Introduzione al corso

Organizzazione, compilatori

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

Generalizzazione

Una minima parte dei linguaggi esistenti,

- impossibile presentarli tutti,
- tuttavia, i linguaggi di programmazione si basano su una serie di idee, principi comuni:
  - ogni linguaggio ha molte similitudini con diversi altri
  - nell'imparare un nuovo linguaggio si possono sfruttare le conoscenze acquisite su altri linguaggi

Queste idee comuni possono essere:

- imparate tramite esempi: studio un certo numero di linguaggi di programmazione;
- presentate in maniera generale, 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

Mettere in risalto:

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

Rendere più facile l'apprendimento di nuovi linguaggi:

- una volta compresi 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 una 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 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. Concepts in programming languages 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,
qualche aggiornamento durante l'anno.

Organizzazione

- periodo didattico: secondo
- orario, eventualmente modificabile,
  nel caso di sovrapposizioni con altri corsi
- orario di ricevimento, anche via Teams, giovedì: 14:30 – 16:30.

Esame

Tradizionale:

- scritto
  - domande di teoria,
  - esercizi, saranno modificati rispetto agli anni passati
    - una maggiore varietà
- esercizi da svolgere a casa e da discutere durante l'orale
- orale obbligatorio per tutti.

Perché tanti linguaggi di programmazione?

Più paradigmi di programmazione:

Imperativo:

- Procedurale (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, F#)
- Logico, basato su vincoli: descrivo, mediante regole di deduzione, un insieme di predicati, il programma determina per quale valore delle variabili un particolare predicato è vero (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, sicurezza del codice, flessibilità
- diversi usi:
  - calcolo scientifico (Fortran),
  - analisi dei dati (R),
  - sistemi embedded (C, Rust),
  - applicazioni web (JavaScript, PHP),
    - ...

Qualità di un linguaggio:

- Semplicità: (BASIC, Pascal, Scheme)
  - ortogonalità: aspetti diversi restano indipendenti (es. tipi dato e passaggio parametri a procedure), conseguenza: si integrano in modo naturale, poche eccezioni
- Espressività, scrivibilità
  - è facile passare dagli algoritmi al codice,
  - polimorfismo: stesso codice su dati diversi;
- Leggibilità
  - chiarezza, naturalità, semplicità:
  - supporto all'astrazione:
  - è facile modificare del codice.
- Affidabilità
  - facilità di verifica, non ci sono errori nascosti
- Costo
  - efficienza

Cosa rende un linguaggio di successo?

- qualità ma anche
- buon supporto: librerie, codice preesistente, IDE: editor, debugger (tutti i linguaggi più diffusi)
- supporto da uno sponsor importante (C#, Visual Basic, F#, Objective C, Swift, Go)
- ampia diffusione a costi minimi, portabilità (JavaScript, Java, Pascal)
- facile da implementare (BASIC, Forth)

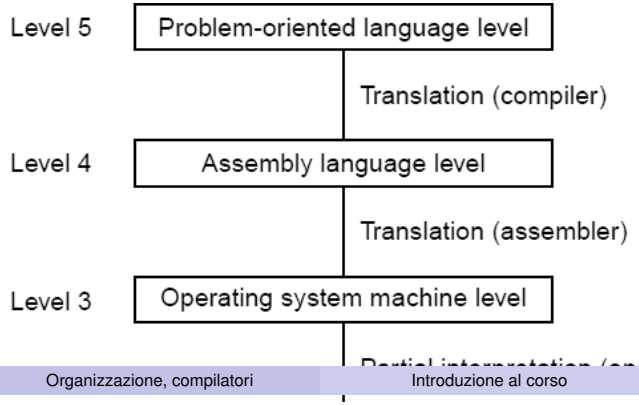
Aspetti di un linguaggio di programmazione

- sintassi: quali sequenze di caratteri costituiscono programmi, la struttura dei programmi
- semantica: 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 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 2 importanti ma più nozionistici, meno concettuali

Macchina astratta

- meccanismo per gestire la complessità del software
- sistema di calcolo diviso in un gerarchia di macchine virtuali (astratte) $\mathcal{M}_i$
- ciascuna costruita sulla precedente, si parte dal livello hardware
- ciascuna caratterizzata dal linguaggio, $\mathcal{L}_i$, che riesce ad eseguire



Esecuzioni del codice

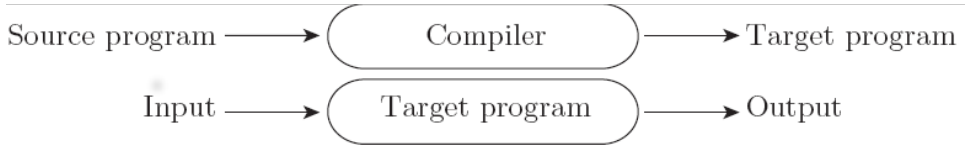
Con quali meccanismi si esegue un programma,
per esempio, scritto in $\mathcal{L}_{Java}$,
quindi relativo alla macchina astratta $\mathcal{M}_{Java}$ (macchina Java)

- 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

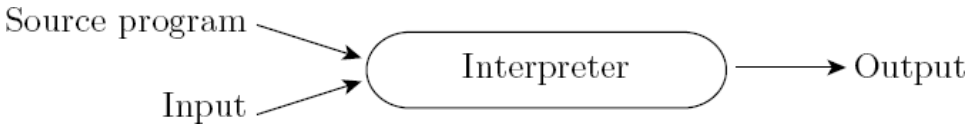
Compilazione pura:

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



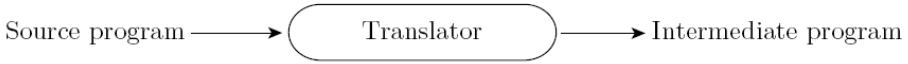
Compilazione vs. Interpretazione

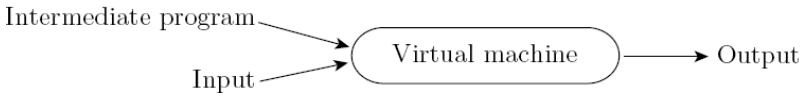
- Compilazione:
  - migliori prestazioni: si evitano traduzioni e controlli a tempo di esecuzione
  - sono possibili controlli prima dell'esecuzione (type-checking statico) errori messi subito in evidenza
- Interpretazione:
  - maggiore flessibilità (Scheme),
  - più semplice da implementare,
  - esecuzione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

Compilazione e 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 +





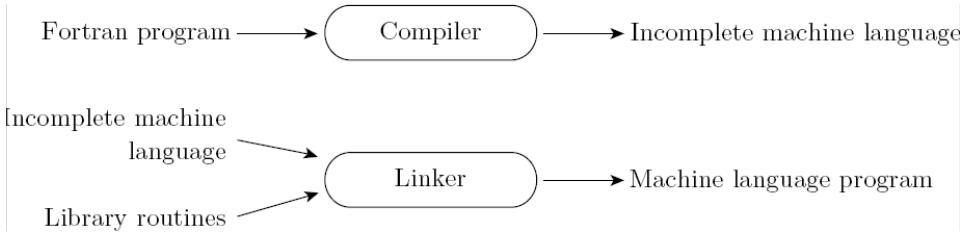
Preprocessing e Interpretazione

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, librerie, subroutine



Assemblaggio post-compilazione

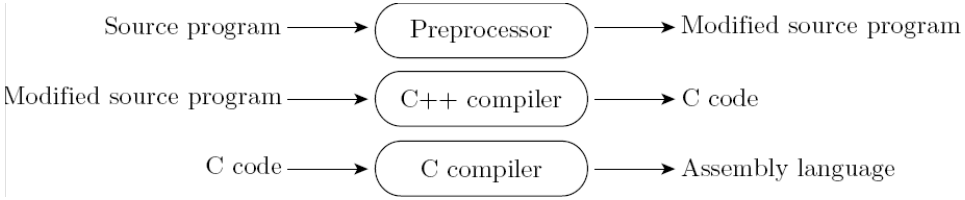
Il compilatore produce assembly (non codice macchina)

- 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 Virtual Machine (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
  - migliora l'efficienza
  - preserva flessibilità e 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.

Bootstrap

meccanismo per costruire compilatori,
letteralmente: 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

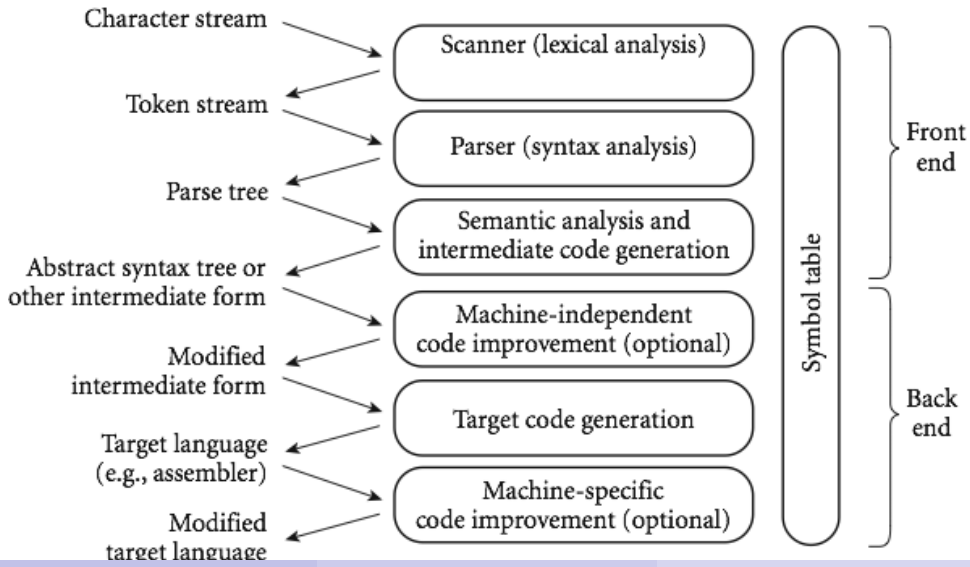
Per ottenere un compilatore Pascal scritto in linguaggio macchina:

- 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 il formalismo aiuti la spiegazione di un meccanismo complesso.

Una panoramica della compilazione

Compilazione divisa in più fasi:



Analisi lessicale - Scansione

- scanner, un compito semplice
- divide il programma in lessemi: le unità più piccole e significative
  - esempi: indentificatori, costanti numeriche, simboli di operazione
- 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
- viene creata una sequenza di token

Analisi sintattica - Parsing

- parser, analizzatore sintattico,
- 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 (principalmente type checking) non implementabili dal parser
  - condizioni non esprimibili con una contex free grammar
    - una funzione viene chiamata con il corretto numero di argomenti
- altri controlli (ad esempio: indice di matrice all'interno del range) 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

Ottimizzazione

trasforma il programma, in codice intermedio,
in uno equivalente (?) ma più efficiente:
esecuzioni più velocemente o con meno memoria

- fase facoltativa
- possibili miglioramenti:
  - rimozione dead code
  - espansione in line di chiamate di funzione
  - fattorizzare sottoespressioni (che appaiono più volte)
  - ottimizzazione dei cicli
    - elimino il ciclo
    - evito il ricalcolo di espressioni costanti

Fase di generazione del codice target

Produce: linguaggio macchina

- con chiamate a funzioni di libreria
- rilocabile,

Può produrre:

- linguaggio assembly,
- codice intermedio (Java bytecode)

Alcune ottimizzazioni specifiche della macchina

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

possono essere eseguite solo durante o dopo la generazione del codice target

Tabella dei simboli

tutte le fasi si basano su una tabella dei simboli (identificatori)

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

  anche dopo che la compilazione è stata completata