Linguaggi e Traduttori: Introduzione al corso

Armando Tacchella

Sistemi e Tecnologie per il Ragionamento Automatico (STAR-Lab)
Dipartimento di Informatica Sistemistica e Telematica (DIST)
Università di Genova

A.A. 2006/2007 - Primo semestre

Outline

Informazioni sul corso

- Docenti, modalità di esame, riferimenti
- Obbiettivi del corso
- Programma del corso

Introduzione ai compilatori

- Terminologia
- Motivazioni
- Architettura

Elementi di un compilatore

- Interfaccia verso il sorgente (Front End)
- Interfaccia verso l'eseguibile (Back End)
- Ottimizzazione
- Ambiente di esecuzione

Outline

Docenti e loro reperibilità

Mauro Di Manzo

Villa Bonino - Piano Ammezzato
Tel. 010-3532818
Riceve su appuntamento
e-mail: `mauro@dist.unige.it`

Armando Tacchella

Villa Bonino - Ultimo Piano
Tel. 010-3532782
Ricevo Lunedì dalle 16.00 alle 17.00
e-mail: `tac@dist.unige.it`
www: `www.star.dist.unige.it/~tac`

Modalità di esame

Progetto didattico (50%)

- Sviluppato in gruppi di 2-3 persone
- Argomento scelto da una lista di proposte (FCFS)
- Supporto dei membri di STAR-Lab
- Elaborato: sorgenti, documentazione e relazione finale

Esame orale (50%)

- Prerequisito: conclusione del progetto didattico
- Discussione del progetto
- Argomenti teorici del corso inerenti al progetto

Riferimenti e materiale didattico

Sito web ufficiale del corso:

www.star.dist.unige.it/~tac/LT-GE

Contiene il seguente materiale:

- Programma dettagliato
- Dispense (inclusa questa introduzione) in formato PDF
- Bibliografia del corso
- Reperibilità dei docenti

Outline

Competenze teoriche

- Principali modelli di computazione
- Formalismi per la descrizione dei linguaggi
- Algoritmi per l'analisi sintattica
- Algoritmi per la traduzione guidata dalla sintassi
- Elementi di semantica dei linguaggi di programmazione

Competenze pratiche

- Struttura di un compilatore
- Strumenti per l'analisi lessicale
- Strumenti per l'analisi sintattica
- Gestione del contesto semantico
- Gestione della rappresentazione intermedia
- Astrazione procedurale
- Generazione di codice

Outline

Struttura del corso

Il corso si divide in due parti:

- Linguaggi formali e traduttori
- Tecnologia dei compilatori

Linguaggi formali e traduttori

- Teoria dei linguaggi e grammatiche
- Linguaggi deterministici
- Trattamento degli errori sintattici
- Traduzioni sintattiche
- Grammatiche ad attributi
- Valutazione degli attributi
- Cenni sulla semantica denotazionale

Tecnologia dei compilatori

- Generazione (automatica) di analizzatori lessicali e sintattici
- Gestione del contesto semantico e traduzione basata sulla sintassi
- Rappresentazioni intermedie: grafiche, lineari, SSA
- Gestione dei simboli (dizionario)
- Astrazione procedurale
- Generazione di codice: espressioni aritmetiche e logiche, vettori, stringhe, strutture, iterazioni, chiam ate a procedura

Outline

Traduttori, interpreti e compilatori

Traduttore Elabora un programma traducendolo in uno di pari significato espresso in un diverso linguaggio.

Interprete Produce gli effetti relativi all'esecuzione di un dato programma.

Compilatore Un traduttore, solitamente da un programma in linguaggio di alto livello (sorgente) in uno in linguaggio più vicino all'hardware (codice intermedio, assembly, codice macchina).

Esempi

- I programmi in C e C++ sono tipicamente compilati
- I programmi PERL e PHP sono tipicamente interpretati
- I programmi in Java vengono compilati (in bytecode) e il risultato è interpretato dalla JVM

Outline

Perché studiare i compilatori?

- I compilatori sono importanti in quanto responsabili per molti aspetti dell'efficienza di un sistema
- I compilatori sono interessanti
  - Includono molte applicazioni pratiche dell'informatica teorica
  - Comportano problematiche a livello algoritmico e ingegneristico
- I compilatori sono molto diffusi
  - Linguaggi all'interno dei programmi applicativi (macro)
  - Formati grafici (per es. Postscript e PDF)
  - Linguaggi special-purpose

Aspetti salienti

- I compilatori svolgono diversi compiti per i quali l'efficienza rappresenta un requisito fondamentale
- Le prestazioni in fase di esecuzione di un programma sono influenzate pesantemente dal compilatore
- Dai compilatori dipende la possibilità di utilizzare efficacemente i costrutti messi a disposizione dai linguaggi
- La crescente complessità delle architetture dei sistemi di elaborazione rende necessari compilatori sempre più sofisticati

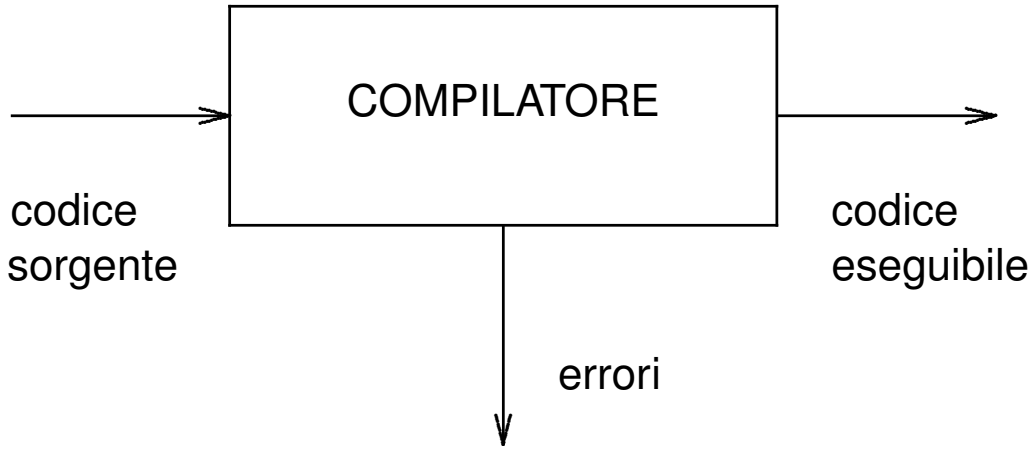
Discipline coinvolte

Lo sviluppo dei compilatori richiede diversi contributi

Intelligenza Artificiale	Algoritmi di ricerca stocastici Tecniche euristiche
Algoritmi	Algoritmi su grafi, Dizionari Programmazione Dinamica
Informatica Teorica	Automi a stati finiti e a pila Grammatiche
Sistemi Operativi	Gestione della memoria Sincronizzazione, Località

Outline

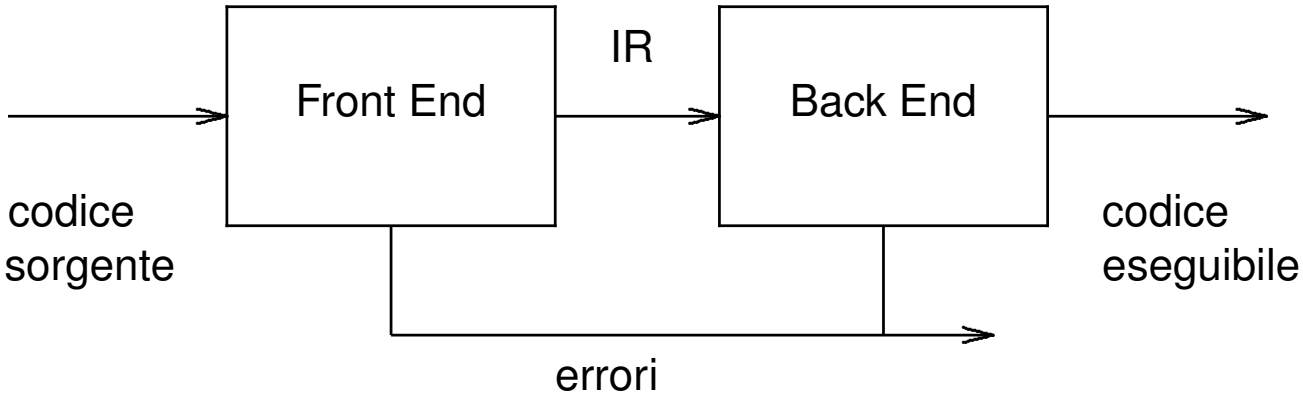
Schema funzionale di un compilatore



Principi fondamentali:

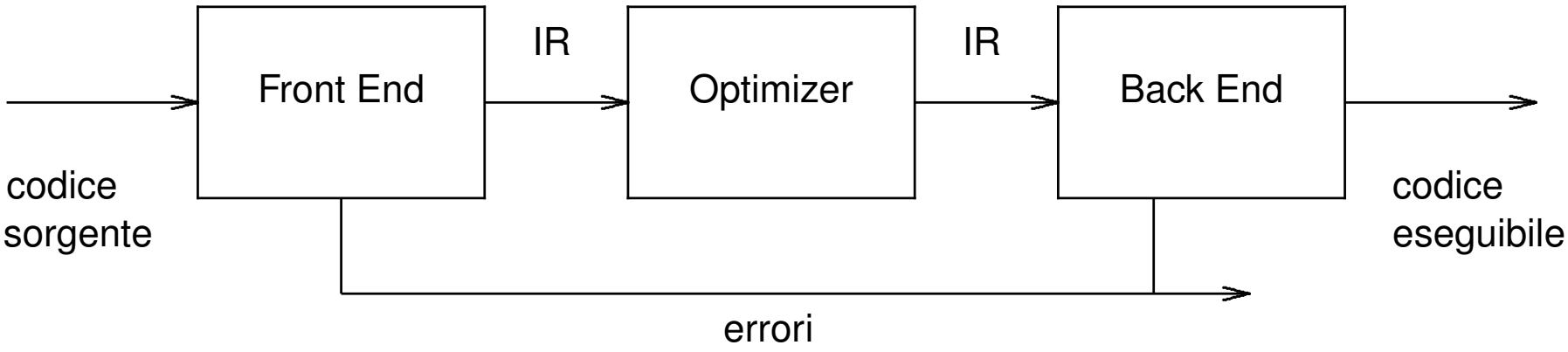
- Il compilatore deve preservare il significato del codice sorgente
- Il compilatore deve migliorare il codice sorgente in modo che gli effetti siano misurabili sul codice eseguibile

Architettura di un compilatore (I)



- Utilizzo di una rappresentazione intermedia (IR)
- È possibile prevedere molteplici Front End
- Il Front End svolge computazioni in $O(n)$, $O(nlogn)$, ...
- Il Back End svolge computazioni potenzialmente in $O(2^n)$

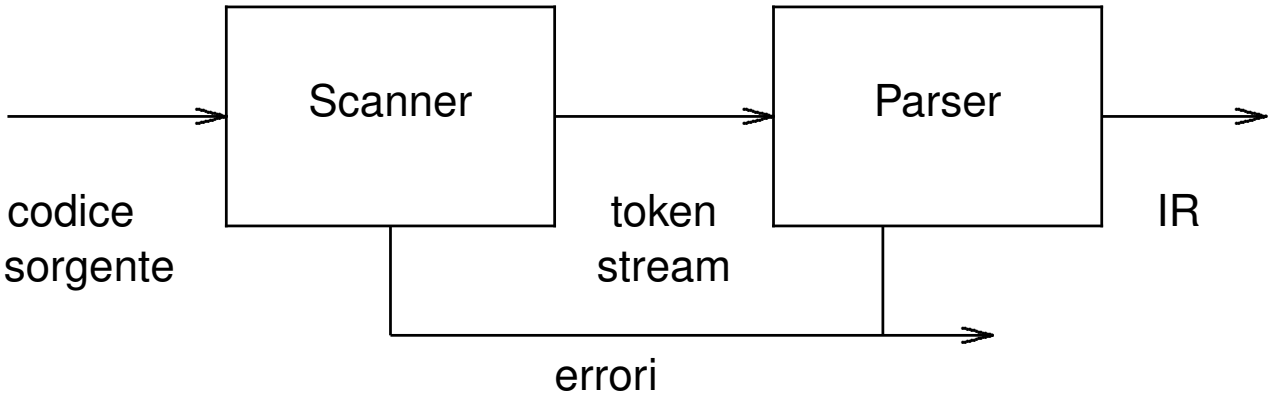
Architettura di un compilatore (II)



- L'Optimizer è un insieme di moduli (passate)
- Migliora il codice IR *in qualche senso*
- Nomenclatura diffusa ma imprecisa: non esiste una definizione di ottimalità sufficientemente generale

Outline

Elementi del Front End



- Riconoscere programmi corretti (errati)
- Emettere messaggi di errore fruibili
- Effettuare analisi lessicali, sintattiche, semantiche
- Produrre codice intermedio e dizionario dei simboli

Elevato grado di automazione

Analizzatore lessicale (Scanner)

- Converte uno stream di caratteri in uno stream di token
- Il token è una coppia (parte del discorso, lessema)
- Esempio:

$$x = x + y \Rightarrow < id, x >< iseq, =>< id, x >< plus, + >< id, y >$$

  - Nel caso di "$=$" e "$+$" è superfluo memorizzare il lessema c'è corrispondenza biunivoca con la parte del discorso
  - Invece un identificatore (*id*) può corrispondere a più lessemi
- Lo scanner "distilla" spazi bianchi, ritorni a capo, ecc.
- La velocità è un fattore cruciale

Analizzatore sintattico (Parser)

- Riconosce la sintassi indipendente dal contesto e le relative violazioni
- Guida l'analisi "semantica" dipendente dal contesto, ad es. controllo dei tipi, numero di parametri, dichiarazione degli identificatori
- Costruisce la rappresentazione intermedia del sorgente

Grammatiche libere dal contesto (Context-free)

- La sintassi indipendente dal contesto è specificata con una grammatica; ad esempio:
  1. goal:=expr
  2. expr:=expr op term | term
  3. term:=NUM | ID
  4. op:=+ | -
- Formalmente, una grammatica $G = (S, N, T, P)$
  - S è il simbolo iniziale
  - N è l'insieme di simboli non terminali
  - T è l'insieme di simboli terminali (o parole)
  - P è un insieme di produzioni o regole di riscrittura
- Nell'esempio S=goal, N={expr,op,term}, T={NUMBER, ID, +, -} e P={1,2,3,4}

Utilizzo di grammatiche context-free (CFG)

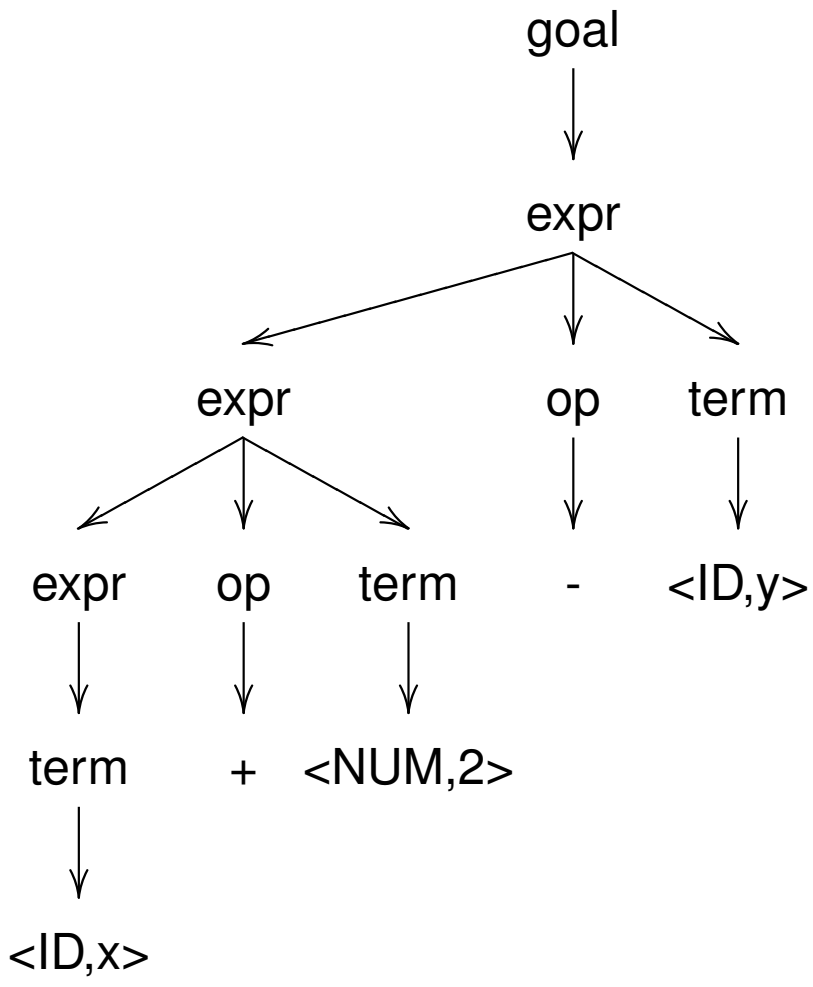
Data una CFG possiamo derivare espressioni utilizzando sostituzioni ripetute; ad esempio:

Produzione	Risultato
	goal
1	expr
2	expr op term
3	expr op *y*
4	expr - *y*
2	expr op term - *y*
3	expr op 2 - *y*
4	expr + 2 - *y*
2	term + 2 - *y*
3	*x* + 2 - *y*

1. goal:=expr
2. expr:=expr op term | term
3. term:=NUM | ID
4. op:=+ | -

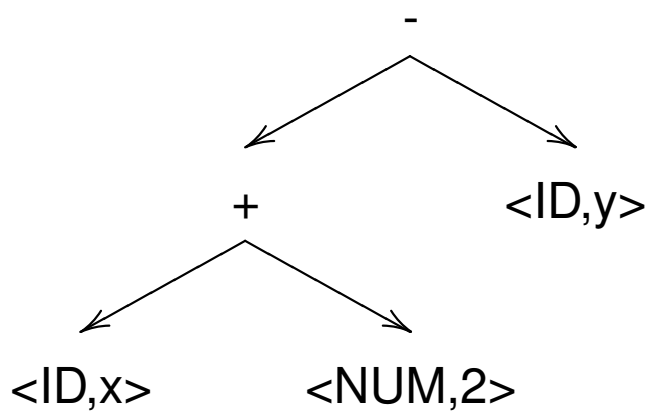
Alberi sintattici

Data un'espressione possiamo interpretarla utilizzando una CFG: tale procedimento può essere rappresentato con un albero sintattico



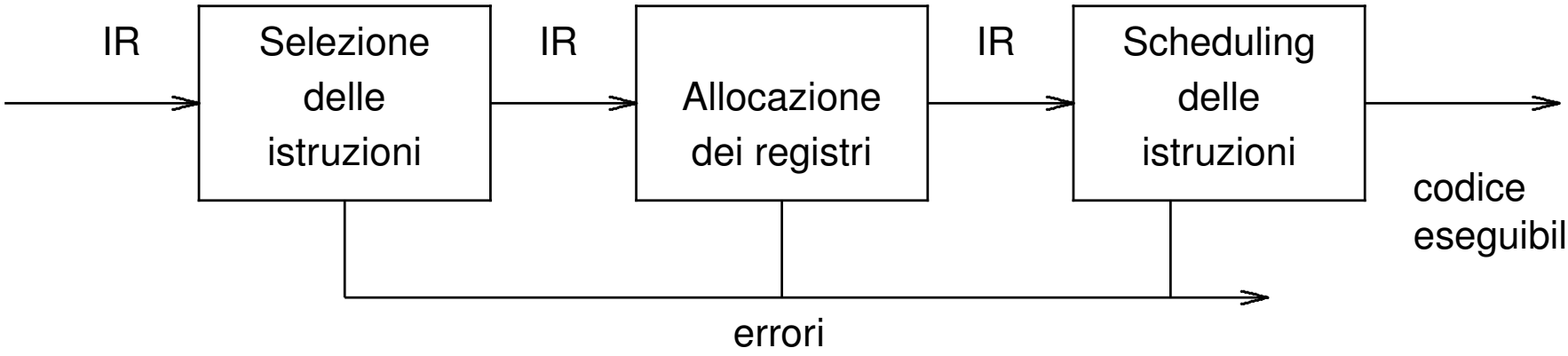
Alberi sintattici astratti (AST)

- Racchiudono le stesse informazioni degli alberi sintattici ma eliminano l'informazione ridondante
- Costituiscono una tipologia di rappresentazione intermedia



Outline

Elementi del Back End



- Tradurre codice IR in codice eseguibile
- Scegliere le istruzioni per realizzare le operazioni
- Decidere i valori da conservare nei registri
- Assicurare il rispetto delle interfacce di sistema

Basso grado di automazione

Selezione delle istruzioni

Obbiettivo: produrre codice veloce e compatto

- Utilizzare proficuamente le caratteristiche dell'HW
- Il "problema del futuro" agli inizi degli anni '80
  - Architetture CISC sempre più complesse
  - L'ortogonalità delle architetture RISC ha semplificato il problema

Allocazione dei registri

Obbiettivo: caricare i valori necessari nei registri prima dell'utilizzo

- Gestione di un numero limitato di risorse
- Può influenzare la scelta delle istruzioni
- Può inserire operazione di LOAD e STORE
- L'allocazione ottima é un problema NP-completo
- I compilatori usano soluzioni approssimate

Riordino delle istruzioni (Scheduling)

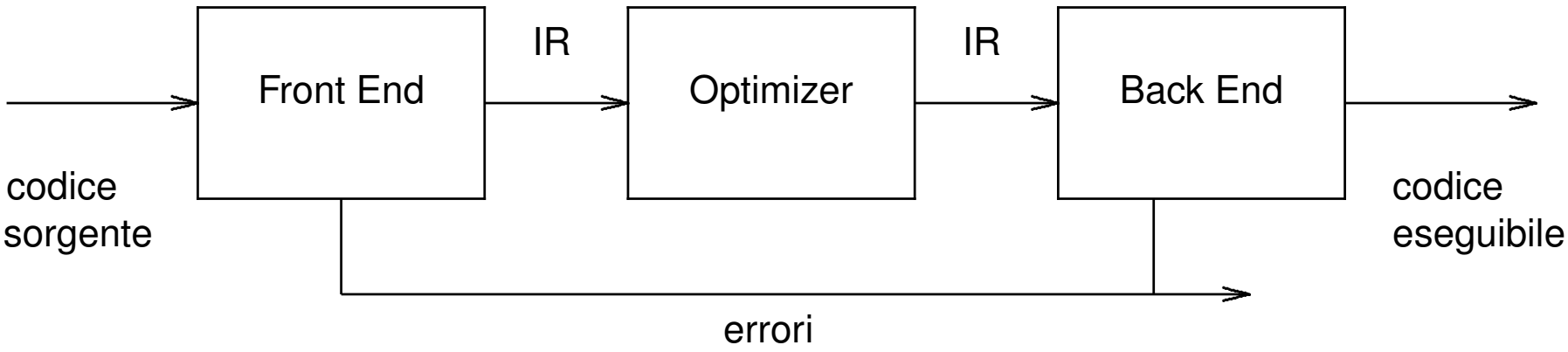
Obbiettivi:

- Minimizzare stalli e bolle nella pipeline
- Utilizzare tutte le unità funzionali efficacemente

Lo scheduling ottimale è un problema NP-completo in quasi tutti i casi di interesse pratico, pertanto i compilatori utilizzano tecniche euristiche

Outline

Schema di compilatore con ottimizzazioni



Obbiettivi:

- Analizzare il codice intermedio e trasformarlo
- Ridurre il tempo di esecuzione (primario)
- Preservare la semantica del codice iniziale

Le dimensioni del codice e il consumo di risorse sono altri obbiettivi che possono essere di prioritari in casi specifici

Esempi di ottimizzazioni possibili

- Scoprire e propagare valori costanti
- Minimizzare il numero di esecuzioni di una computazione dispendiosa
- Specializzare computazioni sulla base del contesto
- Scoprire computazioni ridondanti
- Rimuovere codice inutile o irraggiungibile
- Codificare un idioma in qualche forma particolarmente efficiente

Outline

Ruolo dell'ambiente di esecuzione

- Gestione della memoria
  - allocazione dinamica (heap) e static (stack frame)
  - deallocazione
  - garbage collection
- Controllo dei tipi a run-time
- Gestione degli errori a run-time
- Interfaccia con il sistema operativo
- Eventualmente, gestione di thread, comunicazione, sincronizzazione