymap (lambda (x) (* x x)) nums))
```

Sintassi infissa

- Diverse funzioni predefinite usano la sintassi infissa

```
+   *   :   ^   -
```

- ne esisto altre:
  - ++ concatenazione di stringhe
  - `.` composizione di funzioni `f . g = \ x -> f ( g x)`
- le operazione non rappresentate da identificatori standard usano la forma infissa

```
(#) a b = rem a b
(\\) a b = a + b
```

- possibile specificare tipo di associatività e priorità

```
infixl 9 #    --- infisso, assoc. a sinistra, priorità 9
infixr 5 \\   --- infisso, assoc. a destra, priorità 5
7  # 4 # 3 \\ 2 \\ 5  ---  ((7  # 4) # 3) \\ (2 \\ 5)
```

Passaggio tra notazioni infissa – prefissa

Le parentesi `( )` permettono di passare alla notazione prefissa

```
(+) :: Num a => a -> a -> a
```

- `(+)` coincide con le funzione `\ x y -> x + y`

inoltre

- `(+ 4)` è zucchero sintattico per `\ x -> x + 4`
- `(5 +)` è zucchero sintattico per `\ y -> y + 5`

gli apici permettono la trasformazione opposta:
scrivere una funzione binaria in forma infissa

```
5 `add` 4
```

applicabili a funzioni n-arie

```
add3Values x y z = x + y + z  :: Num a => a -> a -> a -> a
sei = (1 `add3Values` 2) 3
```

List comprehension

- liste sono pervasive di Haskell, utile avere zucchero sintattico

```
[ f x | x <- xs ]
```

applico la funzione `f` a tutti gli elementi della lista xs,

si richiama una notazione insiemistica

```
[ x + 1 | x <- [1..6] ]
map (+1) [1..6]
```

La parte `x <- xs` viene detta generatore,

possibili più generatori, si prendono tutte le combinazioni

```
[  (x * y) | x <- [1..10], y <- [1..10] ]
[[  (x * y) | x <- [1..10]] | y <- [1..10] ]
```

posso introdurre guardie, espressioni boolean per filtrare

```
[ (x * y) | x <- [1..10], y <- [1..10], (mod (x*y) 2) == 1 ]
```

Quicksort

non efficiente come la versione imperativa

```
quickSort [] = []
quickSort (x:xs) =
    quickSort[ y | y <- xs, y < x] ++
    (x : quickSort[ y | y <- xs, x <= y])

quickSort ([3,7..40]++[2,5..30])
quickSort "Hello World"
```

- stringhe liste di caratteri
- sui caratteri esiste una relazione d'ordine

```
quickSort :: Ord a => [a] -> [a]
```

- Haskell ha un meccanismo di type classes, parente delle interface di Java che permette un più sofisticato polimorfismo parametrico

Funzioni di base e Prelude

Haskell, attraverso il modulo precaricato Prelude, mette disposizione:

- le classiche operazioni aritmetiche e logiche, con usuale notazione infissa, implementate in linguaggio macchina,

```
+  -  *  /   ||   &&   <  >  <=  >=  ==  /=  ...
```

- una serie funzioni analitiche, implementazione interna

```
sin  cos tan  exp ...
```

- funzioni di I/O
- funzioni Haskell predefinite su liste, numeri

```
head  tail  take  sum  mcd  lcd  max
```

Documentazione

per le funzioni in Prelude

- pratica “A tour of Prelude”

http://www.cse.chalmers.se/edu/course/TDA555/tourofprelude.html

- più completa: tramite Hackage,

http://hackage.haskell.org/package/base-4.11.0.0/docs/Prelude.html

- in generale una Hackage offre una panoramica dei package, librerie, disponibile per Haskell

Esercizi

reverse: invertire un stringa

- semplice ma poco efficiente
- accumulatore e ricorsione di coda

```
reverseAux xs ys  = (reverse xs) ++ ys
```

- usando la foldl o foldr

Esercizi

reverse: invertire un stringa

- semplice ma poco efficiente
- accumulatore e ricorsione di coda

  ```
  reverseAux xs ys  = (reverse xs) ++ ys
  ```

- usando la foldl o foldr

```
reverse xs = reverseAux xs []
   where reverseAux [] ys  = ys
         reverseAux (x:xs) ys  = reverseAux xs (x:ys)
```

Fold left

Definire fold left, tipo

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

`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) ... )
```

Fold left

Definire fold left, tipo

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

`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) ... )
```

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Definire reverse da foldl

Fold left

Definire fold left, tipo

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

`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) ... )
```

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Definire reverse da foldl

```
reverse xs = foldl (\ys y -> (y:ys)) [] xs
```

alternativamente

```
reverse xs = foldl (\ys -> \y -> (y:ys)) [] xs
reverse xs = foldl (\ys -> (:ys)) [] xs
```

Fold right

Definire fold right, 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 ) ... )
```

Fold right

Definire fold right, 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 ) ... )
```

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Definere reverse con foldr

```
reverse xs = foldr (\ y ys -> (ys ++ [y])) [] xs
```

Pattern matching

Come vengono interpretate, eseguite le definizioni per pattern matching?

```
take 0   _ = []
take _  [] = []
take n (x:xs) = x : take (n-1) xs
```

zucchero sintattico per un costrutto case sull'enupla degli argomenti

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

Case – definizione per casi

Haskell permette definizione per casi `case` che considerano più valori contemporaneamente

Possono essere visti come zucchero sintattico per una serie di case sulle singole componenti

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

- le definizioni per pattern matching rendo il codice più leggibile
- permettono di scrivere in maniera sintetica, chiara, un insieme costrutti case annidati tra loro

Case singolo

Come viene eseguito un case? Es.

```
case x of
   [] ->
   y: ys ->
```

forza la valutazione dell'argomento `x`, passaggio per nome, alle variabili associo closure (espressioni, environment)

come avviene la valutazione di x?

Valutazione dell'argomento del case

dipende dal tipo di x

- x scalare, viene valutato completamente
  il suo valore comparato con quello dei casi
- x tipo `data`
  - viene valutato fino a determinare il suo costruttore
    - es. tipo `Tree Leaf n` o `Branch x1 x2`
    - valutazione lazy, si valuta lo stretto necessario,
      le componenti restano espressioni (closure)
  - i rami di case marcati con i costruttori

```
data Tree a   = Leaf a | Branch (Tree a) (Tree a)
height = \ t -> case t of
    Leaf n -> 0
    Branch t1 t2 -> max (height t1) (height t2)
```

- possibili definizioni non esaustive (non tutti i casi considerati),
- casi non previsti generano errori a run-time.

Pattern matching, risultati per un pattern

- i pattern possono essere piuttosto complessi
- dato un pattern (es. `x : 0 : xs`) ed un'espressione `e` la valutazione del pattern su `e` può
  - fallire, l'espressione valutata non rispetta il pattern, (es. se `e` diventa `3:1:e2` o se diventa `4:[]`)
  - avere successo, es. `e` riduce `5:0:e1`
    - in questo caso
      `x` viene legata a `5`
      `xs` alla closure `e1`
  - divergere (o generare errore), la computazione non produce sufficienti risultati (es. `e` riduce a `1:e3`, e la valutazione di `e3` diverge)

Sintassi

- cosa posso inserire in un espressioni pattern, es `3:x:y:[]`
  - costanti
  - costruttori
  - variabili
    - non possono essere ripetute
    - considerate nuove, variabili locali del pattern, nessun collegamento con variabili preesistenti

```
x = 10
take 0   _ = []
take n (x:xs) = x : take (n-1) xs
```

    - wildcard: caso particolare di variabile, rimarco il disinteresse per il valore istanziato

Ordine di valutazione

Come viene forza la valutazione degli argomenti

- in base all'ordine di scritture delle regole, dalla prima all'ultima
- sulla singola regole, i pattern vengono testati da sinistra a destra
  - se un pattern fallisce: passo 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

Ordine di valutazione

Esempio: le seguenti definizioni:

```
take 0   _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

e

```
take _ [] = []
take 0   _ = []
take n (x:xs) = x : take (n-1) xs
```

non sono equivalenti

Quando possono dare risultati differenti?

Pattern con guardie

Ai pattern posso far seguire una lista di guardie

- espressioni booleane
- esaminate dalla prima all'ultima, se il pattern ha successo
- se una guardia valuta `True`, valuto il corpo corrispondente
- se una guardia valuta `False`, si passa alla successiva
- se tutte `False`, il pattern fallisce e si passa al successivo

```
sign x | x > 0  = 1
       | x == 0 = 0
       | x < 0  = -1
```

- posso usare le guardia anche con il costrutto `case`

```
sign x = case x  of
           y | y > 0  -> 1
             | y == 0 -> 0
             | y < 0  -> -1
```

Guardie: meccanismo generale di programmazione

If then else

Posso usare il costrutto

```
if b then e1 else e2
```

equivalente a

```
case b of
    True -> e1
    False -> e2
```

Forma generale del pattern con guardia

Definizione per pattern matching

```
"funName" "pattern1" | "guardia11" = exp11
                     | ...
                     | "guardia1n" = exp1n
          "pattern2" | "guardia21" = exp21
                     | ..
                     | "guardia2m" = exp2m
          "pattern3" ...
          ...
```

Forma generale del pattern con guardia

Posso usare le guardie anche col costrutto case

```
case "pattern1" | "guardia11" -> exp11
                | ...
                | "guardia1n" -> exp1n
     "pattern2" | "guardia21" -> exp21
                | ..
                | "guardia2m" -> exp2m
     "pattern3" ...
     ...
```

Esempio:

Calcolare la somma dei numeri pari di una lista, definizione per pattern matching.

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
```

Calcolare la somma dei numeri dispari di una lista, definizione via case

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
```

Calcolare la somma dei numeri dispari di una lista, definizione via case

```
sommaDispari = \ ys ->
  case ys of [] ->  0
             (x:xs) | (mod x 2) == 0 -> sommaDispari xs
                    | True           -> x + sommaDispari xs
```

Layout

Separatori e blocchi possono essere usati esplicitamente

```
z = let x = 5 + y
        y = 6
    in x + y
```

può essere scritto come

```
z = let { x = 5 + y; y = 6 } in x + y
```

Una tabulazione imprecisa a porta a errori di compilazione. Es:

```
z = let x = 5 + y
          y = 6
    in x + y
```

- in questi casi i messaggi di errore del compilatore possono essere poco chiari: non si evidenzia l'errore di tabulazione

Record con campi etichettati

Le componenti di un data type sono identificate dalla loro posizione

```
data Point = Pt Float Float
p = Pt 3 4
```

Posso accedere alle componenti per pattern-matching

```
xCoord (Pt x _) = x
```

Alternativamente, come nei record, posso accedere alle componenti attraverso etichette field labels

```
data Point = Pt {xCoord, yCoord :: Float}
```

Record

```
data Point = Pt {xCoord, yCoord :: Float}
p = Pt{ xCoord = 3, yCoord =5}
a = xCoord p
q = p{ xCoord = 5}
a2 = xCoord p
b = xCoord q
```

- attraverso le etichette specifico le componenti di un nuovo Pt
- le etichette diventa funzioni per accedere alle componenti (distruttrici) `xCoord :: Pt -> Float`
- posso definire un nuovo punto `q`, a partire da uno esistente `p`, modificando solo alcune campi, gli altri vengono copiati.

Record

Una stessa etichetta può essere utilizzata da più costruttori.

```
data Point23 = Pt2 {xCoord, yCoord :: Float}
             | Pt3 {xCoord, yCoord, zCoord :: Float}
```

Etichette con tipi union rendono possibili errori di tipo rilevabili solo a run time,

- la dichiarazione seguente accettata,

```
p = Pt2{xCoord = 3, yCoord = 5}
a = zCoord p
```

- genera errore solo se si valuta `a`.

Esercizi

Definire sommaPari e map usando più “idiomi”, metodi di scrittura

- pattern matching
- costrutto case
- ricorsione di coda
- via foldl
- via foldr

Esercizi

Definire tipo di dato albero binario, `BTree` con tutti i nodi etichettati,

Definire le seguenti funzioni sui BTree

- `sumBTree`
- `depthBTree`
- `takeDepthBTree`

Definire le seguenti funzioni su alberi binari di ricerca, BST

- insert
- search
- toList

Ordinare una lista usano i BST.

Type Classes

Meccanismi per incrementare il polimorfismo parametrico

Nella versione semplice il polimorfismo parametrico

- usa espressioni di tipo con variabili (parametri) per descrivere il comportamento di programmi.
  Esempio:

```
map _ []       = []
map f (x:xs)   = f x : map f xs
```

- `map :: (a->b) -> [a] -> [b]`
- tipo derivato mediante un algoritmo di inferenza di tipo
  - associa a `map` in suo tipo parametrico più generale,
  - tutti gli altri possibili tipi per `map`,
    es `(a->a) -> [a] -> [a]` oppure `(Char->b) -> [Char] -> [b]`
    ottenibili per istanziazione del tipo più generale
- esempio di derivazione di tipi per `map` ...

Polimorfismo parametrico

Esistono funzioni polimorfe non descrivibili in questo modo. Esempio

```
quickSort [] = []
quickSort (x:xs) = quickSort[ y | y <- xs, y < x]
                        ++ x : quickSort[ y | y <- xs, x <= y]
```

ha tipo `Ord a => [a] -> [a]`

`quickSort` applicabile ad una lista generica, `[a]`,
a condizioni che i suoi elementi,

- possano essere applicate le funzioni `<` `<=`
- condizione espressa mediante la condizione `Ord a`:
  il tipo `a` appartenete alla Classe `Ord`

Overloading

Affinché `quickSort` sia polimorfa bisogna:

- permettere che le funzioni < <= appartengano a più tipi
  - eventualmente con implementazione diverse, più metodi, codici, per confrontare oggetti
- ossia: permettere il polimorfismo di overloading

Mostreremo dichiarazioni per:

- specificare insiemi di tipi, classi, (esempio `Ord`) con un insieme di funzioni associate
- specificare che un tipo appartiene ad una classe (esempio `Char` appartiene a `Ord`)

Type classes - dichiarazioni e istanziazioni

Le classi definiscono insiemi di tipi che possseggono le necessarie funzioni, chiamate metodi.

```
class Eq a where
  (==)   ::  a -> a -> Bool
```

- Un tipo `T` per appartenere alla classe `Eq` deve avere una funzione == (usata in notazione infissa).

L'associazione di un tipo ad una classe,

- devo essere dichiarata esplicitamente
- bisogna fornire un implementazione per i metodi della classe

```
instance Eq Integer where
  x == y    =  integerEq x y
```

Uso delle type class

- Alla funzione polimorfa

```
elem x []  = False
elem x (y:ys) = x == y || (elem x ys)
```

`viene associato tipo`

```
elem :: (Eq a) => a -> [a] -> Bool
```

- il simbolo => 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`.

Implementazione di default dei metodi

La “vera” definizione di `Eq`, classe predefinita in Prelude,

```
class Eq a where
  (==), (/=)    ::  a -> a -> Bool
  x /= y        = not (x == y)
  x == y        = not (x /= y)
```

- fornisco una implementazione di default per alcuni metodi, basata sugli altri metodi,
- nelle definizioni `instance`
  - posso definire tutti i metodi
  - definirne solo alcuni e sfruttare le implementazioni default per gli altri
  - nella specifica di una classe si definisce anche una lista di definizioni minimali

Implementazione di default dei metodi

```
instance Eq Integer where
  x == y     =  integerEq x y
  x \= y     =  False
```

oppure

```
instance Eq Integer where
  x == y     =  integerEq x y
```

oppure

```
instance Eq Integer where
  x \= y     =  False
```

- definizione semanticamente scorrette accettate da Haskell,
- a livello di documentazione, nei package, vengono specificate le uguaglianze attese ma
- Haskell controlla solo:
  - il tipo dei metodi
  - che i metodi forniti sia sufficienti a determinare tutti gli altri

Definizione di instance polimorfe

```
instance (Eq a) => Eq (Tree a) where
  Leaf l1          == Leaf l2          = l1 == l2
  Branch tl1 tr1 == Branch tl2 tr2 = tl1 == tl2 && tr1 == tr2
  _                == _                = False
```

- per definire l'equivalenza su alberi,
  serve l'equivalenza sugli elementi base, ossia
- sotto la condizione `Eq a` posso dichiarare `Eq (Tree a)`
- nella definizione delle metodo == su `Tree a` uso:
  - `l1 == l2` definita da `Eq a`
  - `tl1 == tl2` chiamata ricorsiva

Queste definizioni strutturali possono essere create automaticamente da Haskell usando `deriving` nella definizione di tipo

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
 deriving (Eq)
```

Relazione di sottoclasse

- Posso dichiarare una classe sottoclasse di un altra

```
class (Eq a) => Ord a where
  (<), (<=), (>), (>=)  :: a -> a -> Bool
  min, max              :: a -> a -> a
  x <= y    = x < y || x == y
  ...
```

- `Ord` e' sottoclasse (subclass) di `Eq`
- `Eq` e' superclasse (superclass) di `Ord`

- dichiara che la classe `Ord` possiede tutti i metodi della classe `Eq` quindi == e \=
- nel definire tipo instance di `Ord` devo fornire anche in metodi di `Eq`
- un tipo `T` instance di `Ord` e' automaticamente instance di `Eq`

Sottoclassi multiple

- Posso definire un tipo sottoclasse di più classi

```
class (Eq a, Show a) => EqAndShow a where
  ...
```

- In tipo instance di `EqAndShow` deve possedere i metodi delle due sottoclassi `Eq` e `Show`
  più eventuali metodi aggiuntivi

Visibilità dei nomi

- classi e metodi definiti a livello globale,
  visibili in tutto il programma
- uno stesso nome di metodo non può essere dichiarato su più classi
- metodi comuni possibile solo attraverso la definizione di sottoclassi
- metodi, variabili, funzioni sono nello stesso name space
  - non posso usare lo stesso nome per un metodo ed una funzione

Relazioni con i linguaggi OO.

Pur con i dovuti distinguo esiste un relazione.

Miglior paragone, Java.

Haskell	Java
Types	Classes
Type Classes	Interfaces
Elements	Objects
`instance Schema Type`	`class Type implements Schema {...}`
Polimorfismo parametrico con classi	Tipo generici con relazione di sottotipo.

Relazioni con i linguaggi OO.

Naturalmente Haskell non OO, quindi una lunga serie di differenze

- 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

Una catalogazione delle espressioni Haskell

- Espressioni semplici,

```
5 + 2
\ x -> x + 1
```

- Espressioni di tipo,

```
Integer
Integer -> Integer
```

- costruttori di 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)
```

- classi, proprietà di tipi

```
Eq
Ord
```

Kind

Struttura piuttosto complessa:

- necessario mettere ordine
- nozione di `kind`
- estensione della nozione di tipo, applicabile ad espressione che non sono elementi e non hanno tipro (tipi, costruttori di tipo, classi)
- ai vari identificatori associa un etichetta che indica a che categoria appartengono
- kind dedotto dal compilatore,
  possile interrogare l'interprete per sapere il kind associato ad un espressione

Kind

- alle espressioni semplici associ un tipo

```
λ> :type (+1)
(+1) :: Num a => a -> a
```

- a tipi, il kind $*$,

```
λ> :kind Integer -> Integer
Integer -> Integer :: *
```

Kind

- ai costruttori di tipo, 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 :: * -> *
```

Kind

- alle classi, proprietà di tipi

```
λ> :kind Eq
Eq :: * -> Constraint
λ> :kind Functor
Functor :: (* -> *) -> Constraint
```

valutazione dei `kind` fatta dal compilatore,
un type checking di secondo livello

Costruttori di tipo e classi predefinite

In Prelude

https://hackage.haskell.org/package/base-4.6.0.1/docs/Prelude.html

vengono definite

- un insieme di classi
  - ricca relazione di sottoclasse
  - principalmente ordini e numeriche
- tipi, costruttori di tipo, definiti anche come istanze delle opportune classi,

Esempi di classes - Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- Functor non definisce un vincolo sui tipi ma sui costruttori di tipo
- Esempio, il costruttore di lista

```
instance Functor [] where
  fmap = map
```

- ci si aspetta che le seguenti regole siano soddisfatte

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

Costruttori di tipo canonici, Maybe

- Maybe: serve a rappresentare eccezioni, errori

```
data Maybe a = Nothing | Just a
```

```
- `Nothing` eccezione
- `Just 5` 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)
```

Costruttori di tipo canonici, Either

- Either, unione tra due valori

```
data Either a b = Left a | Right b
```

- `Either` non è un instance of `Functor` (kind non corretto) `Either Int` si.

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 singoli e doppie)
  - complessi (coppie di floating point)
- Catalogato con un ricco insieme di classi.

Classi numeriche

- `Num` classe più generale,

  non sottoclasse di `Ord` perché i complessi non sono ordinati

```
class (Eq a) => Num a where
(+), (-), (*)  :: a -> a -> a
negate, abs    :: a -> a
fromInteger    :: Integer -> a
```

- `Real` la classe più generale di numeri ordinati

```
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
```

Classi numeriche

- `Real` non possiede la divisione 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
```

Classi numeriche

- Floating fornisce le funzioni analitiche

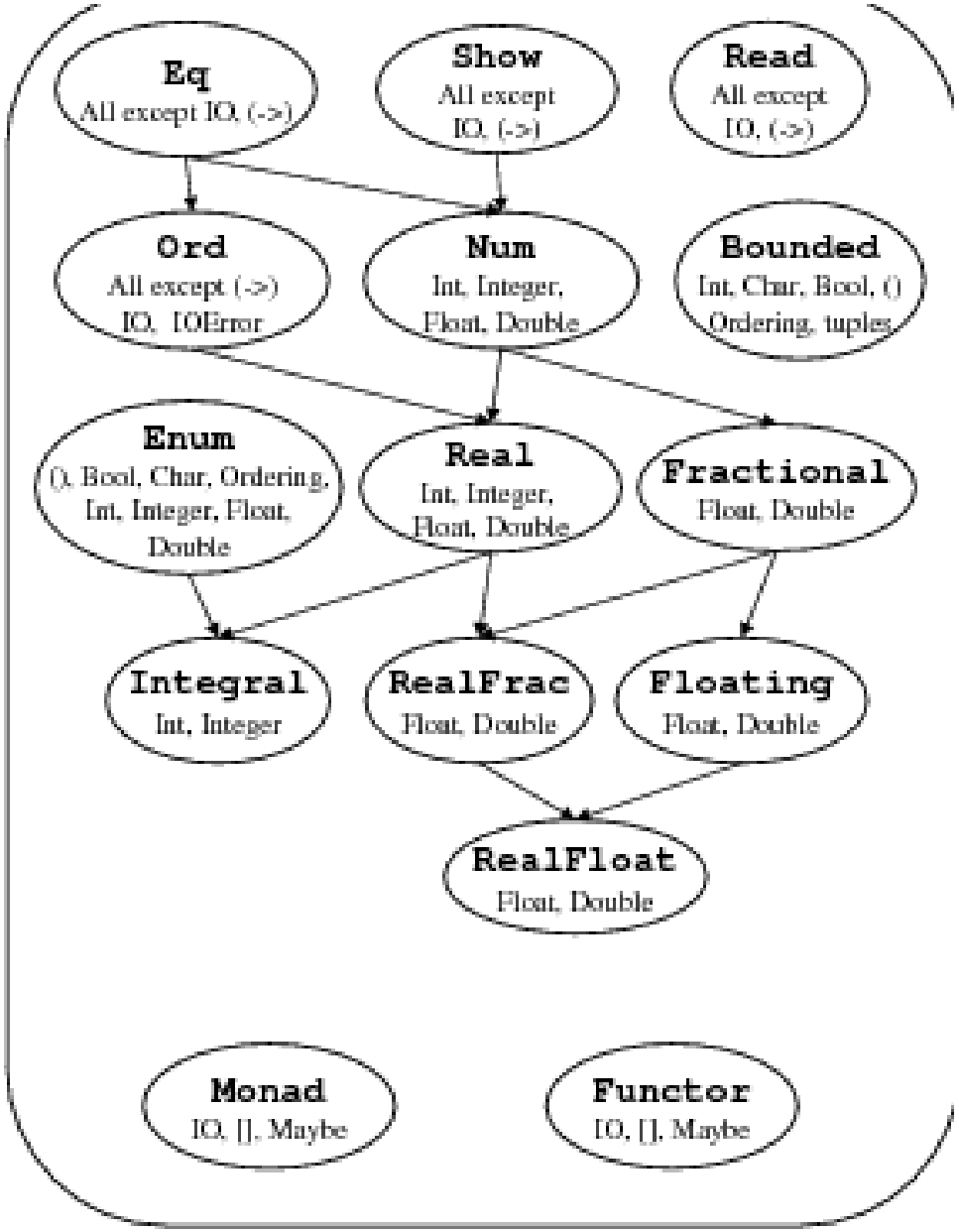
```
class Fractional a => Floating a
```

```
- RealFrac fornisce funzioni di arrotondamento (all'intero pi
```

```
class (Real a, Fractional a) => RealFrac a
```

- con le relative istanze di tipo `Integer`, `Double`, `Float`, `Int32`, `Natural`
  - `Num` conta una 40ina di possibili istanze

Schema


Eq
All except IO, (->)
Show
All except IO, (->)
Read
All except IO, (->)
Ord
All except (->) IO, IOError
Num
Int, Integer, Float, Double
Bounded
Int, Char, Bool, () Ordering, tuples
Enum
(), Bool, Char, Ordering, Int, Integer, Float, Double
Real
Int, Integer, Float, Double
Fractional
Float, Double
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double
RealFloat
Float, Double
Monad
IO, [], Maybe
Functor
IO, [], Maybe

Casting esplicito

- per passare da un tipo numerico all'altro devo esplicitare nel codice la conversione
- 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
fromRational :: Fractional a => Rational -> a
...
ceiling  :: (RealFrac a, Integral b) => a -> b
floor    :: (RealFrac a, Integral b) => a -> b
truncate :: (RealFrac a, Integral b) => a -> b
round    :: (RealFrac a, Integral b) => a -> b
```

I/O Pragmatica

Nell'uso pratico `IO` haskell non troppo complesso

- definisco un variabile `main` di tipi `IO ()`
- codice di `main` simile ad una sequenza di comandi
  - leggono dati da input
  - chiamano altre funzioni per valutare il risultato
  - generano in uscita il risultato finale

l'apparente semplicità nasconde concetti più complessi

- una gestione pulita degli effetti collaterali

I/O

Haskell linguaggio funzionale puro.

- non esiste memoria, stato,
- definisco solo funzioni da un tipo ad un altro, senza effetti collaterali

Questo vincolo rendo complesso gestire l'input-output

- anche in una versione semplice: stringa di input e stringa di output
- la coppia di stringhe costituiscono uno stato, `S`, che può
- viene modificato dalle operazioni di input - output
  - input: consuma caratteri nel file di input per generare un valore
  - output: da un valore genera carattere inseriti nel file di output
- si creano effetti collaterali

Due alternative

- permetto che la valutazione di espressioni abbia effetti collaterali (side effect), senza segnalare questi effetti collaterali nel tipo dell'espressione
  - linguaggio funzionale non puro
  - soluzione di ML (Scheme)
  - il comportamento del programma dipende dall'ordine di valutazione
- introduco esplicitamente lo stato `S` nel tipo delle funzioni
  - una funzione `f :: A -> B` che modifica lo stato, rappresentata da:
  - `fs :: (A, S) -> (B, S)`
    curryficando:
    `fs :: A -> (S -> (B, S))`
- seconda soluzione scelta da Haskell, funzionale puro:
  - usando funzioni con effetti collaterali, definisco nel modulo principale una funzione
    `main :: S -> ((), S)` `main :: IO()`
  - l'esecuzione del programma, provoca la valutazione di `main` che modifica dello stato

Tipo `IO`

- In Haskell esiste un costruttore di tipi predefinito `IO`
  - che posso interpretare, intuitivamente, come

```
type IO a = (S -> (a,S))   --- pseudo definizione
```

  - dove `S` rappresenta lo stato
- espressioni con tipo `IO a` rappresentano action
  - comandi, se hanno tipo tipo `IO ()`
  - espressioni, di tipo `a`, con effetti collaterali, se hanno tipo `IO a`
- la definizione di tipo `IO` nascosta,
  - non esistono costruttori espliciti
  - ma esistono funzione predefinite sui tipi `IO`
    `getChar`, `putChar` ...

Funzioni di input output

- `getChar :: IO Char`
  ossia `S -> (Char, S)`
  dato un stato, mi restituisce un carattere e lo stato modificato,
  (con un carattere in meno nel file di input)
- `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
  - `putChar`, dato un carattere ed uno stato
    restituisce lo stato modificato
    (con un carattere in più nel file di output)

Stato

- Come già scritto, il tipo `IO` nascosto, possiamo immaginare

```
type IO a = (S -> (a,S))
```

- Cosa sia esattamente lo stato `S` non è definito esplicitamente, intuitivamente contiene
  - i file di input e output
  - gli altri file manipolabili
  - lo stato che rappresenta eventuali condizioni di errore, eccezioni,

Costrutti per comandi

Espressioni di tipo `IO` hanno:

- solo un numero limitato di costrutti per manipolarle
  la parti imperativa, non viene mescolato con quella funzionale
- le uniche funzioni `f` definibili di tipo `(IO a) -> Integer` sono funzioni costanti
  funzioni che preso in input un comando (action) `c` lo ignorano, non lo eseguono

Operatori su I0

Posso comporre i comandi base con

- `(>>) :: I0 a -> I0 b`
  `(S -> (a, S)) -> (S -> (b, S)) -> (S, (b -> S))`
  - composizione di due comandi
  - trascurando in valore in `a` generato dal primo
  - `;` nei linguaggi imperativi
- `(>>=) :: I0 a -> ( a -> I0 b) -> I0 b`
  `(S -> (a, S)) -> ((a, S) -> (b, S))) -> (S, (b -> S))`
  - compone i due comandi (azioni)
  - tenendo conto del risultato generato dalla prima

Posso creare un comando, banale, con:

- `return :: a -> I0 a  (a -> (S -> (a, S)))`
  - presa un'espressione `v` di tipo `a`
  - restituisce un'azione che lascia lo stato inalterato e ritorna `v`

Monadi

Gli operatori `(>>)`, `(>>=)`, `return` sono motivati dalla nozione di monade

Nozione presa dalla matematica (teoria delle categorie)

- utile per descrivere diverse costruzione della matematica (insieme delle parti, grammatica libera su un alfabeto)
- in informatica utile per descrivere diversi costruttori di tipo (liste, `IO`, `Maybe`)
- in Haskell, esiste una `type class` per costruttori di tipo `Mondad m : * -> *`
  - `IO`, `[]`, `Maybe` sono `instance of Monad`, esempi di monadi
  - tra le funzioni caratterizzanti delle monadi troviamo
    - `(>>=) :: m a -> (a -> m b) -> m b`
    - `return :: a -> m a`

Monadi esempi

Le funzioni caratteristiche delle monadi possono essere usate come base per la programmazione:

```
[(x,y) | x <- [1,2,3] , y <- [3,2,1], x /= y]
```

```
[1,2,3] >>= (\ x -> [3,2,1] >>= (\y -> return (x/=y) >>=
  (\r -> case r of True -> return (x,y)
                   _    -> fail "")))
```

```
do x <- [1,2,3]
   y  <- [3,2,1]
   True <- return (x /= y)
   return (x,y)
```

Sintassi

La sintassi `(>>)`, `(>>=)` poco intuitiva nel caso di composizione di comandi

- `c1 >> c2 >> c3` associa a destra
  `c1 >> (c2 >> c3)`
  può essere scritto come `do { c1; c2; c3 }`
  o come

```
do c1
   c2
   c3
```

Esempio

```
do getChar
   putChar 'h'
   putChar 'e'
```

Sintassi

- `c1 >>= (\ x -> ( c2 >>= (\ y -> c3)))`
  lo posso scrivere come

```
do x <- c1
   y <- c2
   c3
```

Esempio

```
do c1 <- getChar
   c2 <- getChar
   putChar c2
   putChar c1
```

Separazione tra parte funzionale e IO

- Su comandi, espressioni di tipo `IO a`, posso usare solo gli operatori >> e >>=
  - posso comporre comandi
  - un comando può passare dati ad un altro comando
  - un espressione di tipo standard (`Integer`) non può forzare l'esecuzione di un comando,
    o estrarre informazioni da questo
- Un comando `IO a` costituito dalla composizione mediante
  - `>>` , `>>=`, `return` , `do`
  - di comandi elementari predefiniti,
    ma anche comandi composti
  - posso inserire funzioni da valutare

Riassumendo:

- dentro un comando posso chiamare una funzione
- nelle valutazione di una funzione pura non posso inserire un comando

Programmi compilati, entry point

In un programma compilato, devo definire l'entry point

- l'espressione che deve essere valutata come risultato dell'esecuzione del programma
- questa espressione deve avere:
  - nome `main`
  - tipo `IO`, tipicamente `IO ()`
    se avesse tipo diverso l'esecuzione del programma non avrebbe nessun effetto
- a volte necessario introdurre dichiarazione di tipo per `main`

```
main :: IO()
main = do c <- getChar
          putChar c
          putChar '\n'
```

- nel caso di programma diviso in moduli (vedi più avanti), `main` deve essere inserita nel modulo `main`

Funzioni di IO

Posso costruire comandi più complessi di `getChar` e `putChar`

combinazioni più comandi

```
getLine2 :: IO String
getLine2 = do c <- getChar
              if c == '\n'
                then return ""  --- alias per []
                else do l <- getLine2
                        return (c:l)
```

- getLine presente in Prelude
- funzione `if then else` in notazione infissa

Esercizio: controllare la correttezza di tipo

Stampa stringa

```
putString :: String -> IO ()
putString [] = return ()
putString (x:xs) = do putChar x
                      putString xs
```

- in Prelude `putStr, putStrLn :: String -> IO ()`

Esempi I/O

Posso trattare le action (comandi), come un qualsiasi altro dato
es, costruisco un vettore di comandi

```
toDoList = [putChar 'H',
            do c <- getChar
               putChar c,
            putStrLn "ello" ]
toDoList :: [IO ()]
```

- non posso eseguire un vettore di comandi,
- solo il singolo comando definito in main è valutato
  tutto il resto sono espressioni, che possono essere manipolate,
- devo combinare i comandi in uno, e valutarlo nel main

```
sequence2 [] = return ()
sequence2 (c:cs) = do c
                      sequence2 cs
```

Tipo e definizioni alternative

```haskell
sequence2 :: Monad m => [m a] -> m ()
sequence2 :: [IO a] -> IO ()

-- foldr :: (a -> b -> b) -> b -> [a] -> b
sequence3 :: [IO a] -> IO ()
sequence3 = foldr (>>) (return ())

-- foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
sequence4 :: [IO ()] -> IO ()
sequence4 = foldl (>>) (return ())

c2 = sequence2 toDoList
c3 = sequence3 toDoList
c4 = sequence4 toDoList
```

una seconda definizione per putString

```
putString2 = sequence . map putChar
```

File

- Gestione semplice dei file attraverso le funzioni

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

dove

```
type FilePath = String  -- Defined in ‘GHC.IO’
```

Posso costruire comandi per copiare un file in un altro

```
copy fileS fileD = do s <- readFile fileS
                      writeFile fileD s

c5 = copy "sommaPari.hs" "prova"
```

File

Oppure comandi per copiare il contenuto il contenuto dopo averlo modificato

```
modify fSource fDestin trasf = do s <- readFile fSource
                                  writeFile fDestin (trasf s)

c6 = modify "sommaPari.hs" "prova" tail
```

sorgente e destinazione devono essere diversi.

Classi Show Read

Definiscono funzioni standard per stampare e leggere valori da IO

```
class Show a where
  show :: a -> String
  shows ::  a -> String -> String
  showsPrec :: Int -> a -> String -> String
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}
```

- `show` funzione usata dall'interprete per visualizzare espressioni
  - insieme a `putString` mi permette di stampare dei dati da programma

Show

- `shows :: a -> String -> String`
  funzione con accumulatore per migliorare l'efficienza, evito di dover concatenare liste

```
putStr ("i num. " ++ show x ++  ' ': show y ++ " sono uguali "
```

diventa

```
putStr ("i num. " ++ shows x  (' ': shows y " sono uguali "))
```

Esempio

```
data BTree a = Null | BTree a (BTree a) (BTree a)

showsBTree  Null  = ('_':)
showsBTree (BTree n tl tr) =
  ('(':) . showsBTree tl .  sh
showsBTree :: Show a => BTree a -> String -> String

instance Show a => Show (BTree a) where
  showsPrec _ = showsBTree
```

In alternativa:

```
data BTree a = Null | BTree a (BTree a) (BTree a)
  deriving (Eq, Show)
```

Read

Una type classe per leggere dati
trasformare una stringa di caratteri in un dato

Implementa un parser, usando back-tracking
semplice ma inefficiente

```
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]
  ...
  ...
  {-# MINIMAL readsPrec | readPrec #-}

type ReadS a = String -> [(a, String)]
```

Esempio

```
readsBTree ('_':s) = [(Null,s)]
readsBTree ('(':s) =
  [((BTree n tl tr), s3) | (tl, s1)  <- readsBTree s,
                           (n, s2)   <- reads s1,
                           (tr, ')':s3)  <- readsBTree s2 ]

readsBTree :: Read a => String -> [(BTree a, String)]

instance Read a => Read (BTree a) where
  readsPrec _ = readsBTree
```

Moduli

Sistema per gestire la visibilità dei nomi.

- Quadro generale
  - codice distribuito su più file
  - ogni file contiene una serie di definizione
    - tipi, classi, istanza, funzioni (posso apparire in ogni ordine)
    - solo alcuni resi visibile all'esterno
    - altre definizioni ausiliare
  - il programma principale sceglie
    - da che moduli, file importare
    - seleziona cosa importare

Esempio

```
module BinarySearchTree (Bst (Null, Bst), insert, empty)
  where
data Bst = ...
insert = ...
empty  = ...
```

- la lista `(Bst (Null, Bst), insert, empty)` definisce in nomi esportati
- definizione che non appaiono nella lista non visibili all'esterno
- i costruttori di in tipo di dato messi vicino al tipo per leggibilità
- se ometto la lista dei nomi, tutto viene esportato
  - `Bst(Null, Bst)`
  - `Bst(..)` possibile sintassi
  - `Bst(Null)` costruttore `Bst` non visibile all'esterno.

Tipi di dato astratti

- Posso utilizzare il moduli per implementare i tipi di dato astratti
  - nascondo l'effettiva implementazione del tipo di dato
  - fornisco solo un insieme di funzioni primitive per agire su questo
  - posso cambiare in modo trasparente l'implementazione

```
module BinarySearchTree (Bst, insert, null, empty, search)
```

Importazione

```
module Main (main)
  where
import  BinarySearchTree (Bst (Null), insert)
```

- le dichiarazione di `import` vanno messe in testa
- posso selezionare i nomi da importare
- `import BinarySeartTree` senza lista, importata tutti i nomi esportati
- `import BinarySeartTree hiding (insert)` nasconde alcune entity che posso ridefinire

Qualified names

Attraverso le importazione da moduli

- possibile che lo stesso nome abbia più definizioni
- (ridefinizione dei nomi non possibile all'interno dello stesso modulo)

Tutte le definizione restano valide

- devo eliminare l'ambiguità usando qualified names, nomi con prefissi
  - `BinarySearchTree.insert`
  - `Main.insert`
- per indicare `insert` nel caso
  - abbia importato il modulo `BinarySearchTree`
  - abbia definito nel codice una funzione `insert`.

Array

- In un linguaggio funzionale un array `[]char A` diventa funzione `A :: Int -> Char`, o una lista `A :: [Char]`
- Implementazione inefficiente, per migliorare le prestazioni Haskell fornisce un tipo `Array`, definito nel modulo `Array`, da caricare

```
import Array
```

nel modulo troviamo le seguenti definizioni

Index type

Un type class per gli indici

```
class (Ord a) => Ix a where
    range   :: (a,a) -> [a]
    index   :: (a,a) -> a -> Int
    inRange :: (a,a) -> a -> Bool
```

- posso costruire array con indici di `a` tipo qualsiasi, basta che `a` appartengano alla classe `Ix`
- sono dichiarati istanza di `Ix` gli interi, caratteri, enuple di istanze di `Ix` `(Int,Int)` usabili immediatamente come indici

Funzioni sugli array

Elenchiamo le più utili

Creazione:

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

fornisco estremi degli indici, lista degli elementi, anche fuori ordine, (indice, valore),
posso non inizializzare tutti gli elementi

```
squares = array (1,100) [(i, i*i) | i <- [1..100]]
```

possibili definizioni ricorsive

Accesso ai singoli elementi

```
(!) :: Array a b -> a -> b
squares!7  =>  49
```

Funzioni sugli array

Accesso agli estremi del campo indici

```
bounds:: Array a b -> (a,a)

bounds squares  =>  (1,100)
```

Modifica, multipla di alcuni elementi nell'array:

```
(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b

swapRows :: (Ix a, Ix b, Enum b) =>
            a -> a -> Array (a,b) c -> Array (a,b) c
swapRows i i' a =
  a // ([((i, j), a!(i',j)) | j <- [jLo..jHi]] ++
        [((i',j), a!(i ,j)) | j <- [jLo..jHi]])
    where ((iLo,jLo),(iHi,jHi)) = bounds a
```

Formalmente si crea un nuovo array,
ma evitando di duplicare tutti gli elementi.

Analisi sintattica e lessicale in Haskell

I tool Alex e Happy

Analisi lessicale e sintattica

In precedenza abbia parlato del front end dei compilatori

- analisi lessicale
  - divido il codice del programma una sequenza di parole, lessemi
  - l'insieme di lessemi diviso in classi (identificatori, parole chiave, ...)
  - ogni classe descritta da un espressione regolare
  - Lex, Flex strumenti per costruire scanner, in C
- analisi sintattica
  - dalla sequenza di lessemi, costruisco un albero che rappresenta la struttura sintattica del programma
  - struttura definita da una serie di categorie sintattiche (espressioni, comandi, dichiarazioni, ...)
  - ogni categoria descritta da una grammatica libera
  - Yacc, Bison strumenti per costruire parser, in C
- analisi semantica (principalmente type checking)

Alex e Happy

- Lex, flex producono scanner
  - a partire da un insieme di regole
    - espressioni regolari, azioni corrispondenti
  - dichiarazione ausiliare
    - grammaticali
    - codice C
- Yacc, Bison, similmente producono parser
- Alex e Happy, strumenti analoghi a Lex e Yacc, ma che producono codice Haskell
- concettualmente più semplici e puliti
- li presentiamo con un certo dettaglio
- parte del corso relativa agli aspetti di compilazione del codice

Alex

Genera uno scanner

- stringa, divisa in lessemi, ogni lessema trasformato in un token

Parti principali del codice Alex:

- definizione del tipo dei token
- regole
  - espressione regolare
  - funzione che trasforma la stringa associato in token

Parti ausiliarie

- codice aggiuntivo
- definizione ausiliare

Alex - struttura del sorgente

```
{                           -- codice Haskell da mettere in testa
module Main (main) where
}
%wrapper "basic"            -- definisce il tipo di scanner

$digit = 0-9                -- dichiarazioni ausiliari
$alpha = [a-zA-Z]           -- insiemi di caratteri
@num   = $digit+            -- espressioni regolari

tokens :-                   -- nome arbitrario e separatore
                                    -- regole
  let             { \s -> TokenLet }    -- forma: regexp  azione
  $white+         ;                     -- azione vuota
  @num            { \s -> TokenInt (read s) }
  [\=\+\-\*]      { \s -> TokenSym (head s) }
  ...             -- ogni azione stesso tipo :: String -> Token
```

Alex - struttura del sorgente - continua

```
{                       -- codice Haskell
data Token =            -- definisco il tipo dei token prodotti
    TokenLet            |
    TokenIn             |
    TokenSym Char       |
    TokenVar String     |
    TokenInt Int
    deriving (Eq,Show)

main = do               -- alexScanToken :: String -> [Token]
  s <- getContents
  print (alexScanTokens s)
}
```

File Alex

- deve avere il suffisso file.x
- compilato genera file.hs

Struttura:

- inizio, codice Haskell da inserire in testa al file (Haskell) prodotto inserire nome modulo eventuali import
  - meglio non definire funzioni, possono creare conflitti
- `%wrapper` varie opzioni sul codice prodotto
- dichiarazioni ausiliare macro, definisco
  - insiemi di caratteri
  - espressioni regolari
- segue nome mnemonico e separatore `:-`
- regole
- coda, codice Haskell da inserire in coda.

Regole Alex

- nelle forma:
  - espressione regolare
  - codice associato
- codici associati tutti dello stesso tipo
  - tipo `String -> type` nel caso `%wrapper "basic"`
- se più espressioni regolari riconoscono la stringa di input
  - scelgo quella che riconosce più caratteri
  - a parità di lunghezza, scelgo la prima nella lista

Espressioni regolari di Alex

Simili a Lex, ma con alcune differenze

- separo:
  - espressioni regolari generiche
    - uso @ident come nomi per reg-expre
  - insiemi di caratteri, espressi regolari che generano singoli caratteri
    - uso $ident come nomi per insiemi

Sintassi per gli insiemi in Alex

- 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

Macro

- . – tutti i caratteri escluso `\n` newline
- $printable – tutti i caratteri stampabili
- $white – tutti gli spazi `[\ \t\n\f\v\r]`.

Sintassi per le espressioni regolari

- 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

- `basic`
  - più semplice
  - fornisce funzione `alexScanTokens :: String -> [token]`
  - tutte le azioni devono avere tipo `String -> token`
  - per un qualche tipo token
- `posn`
  - rispetto a `basic` fornisce informazioni sulla posizione, per debugging
  - usa un tipo

```
data AlexPosn = AlexPn !Int  -- absolute character offset
                       !Int  -- line number
                       !Int  -- column number
```

  - tutte le azioni devono avere tipo `AlexPosn -> String -> token`
  - l'argomento `AlexPosn` relativo al primo carattere della stringa riconosciuta
  - fornisce funzione `alexScanTokens :: String -> [token]`

Esempio posn

```
...
%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)
```

Esercizi

Da portarne due all'esame (vedi pagina web)

- riconoscere i numeri romani
- analisi grammaticale
- riconoscere sequenze di numeri in vari formati

Altro:

- riconosce e separare nomi file testo, file grafici, per ogni file generare un token con
  - tipo file
  - nome
  - estensione

Happy

Strumento per costruire parser, in codice Haskell, simile a Yacc.

- interagisce con lo scanner,
  da sequenza di token, ad albero di derivazione (valore)
- si definiscono delle regole che generano una grammatica
- associa ad ogni regola una funzione che
  - a partire dall'albero di derivazione (valori) delle componenti
  - definisce albero di derivazione (valore) della espressione composta
- similitudini con la funzione `read`,
  (da una stringa di simboli ricostruisce il dato)
  - `read` algoritmo semplice, simile automa a pila, bottom-up, non deterministico,
  - `Happy` si basa sugli automi a pila, bottom-up, deterministico, LALR

Happy

- Strutturo la sequenza di token in un albero
  - vedo la sequenza di token come un espressione
  - formata da un certo insieme di sottoespressioni
  - che a loro volta formate da sottoespressioni
  - fino ai token base
- Associo ad ogni espressione un valore

Definizioni principali nel file Happy

- 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

Esempio

- Parser per una grammatica delle espressioni `exp`.
  - interi,
  - variabili
  - operatori aritmetici `+, -, *, /,`
  - costrutto `let var = exp in exp`.
- Il parser struttura la sequenza di lessemi in un albero di espressioni e sottoespressioni
  associata ad ogni sottoespressione un valore (albero) corrispondente

Struttura del file Happy

- Intestazione - codice da inserire in testa al programma
  - dichiarazione di modulo

```
{
module Main where
import Data.Char
}
```

- Sequenza di dichiarazioni

```
%name calc
%tokentype { Token }
%error { parseError }
```

- definisco il nome della funzione di parsing generata
- il tipo dei Token in input `calc :: [Token] -> T`
- definisco il nome della funzione da chiamare in caso di errore

Struttura del file Happy

- lista dei non terminali,
  - scopo: scrivere le regole usando la sintassi di Yacc
  - definisco la lista dei non terminali della grammatica Happy
  - definisco la loro associazione con token Haskell
    - sinistra: nome usato nelle regole Happy
    - destra: token Haskell, usato nel codice
  - $$ placeholder per il valore associato dal costruttore
  - definizione del tipo token successiva

```
%token
      let             { TokenLet }
      in              { TokenIn }
      int             { TokenInt $$ }
      var             { TokenVar $$ }
      '='             { TokenEq }
      '+'             { TokenPlus }
      '-'             { TokenMinus }
      '*'             { TokenTimes }
      '/'             { TokenDiv }
```

Struttura del file Happy

- Separatore

%%

- Regole della grammatica
  - definisco come espandere (ridurre a) ogni singolo non-terminale
  - più alternative
  - l'azione definisce come costruire l'espressione associate 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.

Struttura del file Happy

```
Exp    : let var '=' Exp in Exp   { Let $2 $4 $6 }
       | Exp1                     { Exp1 $1 }

Exp1   : Exp1 '+' Term            { Plus $1 $3 }
       | Exp1 '-' Term            { Minus $1 $3 }
       | Term                     { Term $1 }

Term   : Term '*' Factor          { Times $1 $3 }
       | Term '/' Factor          { Div $1 $3 }
       | Factor                   { Factor $1 }

Factor
       : int                      { Int $1 }
       | var                      { Var $1 }
       | '(' Exp ')'              { Brack $2 }
```

Struttura del file Happy

- codice ausiliario.
  - definisco la funzione per gestire gli errori

```
{
parseError :: [Token] -> a
parseError _ = error "Parse error"
```

- definisco i tipi dei termini generati dal parser, per i vari non-terminali

```
data Exp
      = Let String Exp Exp
      | Exp1 Exp1
      deriving Show
```

segue

```
data Exp1
      = Plus Exp1 Term
      | Minus Exp1 Term
      | Term Term
      deriving Show

data Term
      = Times Term Factor
      | Div Term Factor
      | Factor Factor
      deriving Show

data Factor
      = Int Int
      | Var String
      | Brack Exp
      deriving Show
```

Struttura del file Happy

- il tipo associato ai token

```
data Token
      = TokenLet
      | TokenIn
      | TokenInt Int
      | TokenVar String
      | TokenEq
      | TokenPlus
      | TokenMinus
      | TokenTimes
      | TokenDiv
      | TokenOB
      | TokenCB
 deriving Show
```

Funzione principale

```
main = do inputString <- getContents
          print ( calc (lexer inputString)
}
```

- `calc :: [Token] -> Exp` generata da Happy
  - `%name calc`
- `lexer :: String -> [Token]` funzione scanner
  - generata tramite Alex
    - importo il modulo relativo, e funzione associata `alexScanTokens`

      `lexer = alexScanTokens`
  - nei casi semplici può essere definita in Haskell, internamente

Definizione diretta di lexer

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
      | isSpace c = lexer cs
      | isAlpha c = lexVar (c:cs)
      | isDigit c = lexNum (c:cs)
lexer ('=':cs) = TokenEq : lexer cs
lexer ('+':cs) = TokenPlus : lexer cs
lexer ('-':cs) = TokenMinus : lexer cs
lexer ('*':cs) = TokenTimes : lexer cs
lexer ('/':cs) = TokenDiv : lexer cs
lexer ('(':cs) = TokenOB : lexer cs
lexer (')':cs) = TokenCB : lexer cs
```

segue

```
lexNum cs = TokenInt (read num) : lexer rest
      where (num,rest) = span isDigit cs

lexVar cs =
   case span isAlpha cs of
      ("let",rest) -> TokenLet : lexer rest
      ("in",rest)  -> TokenIn : lexer rest
      (var,rest)   -> TokenVar var : lexer rest
```

Uso

- creare file `example.y` (stessa suffisso di Yacc)
- comando `happy example.y` genera file `example.hs`
- commando happy example.y -i produce un file il `example.info` contenente informazione dettagliate sul parser.

Valutare la stringa di ingresso

- Al posto di tipo espressione il `calc` può valutare e restituire un intero

```
Term   : Term '*' Factor          { $1 * $3 }
       | Term '/' Factor          { $1 / $3 }
       | Factor                   { $1 }
```

- variabili e costrutto `let` forzano una valutazione più complessa,
  - devo gestire un ambiente, `env` implementato come lista

    `[(String, Int)]`, (nome variabile, intero associato)
  - le regole associano ai non terminali una funzione `env -> Int`

Definizioni

```
Exp   : let var '=' Exp in Exp  { \p -> $6 (($2,$4 p):p) }
      | Exp1                    { $1 }

Exp1  : Exp1 '+' Term           { \p -> $1 p + $3 p }
      | Exp1 '-' Term           { \p -> $1 p - $3 p }
      | Term                    { $1 }

Term  : Term '*' Factor         { \p -> $1 p * $3 p }
      | Term '/' Factor         { \p -> $1 p `div` $3 p }
      | Factor                  { $1 }
```

segue

```
Factor
      : int          { \p -> $1 }
      | var          { \p -> case lookup $1 p of
                                 Nothing -> error "no var"
                                 Just i  -> i }
      | '(' Exp ')'      { $2 }
```

Parsing sequences

Caso tipico, sequenza di comandi

- left recursion, più efficiente, inverte la sequenza

```
prods : prod                    { [$1] }
      | prods prod              { $2 : $1 }
```

- right recursion, usa più spazio

```
prods : prod                    { [$1] }
      | prod prods              { $1 : $2 }
```

Definizione delle precedenze.

- Posso usare grammatiche ambigue
- togliere l'ambiguità definendo un ordine di precedenza tra operatori

```
%right in
%nonassoc '>' '<'
%left '+' '-'
%left '*' '/'
%left NEG
%%
Exp   : let var '=' Exp in Exp   { Let $2 $4 $6 }
      | Exp '+' Exp              { Plus $1 $3 }
      | Exp '-' Exp              { Minus $1 $3 }
      | Exp '*' Exp              { Times $1 $3 }
      | Exp '/' Exp              { Div $1 $3 }
      | '(' Exp ')'              { $2 }
      | '-' Exp %prec NEG        { Negate $2 }
      | int                      { Int $1 }
      | var                      { Var $1 }
```

Esercizi

Due esercizi da portare all'esame (vedi pagina web)

Altro:

- da una stringa di input rappresentante dei numeri floating point (secondo il formato standard, definito in seguito), determinare il valore associato ciascuna sottosequenza, ovvero trasformare sequenze di caratteri in numeri flaoting point.

```
floatnumber:    pointfloat | exponentfloat
pointfloat:     [intpart] fraction | intpart ["."]
exponentfloat:  (nonzerodigit digit* | pointfloat) exponent
intpart:        nonzerodigit digit* | "0"
fraction:       "." digit+
exponent:       ("e"|"E") ["+"|"-"] digit+
```

Type Assignment System

Descrizione formale del sistema di tipi

<–# Avviso

Le Aziende del territorio incontrano gli studenti:

Mercoledì 19 Maggio 2021
Ore 15:00

Link Evento: https://bit.ly/3y4DPhf

Programma

15:00-15:10	**Presentazione del Tirocinio Curriculare**
15:10-15:20	**Electrolux (https://www.electrolux.it/)**
15:20-15:30	**BizAway (https://bizaway.com/)**
15:30-15:40	**Mobile 3D (https://mobile3d.it/)**
15:40-15:50	**Webformat (https://www.webformat.com/)**
15:50-16:00	**MEDarchiver (https://www.medarchiver.com/)**
16:00-16:10	**infoFactory (https://infofactory.it/)**
16:10-16:20	**Domande e Chiusura Lavori**

Per ulteriori informazioni contattare la commissione Tirocini:
Prof. Giuseppe Serra
Prof. Ivan Scagnetto
Prof. Andrea Formisano
Prof. Agostino Dovier

Visita: https://www.dmif.uniud.it/tirocini/tirocini-per-studenti/

Analisi semantica

- Terza fase del compilatore
- Controllo statico sul codice (albero sintattico)
  - estrarre informazioni:
    - symbol table: associo a identificatori tipo, scopo
  - eseguire i controlli non realizzabili con grammatiche liberi dal contesto
    - numero dei parametri procedura
    - identificatore dichiarato prima di essere usa
    - cicli for non modificano la variabile di cilco
  - type checking: il controllo principale

Type system

- Testo di riferimento: articolo di Cardelli, vedi pagina web corso:
  - ripete alcune considerazioni già fatte sui sistemi di tipi
  - alcune differenze nell'uso dei termini:
    - trapped errors errori che causano eccezioni,
      untrapped errors errori che passano innoservati
    - save invece di strongly typed
      nessun untrapped error
    - weakly checked invece di weakly typed
    - static type checking previene alcuni trapped errrors, non tutti
    - dinamic checking invece di dinamic type checking
- Un sistema di tipi viene:
  - descritto informalmente nella documentazione
  - implementato come codice nel compilatore, algoritmo di type checking

Type system

È possibile una definizione formale del controllo di tipi

- attraverso un sistema di regole derivo giudizi nella forma
  $x1 : A1, x2 : A2, ...xn : An \vdash M : A$
  - quando un espressione M ha tipo A,
  - nell'ambiente statico (con le ipotesi),
    $x1 : A1, x2 : A2, ...xn : An$
    che assegna un tipo alle variabili in M,
- Altri giudizi ausiliari
  - $x1 : A1, x2 : A2, ...xn : An \vdash A$
    A è un espressione di tipo corretta nella ambiente
    $x1 : A1, x2 : A2, ...xn : An$
  - $x1 : A1, x2 : A2, ...xn : An \vdash \cdot$
    l'ambiente è ben formato.
- Per i linguaggi che considereremo questi giudizi non necessari, posso definire tipi e ambienti corretti attraverso grammatiche libere

Regole di derivazione

- simili alla deduzione naturale, o calcolo dei sequenti
- un insieme di giudizi premessa, porta ad un giudizio conclusione

```
Gamma1 |- M1 : A1   ...   GammaN |- Mn : An
-----------------------------------------
                Gamma |- M : A
```

- regole senza premesse sono gli assiomi
- regole date per induzione sulla struttura di `M`
  - ad ogni costrutto del linguaggio si associa una regola (qualche eccezione)
- all'arricchirsi del linguaggio, aumentano le regole,
  approccio modulare:
  - singole regole restano valide, all'arricchirsi del linguaggio
  - regole piuttosto semplici,
    l'analisi del sistema di derivazione più complessa

Derivazioni

- hanno una struttura ad albero,
  - da un insieme di assiomi: foglie
  - derivo un giudizio finale: radice
- Permettono di derivare i giudizi validi

Sistemi definiti mediante regole di derivazione sono frequenti:

- in logica
- nella descrizione formale dei linguaggi:
  - sistema di tipi
  - semantica operazionale

Type checking, type inference

Dato un sistema di tipi, considero diversi problemi:

- problema di controllo di tipo, type checking,
  - dato un termine `M`,
  - un tipo `A`
  - un ambiente `Gamma`
  - determinare se `Gamma |- M : A` sia valido
- problema di inferenza di tipo, type inference,
  - dato un termine `M` e un ambiente `Gamma`
  - trovare un tipo A tale che `Gamma |- M : A` sia valido
- l'analisi semantica risolve problemi di type inference:
  - nel codice definito il tipo delle variabili
  - necessario inferire il tipo delle (sotto)-espressioni
  - in Python, Haskell possibile non dichiarare il tipi delle variabili: necessario inferire anche l'ambiente
  - l'inferenza di tipo può essere un problema complesso

Type soundness

Errore di tipo, in un espressione `M`

- non esiste `A` tale che `Gamma |- M : A`
  - definizione indiretta

Desiderata: alcuni tipi di trapped error, non tutti, non sono generati da programmi che superano il controllo di tipo

- Per dimostrare la type soundness necessario collegare computazione e type system,
- possibile formulazione del collegamento:
  - se `|- M : A` e
  - `M` riduce a `N` (viene trasformato dalla computazione in `N`)
  - allora `|- N : A`
  - ossia, la computazione preserva i tipi
- dimostrando che nessun programma tipato genera una certa classe di errori
- si ha la garanzia formale dell'assenza di quegli errori nella computazione

Esempi sistemi di tipi

Nel seguito presentiamo, in maniera incrementale

- semplici linguaggi di programmazione
- relative regole di tipo.

Regole illustrative:

- linguaggi diversi usano regole diverse, simili nello spirito,

Regole singolarmente semplici ma sistema di tipi complesso,

- numero di regole elevato
- piccole modifiche, ad una singola regola, possono portare ad inconsistenze

Sistemi di tipi al prim'ordine.

- Prim'ordine, perché nessuna variabile di tipo
  - non ci sono i tipi polimorfi
- Presentiamo alcuni esempi
  - linguaggio base
  - un insieme di estensioni
  - ogni estensione richiede nuove regole
  - le regole precedenti vengono preservate struttura modulare.

Sistema F1, (lambda calcolo tipato semplice)

- quasi un frammento minimo di Haskell,
  ma tipi espliciti, nessun polimorfismo
- Tipi `... A, B`
  - un insieme di tipi base `K in Basic`
  - tipi funzioni `A -> B`
- Termini, codice `... M, N`
  - variabili `x`
  - costanti `c : K`
  - funzioni `\ x:A . M`
  - applicazioni `M N`
- si associa un tipo al parametro formale di ogni funzione
  - diversamente da Haskell
  - similmente a quasi tutti gli altri linguaggi

Regole

- ovvie per i giudizi di ‘buona formazione’ e ‘buon tipo’,
  - definibili mediante grammatiche libere
- (Var)

```
G, x:A, G' |- x:A
```

- (Fun)

```
     G, x:A |- M:B
------------------------
G |- (\ x:A . M) : A->B
```

- (App)

```
G |- M : A->B      G |- N:A
--------------------------
    G |-  MN :  B
```

Esempio type inference per

```
\ f : A->A . ( \ x:A . f(f x) )
```

Costruzione di una derivazione

- Costruzione top-down
  a partire da un termine da tipare - seleziono la regola da applicare - riduco il problema a quello di tipare i sottotermini
- Ad ogni passo due problemi:
  - selezionare la regola (schema di regole) da applicare
    sufficiente osservare il costrutto principale del termine
  - istanziare lo schema di regole
    - regole generiche, fanno riferimento a generici termini, tipi
    - da istanziare nel caso particolare
    - simili a meccanismi di pattern matching

Tipi base e costanti

Ad ogni costante si associa una regola che ne assegna tipo

Tipo base semplice con un unico elemento

- (unit)

```
G |- unit : Unit
```

In Haskell: enupla vuota `() : ()`

Booleani

Costanti - (true) (false)

```
G |- true : Bool      G |- false : Bool
```

Funzioni collegate: - (ite)

```
 G |- (if_A _ then _ else _) : Bool -> A -> A -> A
```

o alternativamente

```
G |- M : Bool   G |- N1 : A  G |- N2 : A
-----------------------------------------
   G |- if_A M then N1 else N2 : A
```

Marcare il costrutto `if_A` con il tipo `A` semplifica l'Inferenza altrimenti più complessa

Naturali

linguaggio minimale

```
G |- 0 : Nat        G |- succ : Nat -> Nat

G |- pred : Nat -> Nat        G |- isZero : Nat -> Bool
```

estensioni

- insieme infinito di regole per infinite constanti

```
G |- 1 : Nat      G |- 2 : Nat      G |- 3 : Nat   ...
```

- operazioni aritmetiche

```
G |- N1 : Nat  G |- N2 : Nat
---------------------------
   G |- N1 + N2 : Nat
```

Naturali

O in alternativa:

```
G |- (+) : Nat -> Nat -> Nat
```

Esempio

```
\ y:Nat . \ z:Nat . (y + 2) + z
```

Tipi prodotto, coppia

- nuovo costruttore di tipi `A * B`
  in Haskell `(A,B)`

Regole

- (Pair)

```
G |- M : A      G |- N : B
--------------------------
     G |- (M,N) : A*B
```

- (First) (Second)

```
G |- first : A*B -> A      G |- second : A*B -> B
```

Esempio

```
\ y : (Int*Int) . (first y) + (second y)
```

Tipi unione

- nuovo costruttore di tipi `A + B`

Regole

- (InLeft), (InRight)

```
G |- inLeft : A -> A+B      G |- inRight : B -> A+B
```

- (IsLeft), (IsRight)

```
G |- isLeft : A+B -> Bool      G |- isRight : A+B -> Bool
```

- (AsLeft), (AsRight)

```
G |- asLeft : A+B -> A      G |- asRight : A+B -> B
```

possono causare errore di tipo a tempo di esecuzione
forzano un type checking dinamico (dynamic checking)

segue

È possibile evitare il type checking dinamico con il costrutto (case)

```
G |- M : A1 + A2     G, x1:A1 |- N1:B     G, x2:A2 |- N2:B
-------------------------------------------------------------
G |- case M of(inLeft(x1:A1) -> N1)(inRight(x2:A2) -> N2)  : B
```

- sintassi alternativa [Cardelli]
  - `case M of (inLeft(x1:A1) -> N1) (inRight(x2:A2) -> N2) : B`
    scritto come:
    `case M of x1:A1 then N1 | x2:A2 then N2`

Record (Struct)

- Estensione del tipo prodotto
  - un numero arbitrario di componenti
  - sistema di etichette `l`
- nuovo costruttore di tipo `{ l1 : A1, ... , ln : An }`

regole

- (Record)

```
G |- M1 : A1      G |- M2:A2     ...     G |- Mn:An
-----------------------------------------------------
G |- { l1:M1, ..., ln:Mn } : { l1:A1, ..., ln:An }
```

- (Record Select)

```
G |- M : { l1 : A1, ... , ln : An }
-----------------------------------
        G |-  M.li : Ai
```

segue

Un alternativa, poco usata, a (Record Select)

- (Record With)

```
G |- M:{l1:A1, ..., ln:An}      G, x1:A1,... , xn:An |- N : B
-------------------------------------------------------------
     G |- case M of {x1:A1,... , xn:An} -> N : B
```

Variant type

- Estensione del tipo unione
  - un numero arbitrario di componenti
  - sistema di etichette `l`

Reference type

- definiscono locazioni di memoria
- introducono la memoria, lo stato
- distinguo in maniera esplicita
  - la locazione di memoria
  - dal suo contenuto

  separazione sfumata in altri linguaggi
- costrutto simile in ML
- costruttore di tipo `Ref A`

Costrutti e regole

- (Ref)

```
  G |- M : A
-----------------
G |- ref M : Ref A
```

`Ref M` definisce una nuova locazione, inizializzata con valore `M`

- (Deref) accedo al contenuto

```
G |- deref : (Ref A) -> A
```

`deref` corrisponde a `!` in ML `!`, o `*` in C

- (Assign)

```
G |- M : Ref A      G |- N : A
-------------------------------
   G |- M = N  :  Unit
```

Linguaggio imperativo dentro uno funzionale

- posso tradurre

```
var x = M;
N
```

- con

```
let x = ref M in N
```

- oppure con

```
(\ x . N) (ref M)
```

- posso tradurre la composizione di comandi `C1; C2` con

```
`(\ y : Unit . C2) C1`
```

in un linguaggio call-by-value

Linguaggio imperativo dentro uno funzionale

In generale

- posso tradurre un linguaggio imperativo in uno funzionale + Ref

E interessante analizzare

- quali costrutti di un linguaggio posso essere derivati dagli altri
- definibili come zucchero sintattico
- regole di tipo e semantica definibili mediante traduzione

Esempio: la regola per Composition derivabile

- (Composition)

```
G |- C1 : Unit      G |- C2 : Unit
-------------------------------
   G |- C1; C2  :  Unit

(C1;C2)  ::=  (\ y : Unit . C2) C1
```

Esempi

```
var x = 3;
    x = (deref x) + 1

(\ x :(Ref Nat) .  x = (deref x) + 1) (ref 3)
```

Esempi

```
var x = 3;
    x = x + 1;
    x = x + 2;

(\ x :(Ref Nat) .  ((\y : Unit .   x = (deref x) + 2)
    (x := (deref x) + 1)) (ref 3)
```

Tipi di dati ricorsivi

Il tipo `data` di Haskell ne sono un esempio.

- Introduco variabili di tipo `Y`
- Un funzione di punto fisso `mu` sui tipi

  `mu Y.A`
- `mu Y.A` denota un tipo isomorfo a

  `A [ mu Y. A / Y]`
- Analogia con Haskell
  - liste di naturali

    ```
    data ListInt = Nil | Cons Int ListInt`
    ```
  - diventa

    ```
    mu Y . Unit + (Int * Y)
    ```
  - isomorfo a

    ```
    Unit + (Int * (mu Y . Unit + (Int * Y))
    ```

Fold, unfold

- Per costruire valori di tipo ricorsivo, si usa una famiglia di costruttori

```
fold_(mu Y.A)
```

```
uno per ogni tipo ricorsivo.
```

- Regola (Fold)

```
G |- M : A [ mu Y. A / Y]
-------------------------
G |- fold  M : mu Y . A
```

```
per brevità ometto l'indice di tipo
```

- Per esaminare valori di tipo ricorsivo, si usa una famiglia di distruttori

```
unfold_(mu Y.A)
```

Regole

- (Unfold)

```
      G |- M :  mu Y. A
-------------------------------
G |- unfold M : A [ mu X. A / Y]
```

Esempio:

- i costruttori di liste si possono definire come:

```
Nil  =  fold(inLeft unit)
Cons x xs =  fold(inRight (x,xs))
```

- i distruttori di liste

```
head xs =  first (asRight (unfold xs))
tail xs =  second (asRight (unfold xs))
```

- i costrutti (`mu, fold, unfold`) sono:
  concettualmente semplici, ma poco pratici, i costrutti standard modellabili attraverso di essi
- Tutti i dati ricorsivi, non mutuamente ricorsivi e non parametrici, di Haskell, trattabili in questo modo.
- codifica non troppo complessa

Esempio

```
Nil  =  fold(inLeft unit)
```

Esempio

```
as: ListInt |- head as : Int

as : mu Y . Unit + (Int * Y)
   |-  first (asRight (unfold as)) : Int `
```

Linguaggio imperativo, simil C.

Considero tre categorie sintattiche

- Espressioni

```
E ::= const  |  id  |  E binop E  |  unop E
```

- Comandi

```
C ::= id = E  |  C; C  |  while E {C}  |
            if E then C else C  |  I(E, ... E)  |  {D ; C}
```

- Dichiarazione

```
D  ::=  A id = E  |  id(A1 id1, ... , An idn) { C }  |
            epsilon  |  D; D
```

Regole, espressioni

Per le espressione, valgono le corrispondenti regole del linguaggio funzionale

- (Id, (Var))

```
G, id:A, G' |- id : A
```

```
G |- true : Bool      G |- false : Bool
```

- (ite)

```
 G |- (if _ then _ else _) : Bool -> A -> A -> A
```

```
G |- 1 : Nat      G |- 2 : Nat      G |- 3 : Nat  ...
```

- operazioni aritmetiche

```
G |-  E1 : Nat      G |- E2 : Nat
------------------------------
    G |-  E1 + E2 : Nat
```

Comandi

- (Assign)

```
G |- id : A      G |- E : A
--------------------------
   G |- id = E  :  Unit
```

- (Sequence)

```
G |- C1 : Unit  G |- C2 : Unit
------------------------------
  G |- C1; C2  :  Unit
```

- (While)

```
G |- E : Bool      G |- C : Unit
--------------------------------
   G |- while E {C}  :  Unit
```

segue

- (If Then Else)

```
G |- E : Bool     G |- C1 : Unit  G |- C2 : Unit
-----------------------------------------------
   G |- if E then C1 else C2  :  Unit
```

- (Procedure)

```
G |- id : (A1 * ... * An) -> Unit    G |- E1 : A1 ...  G |- En
-----------------------------------------------------------
   G |- id (E1, ..., En) : Unit
```

- (Blocco)

```
G |- D :: G1     G, G1 |- C : Unit
---------------------------------
   G |- {D;C} : Unit
```

Dichiarazioni

Le dichiarazione necessitano di un nuovo tipo di giudizio

- Una dichiarazione crea un ambiente, che viene utilizzato nel blocco della dichiarazione
- Giudizi nella forma

```
G |- D :: G1
```

Regole

- (Id)

```
G |-  E : A      (A tipo memorizzabile)
-------------------------------------
    G |-  A id = E  ::  (id : A)
```

- (Proc)

```
G, id1 : A1, ..., idn : An |-  C : Unit
-----------------------------------------------------
G |- id(A1 id1, ... idn){ C } :: id : (A1 x...x  An) -> Unit
```

- (Recursive Proc)

```
G, id1 : A1, ... An, id : (A1 *...* An) -> Unit  |-  C : Unit
-----------------------------------------------------------------
G |- id(A1 id1, ... idn){ C } :: id : (A1 x...x  An) -> Unit
```

segue

- (Sequenza)

```
G  |-  D1 :: G1          G, G1 |-  D2 :: G2
-----------------------------------------
        G |-  D1; D2 :: G1, G2
```

Array

- Devo formalizzare le due differente interpretazioni delle espressioni
  - a sinistra della assegnazione '='
  - a destra dell'assegnazione
- finora a sinistra solo identificatori 'id'

Tipi, aggiungo un costruttore di tipo

- `A[B]` con le opportune restrizioni
  - `A` a memorizzabile
  - `B` tipo enumerazione

Espressioni, distinguo tra espressioni sinistre e destre

- `LE ::= id | LE[RE]`
- `RE ::= LE | const | RE binop RE | unop RE`

Una nuova versione dell'assegnamento

- `C ::= LE = RE | ...`

Regole

Per le espressioni distinguo due giudizi,

- `G |-l E : A` (`E` denota una locazione di tipo `A`)
- `G |-r E : A` (`E` denota un valore di tipo `A`)
- (Assign)

```
G |-l  E1 : A      G |-r  E2 : A
-------------------------------
G |-l  E1 = E2 : Unit
```

- (Left-Right)

```
G |-l  E : A
-------------
G |-r  E : A
```

- (Var)

```
G, x:A, G' |-l x : A
```

segue

- (Array)

```
G |-l  E : A[B]      G |-r  E1 : B
-------------------------------
G |-l  E[E1] : A
```

- Dichiarazione

```
-------------------------------
G |-  id : A[B]  ::  id : A[B]
```

- Le restanti regole per le espressioni restano inalterate, diventando regole per giudizi right `|-r`.

Polimorfismo di sottotipo.

Maggiore flessibilità permetto ad un espressione di avere più tipi

- Varie forme di polimorfismo
  - ad hoc
  - di sottotipo
  - parametrico
  - combinazioni dei precedenti
- Sottotipo (e ad hoc)
  - introduco una relazione di sottotipo, con relativi giudizi e regole

```
G |- A <: B
```

- (Subsumtion)

```
G |-r  E : A     G |-  A <: B
-----------------------------
       G |-r  E : B
```

Regole per i giudizi di sottotipo

Posso trattare il polimorfismo ad-hoc, con assiomi del tipo.

```
G |-  Int <: Float
```

Polimorfismo dei linguaggi ad oggetti:

- un tipo oggetto con più campi, sottoggetto di uno con meno.
- formalizziamolo con i tipi Record

```
G |- A1 <: B1 ...      G |- Am <:Bm      m < n
-----------------------------------------------
G |- { l1:A1, ..., ln:An } <: { l1:B1, ..., lm:Bm}
```

segue

Definisco regole di sottotipo per ogni costruttore di tipi.

- (Prod <:)

```
G |- A1 <: B1       G |- A2 <:B2
-------------------------------
G |- A1 x A2  <:  B1 x B2
```

Regole sempre covariante con eccezione

- (Arrow <:)

```
G |- A1 <: B1       G |- A2 <:B2
-------------------------------
G |- B1 -> A2  <:  A1 -> B2
```

`controvariante sul primo argomento.`

Regola non ovvia, tipi ricorsivi.

- (Mu <:)

```
G, X <: Y  |-  A <: B
-------------------------
G |- mu X. A  <:  mu Y . B
```

la regola

- (Mu <: Wrong)

```
G, X  |-  A <: B
-------------------------
G |- mu X. A  <:  mu X . B
```

mi permette di derivare

```
G |- mu X. X -> Int  <:  mu X . X ->  Float
```

non corretto.

Esempi

```
{ V : Int[Int]; V[0] = 0  ; V[V[0]] = V[0]}
```

Programmazione concorrente

Programmazione concorrente

Non un diverso paradigma di programmazione, ma
una feature aggiuntiva a paradigmi preesistenti

- Programmazione concorrente all'interno di
  - linguaggi imperati, oggetti,
  - logici, funzionali (avvantaggiati dalla mancanza di stato)
    - Erlang
    - Haskell
- Argomento vasto, implementato in molti modi diversi
  tenteremo un catalogazione dei vari aspetti della programmazione concorrente

Motivazioni

- sfruttare l'evoluzione dell'hardware:
  - negli ultimi anni i miglioramenti della prestazioni ottenuti principalmente
    con aumento parallelismo (numero dei core)
- alcuni problemi si descrivono meglio in maniera concorrente
  divido un compito in sottocompiti da svolgere in parallelo e in maniera indipendente
  - esempio: un programma browser
  - un thread per ogni parte della pagina da visualizzare
  - le immagini non bloccano la visualizzazione del testo
- gestire hardware distribuito:
  - applicazione in esecuzione su più calcolatori collegati via web
  - sistema di controllo di un impianto

Concorrenza a vari livelli nelle macchine virtuali

Concorrenza fisica, a livello di macchina fisica:

- esecuzione simultanea di istruzioni
- diversi, modi, gradi di granularità:
  - pipeline
  - superscalari (parallelismo a livello di istruzione)
  - istruzioni vettoriali
  - multicore (parallelismo a livello di processi)
  - GPU (computazione vettoriale (SIMD)
  - multicomputer
  - reti di calcolatori

Concorrenza logica.

A livello di linguaggi di programmazione

- programmazione parallela
  - si cerca di parallelizzare l'esecuzione di un singolo problema
    calcolo scientifico, algoritmi di simulazione
  - algoritmi paralleli
- programmazione multithreaded:
  - più thread o processi attivi contemporaneamente e che girano su una stessa macchina fisica
  - Es: thread in Java
  - Modello della memoria:
    - a memoria condivisa
    - a scambio di messaggi
- programmazione distribuita:
  - programmi concorrenti, eseguiti su macchine separate
  - es: multicomputer con memoria distribuita oppure
    reti di calcolatori con varie architetture e topologie
  - non si assume la memoria condivisa

SOC e Cloud computing

esempi di programmazione distribuita, con una precisa struttura

- SOC (Service Oriented Computing): paradigma di programmazione basata sulla composizione di componenti (servizi) che interagiscono
  - caso tipico: web services
  - calcolatori mettono a disposizioni funzionalità, servizi
  - possibilità di comporre più servizi semplici, in servizi più complessi
  - occorrono
    - cataloghi dei servizi disponibili
    - opportune primitive di comunicazione
  - linguaggi specifici: BPEL, JOLIE, ...

Cloud computing:

- stesse idee dei SOC ma meno strutturate, con una granularità molto più fine
  - anche servizi elementari (spazio disco, CPU ecc.)
- insieme di tecnologie (infrastrutture, modelli di comunicazione ecc.) che permettono lo sviluppo di applicazioni distribuite su scala planetaria
- accesso on demand, semplice, a risorse configurabili distribuite sulla rete
- problemi di sicurezza

Separazione tra i vari livelli di parallelismo

Parallelismo logico e fisico non corrispondono,

esempi:

- pipeline, superscalari: parallelismo sono fisico, invisibile a linguaggio di programmazione
- un programma multithread può essere eseguito in vari modi casi possibili:
  - ogni thread logico eseguito su core distinto
  - tutti i thread eseguiti su di un unico core (interleaving)
  - molti core ciascuno con molti thread in esecuzione
  - . . .
- distinguibili solo in termini di prestazione,
  stesso insieme di possibili risultati

Metodi per la programmazione concorrente

- parallelismo attraverso chiamate di libreria:
  - uso un linguaggio sequenziale,
  - lancio threads con chiamate a funzioni di libreria
    funzione che riceve come argomento il codice da eseguire in parallelo
  - es C e POSIX thread
- estensioni, supportate dal compilatore, di un linguaggio sequenziale
  - Fortran e OpenPM, con direttive al compilatore (pragma)
- linguaggi di programmazione con costrutti per la concorrenza

Aspetti programmazione concorrente

Molti modi diversi di strutturare la programmazione concorrente.

In tutti esistono più thread (esecuzioni attive)
si distingue su come avviene:

- la comunicazione: i thread si scambiano informazioni
- la sincronizzazione: i thread regolano la velocità relativa
  - attesa attiva
  - scheduler (rilascio della CPU)

Altri aspetti:

- come avviene la creazione di nuovi thread

Thread e processi

- Thread (del controllo): l'esecuzione di una sequenza di comandi, una specifica computazione,
  con spazio di indirizzamento comune
- Processo (pesante): generico insieme di istruzioni in esecuzione, con il proprio spazio di indirizzamento.

  un processo può essere costituito da più thread diversi
- Terminologia spesso non univoca (thread, processi leggeri, task..)

Creazione di nuovi thread:

- fork/join:
  primitive per lanciare un nuovo processo e concludere l'esecuzione
- co-begin:

```
co-begin
    stm_1
    ...
    stm_n
end
```

- parallel loops: vengono definiti cicli `for` con indicazione al compilatore (#pragma) di poter eseguire in parallelo i diversi loop
- early replay:
  una procedura restituisce il controllo al chiamante prima della sua conclusione, chiamato e chiamante in esecuzione contemporanea
- ...

Meccanismi di comunicazione

Memoria condivisa:

- accesso in lettura/scrittura ad una zona comune di memoria
- presuppone l'esistenza di una memoria comune (almeno concettualmente)
  - possibili differenze tra livello logico e fisico
    memoria comune simulata a livello di sistema operativo, librerie

Scambio di messaggi:

- uso di primitive esplicite di invio (send) e ricezione (receive) di messaggi
- un'opportuna struttura di comunicazione (canale) deve fornire un percorso logico fra mittente (send) e destinatario (receive)

Blackboard: modello intermedio fra i precedenti.

- i processi condividono una stessa zona di memoria (blackboard).
- comunicazione mediante send e receive al/dal blackboard

Meccanismi di sincronizzazione

- Sono i meccanismi che permettono di controllare l'ordine relativo delle varie attività dei processi.
- Essenziali per la correttezza, race condition.
- Esempio: due processi devono incrementare una variabile condivisa

  processo P1 in marcato con +, processo P2 non marcato.

```
+   leggi x in reg1;
    leggi x in reg2;
    incrementa reg2;
    scrivi reg2 in x;
+   incrementa reg1;
+   scrivi reg1 in x;
```

`x` è incrementata solo di 1

Race condition

- Race condition non necessariamente negative
  - programmi concorrenti intrinsecamente non-deterministici
    - diverso ordine di esecuzioni possibili
    - spesso le varie alternative sono accettabili
    - qualcuna va esclusa

Meccanismi di sincronizzazione

Con memoria condivisa:

mutua esclusione: dati, regioni critiche del codice, non sono accessibili contemporaneamente a più processi

sincronizzazione su condizione: si sospende l'esecuzione di un processo fino al verificarsi di una opportuna condizione

Con scambio di messaggi

- di solito impliciti nelle primitive di send e receive
  posso ricevere un messaggio solo dopo il suo invio

Come implementare la sincronizzazione:

- attesa attiva (busy waiting o spinning)
  - ha senso solo su multiprocessori
- sincronizzazione basata sullo scheduler
  (auto-)sospensione del thread

Mutua esclusione mediante attesa attiva: lock

- Utile un'operazione test-and-set(B) atomica, non interrompibile altrimenti sono necessari algoritmi più complessi

  - `in programmazione concorrente è importante definire le a`

Struttura processo:

```
process Pi {
        sezione non critica;
        acquisisci_lock(B);
        sezione critica;
        rilascia_lock(B);
        sezione non critica;
    }
```

Mutua esclusione mediante attesa attiva: lock

- Acquisizione lock:
  - lock variabile booleana
  - `true` bloccata, `false` libera
  - si sfrutta un istruzione `read and write` atomica
    - `test and set`
    - possibile anche senza `test and set`
    - algoritmi più complessi, diverse alternative

```
void acquisisci_lock(ref B: bool) {
    while test_and_set(B) do skip;
}
```

- Rilascio lock

```
void rilascia_lock(ref B: bool) {
            B = false;
}
```

Algoritmi alternativi

- Algoritmi più complessi che non usano `test_and_set`
- Problemi di fairness
  - algoritmo precedente non assicura la fairness
  - nel caso di più richieste, l'assegnazione fatta arbitrariamente
  - un processo potrebbe dover attendere all'infinito
  - algoritmi più complessi assicurano la fairness
- Conention per l'uso della memoria e dei bus nei sistemi multiprocessori:
  - un processo in attesa, usa continuamente alla memoria, risorsa condivisa,
  - la versione test and test_and_set(B) migliora le prestazioni,

```
void acquisisci_lock(ref B: bool) {
    while B do skip;
    while test_and_set(B) do skip;
}
```

Sincronizzazione su condizione: barriere

Meccanismo di sincronizzazione

- processo in attesa che una condizione globale non venga soddisfatta.
  - Esempio:

    un compito comune suddiviso tra più processi

    ogni processo attende che tutti gli altri terminato prima di proseguire allo stadio successivo

    possibile implementazione,

    - `contatore condiviso`
    - `conta il numero di processi che devono terminare`
    - `uso sezioni critiche per aggiornare il contatore`

Attesa attiva: problemi

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
  - conveniente per attese brevi, ma inefficiente su attese lunghe
  - poco sensato in sistemi singolo processore

Sincronizzazione basata sullo scheduler

- Il processo che deve essere posto in attesa, rilascia la CPU
- I processi sospesi sono gestiti dallo scheduler dei processi (nucleo s.o.)

Possibili costrutti per la sincronizzazione con scheduler

- semafori
- monitor

Semafori

- Semaforo: tipo di dato con
  - l'insieme dei valori, costituito dai numeri interi >=0
  - due operazioni atomiche chiamate P e V

    `sem s`
- P(s): accesso semaforo
  - se s > 0 s decrementata (operazione atomica)
  - se s = 0 processo sospeso
- V(s): rilascio semaforo
  - s incrementata (op. atomica)
  - si attiva lo sblocco di eventuali processi in attesa
    - fairness
    - FIFO

Esempio di uso dei semafori: filosofi a cena

Problema si sincronizzazione:

- proposto da Dijkstra - Hoare
- illustra la possibilità di deadlock
  - tutti i processi in attese di un evento
  - sistema bloccato

Esempio: filosofi a cena

Soluzione:

- le forchette in posizione dispari richieste per prime
- un volta ottenuta una forchetta dispari, un filoso può essere bloccato solo da un filosofo mangiante
- problema: se tutti fanno richiesta, quale percentuale di filosofi riesce sicuramente a mangiare?

Modelizzazione:

- filosofi – processi
- forchette sono risorse condivise
  - mutua esclusione gestita tramite semafori

Esempio: filosofi a cena

```
sem forchette[5];
for (i=1, i<=5, i+=1) {forchette[i] = 1};

process Filosofo[i]{        %  i = 1, 3, 5
    while true {
        pensa;
        P(forchette[i]);    // acquisisce forchetta di destra
        P(forchette[i-1])   // acquisisce forchetta di sinistra
        mangia;
        V(forchette[i]);     // rilascia forchetta di destra
        V(forchette[i-1])    // rilascia forchetta di sinistra
    }
}
```

Esempio: filosofi a cena

```
process Filosofo[i]{        %  i = 2 , 4
    while true {
        pensa;
        P(forchette[i-1]);  // acquisisce forchetta di sinistra
        P(forchette[i])     // acquisisce forchetta di destra
        mangia;
        V(forchette[i-1]);  // rilascia forchetta di sinistra
        V(forchette[i])     // rilascia forchetta di destra
    }
}

Filosofo[1] || Filosofo[2] || Filosofo[3] ||
  Filosofo[4] || Filosofo[5]
```

Monitor

- offrono un livello di astrazione e strutturazione maggiore dei semafori:
  - semplici contatori condivisi
  - la gestione corretta di risorse condivise affidata al programmatore
- oggetto (classe, modulo) condiviso tra più thread,
  - contiene al suo interno le risorse
  - procedure pubbliche per l'uso di risorse condivise, non si accede alle risorse direttamente
  - variabili permanenti (realizzano lo stato);
  - procedure
  - `init` procedure di inizializzazione
- monitor: oggetti thread-safe
  - la mutua esclusione è garantita, un solo thread alla volta può avere accesso ai metodi
  - non necessariamente vero per oggetti generici
- terminologia
  - monitor componenti passivi (eseguiti se invocati)
  - thread componenti attivi

Variabili condizionali

- Componente aggiuntive al monitor,
  rappresentano lo disponibilità, o meno, di una risorsa condivisa
- permetto 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
- più thread all'interno del monitor
  - al più uno attivo
  - gli altri in attesa
- esempio:
  - monitor gestisce una coda di lavoro
  - thread consumer: prelevano elementi
  - thread producer: inseriscono elementi
  - sospesi in caso di coda piena o vuota

Costrutti per variabili condizionali

- dichiarazione

  `cond namevar`

vista come coda dei thread in attesa

- interrogazione

  `empty(namevar)`

- sospensione

  `wait(namevar)`

nel caso risorsa non disponibile, thread in attesa,
altri thread possono ‘entrare’ nel monito

Costrutti per variabili condizionali

- rilascio, risveglia un processo in attesa

  `signal(namevar)`

Due alternative:

- segnala e continua
- segnala e sospendi

Comunicazione mediante scambio di messaggi

Esistono due tipi di comunicazione,
con meccanismi di sincronizzazione diversi:

- sincrona: invio e ricezione di un messaggio allo stesso tempo
  - send e receive (concettualmente) allo stesso tempo
  - send (e receive) bloccante
  - più semplice da implementare
    coda canale di lunghezza uno
  - errori gestiti immediatamente
  - più difficile evitare i deadlock, (deadlock prone)
- asincrona: invio e ricezione di un messaggio in momenti diversi
  - send e receive i momenti diversi
  - send non bloccante,
    - necessario implementare una coda di messagggi inviati
  - receive bloccante
    - l'invio di messaggi risveglia (potenzialmente) processi

Mutua implementazione

Possibile simulare la comunicazione sincrona usando quella asincrona

- dopo aver spedito il messaggio si attende (receive)
  un acknowledgement da parte del ricevente

E viceversa

- si crea un processo buffer intermedio,
  - riceve messaggi dal mittente, li inserisce in un coda
  - contemporaneamente, cerca di spedirli al ricevente

Meccanismi di naming

- nomi espliciti di processi
  - calcolo CSP, processi memoria separata
  - primitive di comunicazione
    - `Proc1 ! dato` invio dato
    - `Proc2 ? variabile` lettera data
- porte
  - meccanismo usato nelle reti protocollo TCP
    - si definiscono degli identificatore di porta
    - ogni porta associata ad un tipo di comunicazione, servizio
    - si specifica (indirizzo IP) - numero porta (numero di porta)
    - stessa macchina fisica - più connessioni simboliche
  - usato anche in linguaggi di programmazione,

Esempio in ADA

- definite nelle interfacce di un processo `task`

```
task  TypeTask is
    entry portaIn (dato : in integer);
    entry portaOut (dato : out integer);
end TypeTask
```

- usate all'interno del codice

```
task body TypeTask is
    ...
    accept portaIn(dato : in integer) do ... end portaInf
    ...
    accept portaOut(dato : out integer) do ... end portaInf
    ...
end TypeTask
```

- posso definire un processo `Pr` di tipo `TypeTask`
- inviarli dati con il comando `Pr.portaIn(mioDato)`

Meccanismi di naming

- canali
  - simili alle porte ma più generale astratto
  - concettualmente qualsiasi meccanismo di comunicazione tra processi
    - non si fanno ipotesi sul numero di processi coinvolti, o sulla direzione dei dati

Meccanismi di uso:

Dichiarazione

```
channel nomeCh (Type)
```

- processi che condividono un canale posso comunicare tra loro
- nel π-calcolo il nome di un canale può essere scambiato tra processi
  - cambia la geometria delle comunicazione

segue

Invio dati

```
send NomeCanale(dati)
```

- aggiunge un messaggio alla coda del canale
- bloccante solo nella comunicazione sincrona

Ricezione

```
receive NomeCanale(var)
```

- preleva un messaggio dalla coda del canale
- sempre bloccante (caso della coda è vuota)

Più tipi di canali

Verso:

- monodirezionali,
- bidirezionali

Numero di connessioni

- link (un destinatario - un mittente)
- input port (più mittenti - un destinatario)
- mailbox (più mittenti - più destinatari)

Sincronizzazione

- sincroni
- asincroni

Esempio (interazione client server)

```
channel richiesta (int client, char dati);
channel risposta1 (char ris);
channel risposta2 (char ris);

process Client1{
    char valori;
    char risultati;
    ....                    // Definizione dei valori
    send richiesta(1, valori);
    receive risposta1(risultati};
    ....                    // Uso dei risultati
}
process Client2{
    char valori;
    char risultati;
    ....                    // Definizione dei valori
    send richiesta(2, valori);
```

Esempio (interazione client server)

```
process Server{
    int cliente;
    char valori;
    char risultati;
    ....                    // Inizializzazione
    while true {
        receive richiesta (cliente,  valori);
        if cliente = 1 then
            {   ... // Elaborazione valori
             send risposta1(risultati);
             }
            if cliente = 2 then
            {   ... // Elaborazione valori
             send risposta2(risultati);
             }
    }
}
```

Esempio

Alternative:

- passare il nome di un canale
  - al posto dell'identificativo di processo
  - permette comunicazione privata: il client invia un nome privato di canale su cui ricevere la risposta,
  - permette al server di gestire un numero arbitrario di processi, in modo semplice.
- receive bloccante, per evitare i blocchi:
  - primitive per controllare lo stato della coda di input

RPC

Ulteriore meccanismo di comunicazione:

- sistemi distribuiti,
- client-server

Si invoca su una macchina remota una procedura:

- comunicazione: passaggio dei parametri, risultato
- meccanismi per rendere pubblici i nomi, parametri delle procedure remote
  - tipicamente moduli
  - `call NomeModulo.NomeProc (parametriAttuali)`
- nella macchina remota meccanismi per attivare un processo all'arrivo della richiesta
- disponibile in Java
- realizzabile tramite stub

Rendez-vous

- nel rendez-vous esiste già un processo attivo sul destinatario che gestisce la risposta
  - disponibile in Ada, esempio ‘port’ precedente

```
task body TypeTask is
    ...
    accept portaIn(dato : in integer) do ... end portaInf
    ...
    accept portaOut(dato : out integer) do ... end portaInf
    ...
end TypeTask
```

Non determinismo

- introduco comandi imperativi non deterministici
- oltre al non-determinismo indotto dalla concorrenza
- Dijkstra guarded command

```
guardia -> comando
[]
guardia -> comando
[]
...
[]
guardia -> comando
```

Usi

- evito di imporre scelte

```
x <= y -> max = y
[]
y <= x -> max = x
```

- guardia: condizione booleana + test sullo stato di un canale
- utile per gestire la concorrenza
  - evito di dover definire uno scheduler

Esempio: server con due client

- server che deve gestire richieste di lettura e scrittura provenienti da due client
- soluzione deterministica => ordinamento nella gestione dei messaggi => possibile deadlock

```
channel lettura (int dati);
channel scrittura (int dati);

process ClientLettura{
    int risultati;
    ....
    send lettura(risultati);
    ....                    // Uso di risultati
}
```

Esempio

```
process ClientScrittura{
    int valori;
    ....              // Definizione di valori
    send scrittura(valori);
    ....              }

process Server{
   int v;
   int r;
   while true{
      receive lettura(v) -> servi richiesta lettura
      []
      receive scrittura (r) -> servi richiesta scrittura
      } }
```

Implementazione

- scelta pseudo deterministica tra le guardie abilitate,
- problemi di fairness

Composizione parallela

Meccanismo astratto per la definizione di thread - sottoprocessi

```
P   ||   Q   ||   R
```

- indica l'esecuzione parallela di P, Q, R
- visto in precedenza
  - cobegin
  - nell'esempio dei filosofi
- implementazione
  - true concurrency: più processi eseguiti allo stesso istante
  - interleaving: esecuzione alternate delle istruzioni nei vari processi

Esercizi

Grammatiche, scope, Haskell

Grammatiche ambigue

- dimostrare che le seguenti grammatiche sono ambigue esibendo due alberi di derivazione distinti per la stessa stringa:

```
S ::= a | SS

S ::= () | (S) | S S

S ::=   | a  | aR
R ::= b | bS
```

- proporre delle grammatiche non-ambigue equivalenti.
- proporre delle espressioni regolari equivalenti, se esistono.
- proporre degli NFA equivalenti.

Soluzione . . .

Linguaggi e grammatiche

- Data la grammatica

```
S ::= S 0 | R 1
R ::= R 0 | S 1 | 1
```

- si stabilisca se è ambigua,
- nel caso lo sia, si definisca una grammatica non ambigua equivalente
- si definisca il linguaggio generato
- si stabilisca se è un linguaggio regolare,
- nel caso lo si descriva come espressione regolare, e come NFA,
- usano Lex-Alex YACC-Happy, costruire un programma riconoscitore.

Soluzione . . .

Linguaggi e grammatiche

- Ripetere l'esercizio precedente considerando la grammatica

```
S ::= a | S b S
```

Linguaggi e grammatiche

- A partire dalla grammatica degli identificatori `Id` e delle espressioni `Exp`, definire, in maniera non ambigua la grammatica, C, di
  - istruzioni di assegnamento
  - sequenze di istruzioni di assegnamento
  - blocchi
  - cicli for

Linguaggi e grammatiche

- Definire i numeri divisibili per 2 in base 3, come automa, come espressione regolare e come grammatica libera da contesto.
- Riconoscitore Lex o Alex che cataloga le sequenze di numeri ternari in base alla parità.

Linguaggi e grammatiche

- Sia
  - `L1 := {xwy | x w = wR y,  x,y ∈ {a,b}, w ∈ {a,b}* }`
  - `L2 := {w | w = wR , w ∈ {a,b}* }`
  - dove wR è la stringa w rovesciata
  - si diano le stringhe di L1 ed L2 di lunghezza ≤ 4.
  - si descrivano i due linguaggi
  - si diano due grammatiche non ambigue, con simboli iniziali S1 ed S2 per generare L1 ed L2
  - si dica se la grammatica ottenuta unendo le due precedenti e la produzione S $\rightarrow$ S1 | S2 è ambigua

Soluzione . . .

Linguaggi e grammatiche

- Date le produzioni:

$$\texttt{S1} \rightarrow \texttt{a} \mid \texttt{a S1 a} \mid \texttt{b A b}$$
$$\texttt{S2} \rightarrow \texttt{S1 a} \mid \texttt{A}$$
$$\texttt{A} \rightarrow \varepsilon \mid \texttt{a A a} \mid \texttt{b A b}$$
$$\texttt{S} \rightarrow \texttt{S1} \mid \texttt{S2}$$

- e sia LR il linguaggio {v | v = vR , v ∈ {a, b}* }
- si diano le stringhe di L(S1) e L(S2) di lunghezza ≤ 4
- determinare se L(A) e LR coincidono
- descrivere a parole i linguaggi L(S1), L(A)
- si caratterizzino tutte le stringhe di LR \ L(S1)
- determinare quali tra le grammatiche aventi come simboli iniziali S1, S2, A e S sono ambigue
- infine si caratterizzino tutte le stringhe ambigue nella grammatica con simbolo iniziale S

Soluzione . . .

Linguaggi e grammatiche

Analizzare la grammatica (con P simbolo iniziale):

$$\begin{aligned} \texttt{P} &\rightarrow \quad \texttt{a D} \mid \texttt{b P} \\ \texttt{D} &\rightarrow \varepsilon \mid \texttt{a P} \mid \texttt{b D} \end{aligned}$$

Linguaggi e grammatiche

- Si dia la grammatica delle stringe palindrome, costruite a partire dai caratteri `a` e `b`.
  - si consideri il problema di determinare se le il linguaggio generato può essere riconosciuto da un automa a pila
    - non deterministico
    - deterministico

Scope esercizi 2/7/19

Scope dinamico, deep binding, r-value prima di l-value, espressioni argomenti da sinistra a destra, indici vettore da 1:

```
int w[3]={2,4,6}, y=3, z=5;
int foo(valres int[] v, name int z){
  foreach(int x:v)
    {write(v[y--] = x + z)};
  return v[z];
}
write(foo(w, y + z--));
write(w[z]);
```

segue

```
int x = 5;
int y = 7;
void P(ref int x, int z, int R(name int)){
  z = R(x + y);
  write(x, y, z);
}
int Q(name int w){
  return(w + x++);
}
P(y, x, Q);
write (x, y);
```

Scope compito 17/6/19

Scoping statico, assegnamento che calcola prima l'r-value e poi l'l-value, valutazione delle espressioni da sinistra a destra e indici vettori inizianti da 0

```
int i=2, j=1, v[3]={4,3,5};
int foo(val int i, ref int z){
  while ((v[i--] += v[i]) < 15){
    z = v[++i] + (v[j]++);
    write (v[i]);};
  return i;
};
write(foo(j++, v[j--]));
write(v[i],j);
```

segue

```
int x=1;
int y=0;
void P(valueresult int y, ref int z, int R(name int)){
  z = y++ + R(x + y);
  write(x, y, z);
  z = R(z++);
}
{
  int x =3;
  int Q(name int w){
    return(w + x);
  }
  P(x, y, Q);
  write(y, x++);
}
write(y, x);
```

Scope

- Si mostri l'evoluzione delle variabili e dell'output del seguente frammento di programma C-like con: assegnamento che calcola l'r-value prima dell'l-value, indici dei vettori che iniziano da 0, valutazione degli argomenti da sinistra a destra:

```
{
char x[10] = "abcdefghij" ;
int i = 4 ;
char magic(ref char y, ref int j){
    char c = y ;
    x[j] = x[j--] = x[++i];
    write(y) ;
    return c ;
}
write(magic(x[i++], i) );
write(x[i++], i--);
}
```

Scope

Si mostri l'evoluzione delle variabili e l'output del seguente frammento di programma C-like con: assegnamento che calcola r-value dopo l-value, espressioni da destra a sinistra, argomenti chiamate da destra a sinistra e indici vettori inizianti da 1:

```
int v[5]={5,4,3,2,1}, i=3, j=2;
void f(name int k, valres int[] w, val int h)
{for (int i=j++ to j++)
   write(w[i] += v[i] *= ++w[j]-h );
}
f(v[--j]++,v,v[i--]++);
write(v[1]+v[2]);
```

Scope

Si mostri l'evoluzione delle variabili e l'output del seguente frammento di programma C-like con: assegnamento che calcola r-value dopo l-value, espressioni da sinistra a destra e indici vettori inizianti da 1

```
int v[3]={2,4,6}, i=-4, j=5;
int f(valres int k, name int h) {
  int i = 0;
  for (int i = h to h+k-4)
    write(v[i] += (j *= i));
  return k++;
}
write(f(j,i++ + j) + ++j);
write(v[i+3], i++, j);
```

Stack di attivazione

Si mostri l'evoluzione dello stack di attivazione e l'output del seguente programma espresso C-like con:

- scope dinamico e shallow binding,
- assegnamento che calcola l-value dopo r-value,
- valutazione delle espressioni da destra a sinistra,
- argomenti chiamate da destra a sinistra e
- indici dei vettori inizianti da 0 (% è il modulo aritmetico, $-1\%3 = 2$).

segue

```
{
int x=1, y=5 , v[]={ y++, x + y, --x };

int F ( valres int y , ref int z ) {
  int v []= { --z, y ++ + z , x };
  return ( v[0] + v[1] + v[2]);
}
int G(int H (valres int, int ref) , name int z){
  int x = 2;
  write(v[ z %3]);
  return(H(y, x) + x++);
}
{
int x = 4;
write(v[(x ++)%3]++);}
write( G(F, x + y));
}
```

Stack di attivazione

Si mostri l'evoluzione dello stack di attivazione e l'output del seguente programma espresso C-like con:

- scope statico e shallow binding,
- assegnamento che calcola l-value dopo r-value,
- valutazione delle espressioni da destra a sinistra,
- argomenti chiamate da destra a sinistra e
- indici dei vettori inizianti da 0 (% è il modulo aritmetico, $-1\%3 = 2$).

segue

```
int x=4, y=2 ,v[]={x-- , y , ++ y };
int F(int R(valres int), ref int z){
  int x=3;
  write(R(z));
  return(z -= x);}
{
int y=6, v[] = {x-- ,x, --y};
int G(name int x){
  int H(valres int w){
    return(v[(w++)%3]);
  }
  write(v[(y++) %3] );
  return(F(H, x) - x++);
}
write(G(v[x %3]));
}
```

Scope

Si mostri l'evoluzione delle variabili e l'output del seguente frammento di programma C-like con:

```
int x = 3, y = 2;
int foo (name int y){
    int x = 2;
    y += x;
    write (y);
    y++;
}
foo (x);
foo (y);
```

Stack di attivazione

Si mostri l'evoluzione dello stack di attivazione del seguente frammento di programma in un linguaggio C-like con:

- assegnamento che calcola l-value dopo r-value,
- espressioni da destra a sinistra,
- argomenti chiamate da destra a sinistra
- indici vettori inizianti da 0.
- % è il modulo aritmetico, $-1\%3 = 2$;
- nel foreach ad ogni passo di iterazione al parametro w viene assegnato un elemento di a con semantica per riferimento.

Codice

```
int a[3]={1, 2, 3}, y=3, x=6;
int f(ref int x, name int k){
  foreach (int w : a)
    write (a[(--x )%3] += --w + k);
  return ++ x ;
}
write(f(x, y + --x ));
write(++a[y %3], a[(++ x )%3], x);
```

Stack di attivazione

Si mostri l'evoluzione delle variabili e l'output del seguente frammento di programma espresso in un linguaggio C-like con

- scope dinamico, deep binding,
- assegnamento che calcola prima l-value e poi r-value
- valutazione delle espressioni da sinistra a destra:

Si mostri infine l'evoluzione del CRT

codice

```
int x=1 ,y=2;
int F(name v){
  y = x + v;
  write(x, y, v);
  x += v;
  return v++;
}
int Q(ref int x, int R(name int)){
  int y = R(x++);
  y += x;
  write(y);
  return (--x + y);
}
write (Q(y, F));
write (x , y);
```

codice

```
int x=-3, y=1 , z=2;
int F(valres int z, name v){
  z = ++x;
  write(x, y, z);
  y += z;
  write(v);
  return z++;
}
int Q(name int v , ref int x, int R(valres int, name int))
  int w = R(y, v+x);
  y = w+x; write(y);
  return (--x + w);
}
write (Q( x++, y, F));
write (x ,y);
```

Stack attivazione

Si mostri l'evoluzione dello stack di attivazione e l'output del seguente frammento di programma espresso in un linguaggio C-like con

- scope statico e deep binding,
- assegnamento che calcola prima l-value e poi r-value
- valutazione delle espressioni da sinistra a destra e
- indici vettori inizianti da 0:

Si mostri infine l'evoluzione del display.

Codice

```
int x=2, y= -3, v[4]={-7 ,1 ,2, 0};
void G(int w, int R(ref int)){
  int x = w++;
  write(R(x), y++, v);
}
int F(name int w){
  int x = ++w + y;
  int H(ref int w){
    return(w++ - ++v[0]);
  }
  G(H(w), H);
  return y++;
}
write (F(v[x--]), y++, v);
write (x, y++, v);
```

Matrici

- In una matrice `bool A [8][5]`, memorizzata per righe, colonne, con indici che iniziano da 1, in che posizione di memoria, rispetto all'indirizzo base, b, si trova l'elemento `A[2][4]`. Si supponga che ogni intero richieda 4 locazioni di memoria.
- In una matrice `int A [4][8][16]`, memorizzata per piani, righe, colonne, con indici che iniziano da 0, in che posizione di memoria, rispetto all'indirizzo base, b, si trova l'elemento `A[2][4][8]`. Si supponga che ogni intero richieda 4 locazioni di memoria.
- In un matrice quadrata 5x5, quali elementi sono memorizzati sempre nella stessa posizione, sia che la matrice sia memorizzata per righe che per colonne.
  - in una matrice rettangolare 2x4
  - 3 x 5
  - 4 x 7
  - generalizzare

Matrici

- Gli array multidimensionali in Java, sono memorizzati come righe di puntatori (row pointer). Determinare le istruzioni assembly ARM per accedere all'elemento
  - `A[2][4]` di un vettore `int A [4][8]`,
  - `A[2][4][8]` di un vettore `int A [4][8][16]`,

Dimensioni dati

Nelle ipotesi di parole di memoria di 4 byte, memorizzazione a parole allineate, codice caratteri ASCII (UNICODE), quanti byte occupano i seguenti record:

```
struct {
int i;
char a[5];
int j;
char b;
}
```

```
struct {
int i;
char a[5];
char b;
int j;
}
```

Relazioni tra tipi.

L'equivalenza tra tipi in C è strutturale, in generale, ma per nome sul costrutto `struct`.

Determinare, nelle definizioni seguenti, quali variabili sono equivalenti per tipo rispetto a

- equivalenza per nome
- equivalenza per nome lasca
- equivalenza tra tipi i C
- equivalenza strutturale

Definizioni A

```
typedef int integer;
typedef struct{int a; integer b} pair;
typedef struct coppia {integer a; int b} coppia2;
typedef pair[10] array;
typedef struct coppia[10] vettore;
typedef coppia2[10] vettore2
int a;
integer b;
pair c;
struct coppia d;
coppia2 e;
vettore f;
vettore2 g;
coppia[10] i, j;
array k;
```

Definizioni B

```
typedef bool  boolean;
typedef boolean[32] parola
typedef struct{parola a; parola b} coppia
typedef coppia pair

parola a;
boolean[32] b;
bool[32] c;
coppia d
pair e;
struct{parola a; parola b} f
struct{parola b; bool[32] a} g
struct{parola a; bool[32] b} h
```

Conversioni di tipo

Che valore assume la variabile i al termine dell'esecuzione del seguente codice:

```
int i = 2;
float f = i;
union Data {
 int i;
 float f;
} data;
data.f = f;
i = data.i;
```

Tipi polimorfi

Trovare il tipo polimorfo più generale per i seguenti funzionali:

```
(define G (lambda (f x)((f(fx)))))
(define (G f x)(f(f x)))
(define (G f g x)(f(g x)))
(define (G f g x)(f(g (g x))))
(define (H t x y)(if (t x y) x y))
(define (H t x y)(if (t x) x (t y)))
(define (H t x y)(if (t x) y (x y)))
(define (H t x y)(if (t x y) x (t y)))
```

Haskell

Definire le funzioni che risolvano i seguenti problemi. Per ogni funzione definita, comprese quelle ausiliare, descrivere il relativo tipo Haskell.

- Date due liste, costruire la lista di tutte le possibili coppie: elemento della prima lista, elemento della seconda.
- controllare se due liste sono l'una una permutazione dell'altra. L'unica operazione permessa sugli elementi della lista è il test di uguaglianza. Si definisca inoltre il tipo Haskell della funzione definita e delle funzioni ausiliarie

Haskell

- Il grafo di una funzione parziale f : A $\rightarrow$ B è definito come l'insieme di coppie {(a, b) | b = f (a)}, ossia l'insieme di coppie in cui i primi elementi costituiscono il dominio della funzione e i secondi elementi definiscono il comportamento della funzione, sui corrispondenti primi elementi. Scrivere una funzione Haskell che preso il grafo di una funzione f da A in B, rappresentato come lista di coppie, ed una lista l di elementi di tipo A, applica la funzione f a tutti gli elementi della lista. Si tenga presente che la funzione f può essere parziale (non definita su alcuni elementi), nel caso la lista l contenga un elemento su cui f non è definita, l'elemento viene rimosso dalla lista. Scrivere inoltre una funzione Haskell che, dato un lista di coppie, controlli che la lista non contenga due coppie con il primo elemento uguale.

Haskell

- Data una lista, costruire la lista, di liste, contenente tutte le possibile permutazioni della lista originaria.
- Data una matrice, vista come lista di liste, determinare se si tratta di una matrice quadrata.

Si considerino gli alberi generici definiti come:

```
data Tree a =  Tree a [Tree a]
```

- Dato un albero, definire la lista che rappresenta il suo cammino massimo.
- Dato un albero, calcolare il suo diametro.

Sistemi di assegnazione di tipo

Nel sistema di tipi per il linguaggio F1, descritto nei lucidi, costruire la derivazione di tipo per le seguenti espressioni. Nota, nel caso di alberi di derivazione piuttosto complessi, i primi passi di derivazione, a partire dagli assiomi, se elementari, possono essere omessi.

```
\ z : Bool * Bool . if (first z) then True
                                  else (second z)
```

```
\ z : Bool * Bool . if (first z) then (True, (second z))
                                  else ((second z), False)
```

```
\ z : Nat + Bool . case z of
                x : Nat then x + 2 |
                y : Bool then if y then 1
                                   else 0
```

```
\ x : (Ref Nat) . \ y : (Ref Nat) .
   x := (deref x) + (deref y)
```

TAS per linguaggio imperativo

```
{ Nat x = 1;
  Nat y = 2;
  if x < y then x = y else x = x  }
```

```
{ Nat x = 1;
  Nat y = 2;
  while 0 < x {x = x-1;  y = y+1}
```

```
{ Nat x = 1;
  inc ( Nat z){
     x := x + z };
  inc (1);
  inc (x)
}
```