Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2011-2012

Linguaggi Formali e Compilatori

Generazione/Ottimizzazione del codice

Giacomo PISCITELLI

Generazione del codice

La fase finale del compilatore è quella della generazione del codice e conseguente sua ottimizzazione. In tale fase viene preso il codice nella sua rappresentazione intermedia (IR), insieme con le informazioni contenute nella *symbol table* e viene prodotto un programma target semanticamente equivalente.

I compilatori più sofisticati effettuano più passi di elaborazione della IR. Ciò è dovuto alla maggiore facilità di applicazione, uno alla volta, degli algoritmi di ottimizzazione o al fatto che l'input ad una ottimizzazione dipende dall'output prodotto da un'altra ottimizzazione.

Inoltre questa struttura del processo di generazione del codice facilita la creazione di un singolo compilatore che può sfruttare più architetture, in quanto solo l'ultimo stadio di generazione del codice necessita dei cambiamenti dipendenti dalla macchina target.

Requisiti del generatore di codice

I requisiti a cui deve ottemperare un generatore di codice sono stringenti e severi: il programma generato deve preservare il significato semantico del programma sorgente ed essere di qualità; deve, cioè, fare un uso efficace delle risorse della macchina target. Il generatore di codice stesso deve essere efficiente.

La sfida nasce dal fatto che, matematicamente, il problema di generare un codice target ottimale per un dato programma sorgente è indecidibile: molti dei sottoproblemi posti dalla generazione del codice, come ad esempio l'allocazione dei registri, sono computazionalmente intrattabili.

In pratica bisogna accontentarsi delle tecniche euristiche per produrre un buon codice, anche se non ottimale. Infatti, fortunatamente, l'euristica è maturata tanto da assicurare che un generatore di codice ben progettato possa produrre codice diverse volte più efficiente e veloce di uno improvvisato.

Compilatori che necessitano di produrre programmi target efficienti prevedono una fase di ottimizzazione già per il codice in IR.

OTTIMIZZAZIONE DEL CODICE: ottimizzazione?

The Word "Optimization"

A clear misnomer!!

Even for simple programs, we cannot prove that a given version is optimal on a specific machine.
Let alone create a compiler to generate that optimal version.

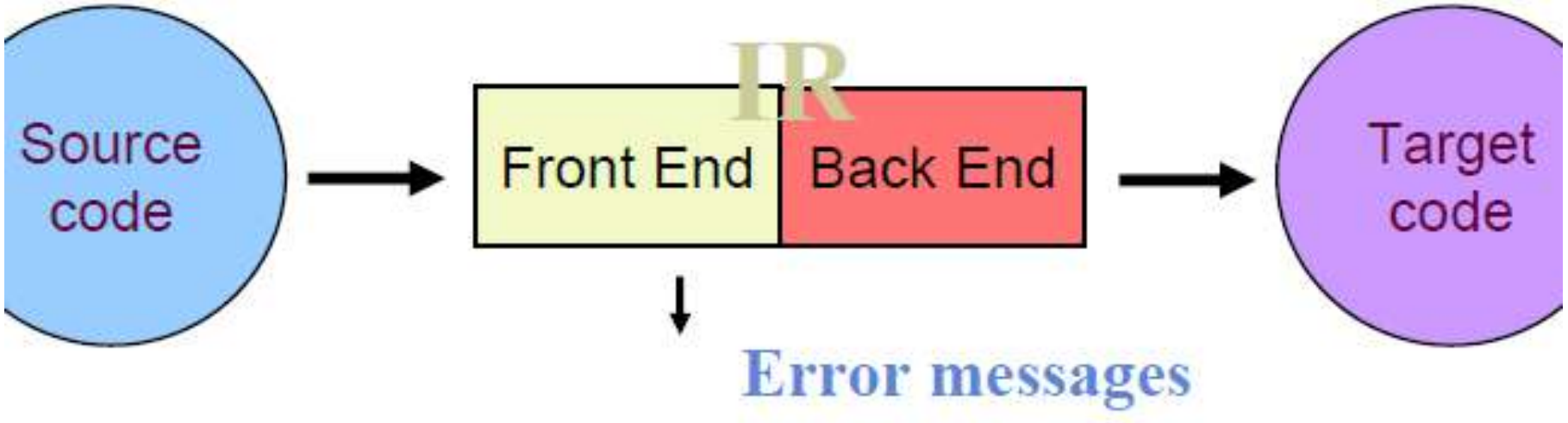
Better:

- produce "improved" code, not "optimal" code
- can sometimes produce worse code
- range of speedup might be from 1.01 to 4 *(or more*)

Ottimizzazione del codice

In aggiunta alla basilare conversione da IR a sequenza lineare d'istruzioni di macchina, un tipico generatore di codice tenta di migliorare il codice generato: usando istruzioni del set più veloci, riducendo il numero delle istruzioni, sfruttando tutti i registri disponibili ed evitando computazioni ridondanti.

Traditional Two-pass Compiler



Typically, front end is O(n) or O(n log n), while back end is NP-C (NP complete)

OTTIMIZZAZIONE DEL CODICE: dove è effettuata?

Very little scope for optimization in front end

- Computation of arithmetic expressions, simplification of logical expressions
- Any work performed here is dependent on parsing process: "syntax-directed"
- Semantic analysis begins to gather information that may help in optimization

Important optimizations take place in back end

- Adapt code to use actual registers provided, or to better exploit functional units

Classical back end optimizations

- change ordering of instructions (latency)
- execute instructions in parallel
- modify data placement (registers, cache)
- reduce power consumption

How can optimizations improve code quality?

Machine independent transformations

1. replace a redundant computation with a reference
2. move evaluation to a less frequently executed place
3. specialize some general purpose code
4. find useless code and remove it
5. expose an opportunity for another optimization

Machine dependent transformations

1. replace a costly operation with a cheaper one
2. replace a sequence of instructions with a more powerful one
3. hide latency
4. improve locality
5. lower power consumption

Machine dependent optimization

"*Optimization for scalar machines is a problem that was solved ten years ago*"
David Kuck - 1990 - Rice University

Machines have changed since 1980

⬇

Changes in architecture implies changes in compilers

- new features present new problems
- changing costs lead to different concerns
- must re-engineer well-known solutions

- Register allocation: goal is to minimize CPU delay waiting for data (typically 60% of total execution time is actually waiting for data)
- Instruction selection
- Instruction scheduling

Particularly important with long-delay instructions and on wide-issue machines (superscalar and Very Long Instruction Word or VLIW, a microprocessors architecture developed for taking advantage from Instruction Level Parallelism)

⬇

Compiler optimizations can significantly improve performance

Scope of Optimization: f.e. register allocation

Local *(or single block)*

track register contents and reuse variables & constants from registers.

- confined to straight line code
- simplest to analyze
- *time frame*: sixties, seventies, maybe now?

Intraprocedural *(or global)*

Within a subprogram, many frequently accessed variables & constants are allocated to few registers.

- consider the whole procedure
- What do we need to optimize an entire procedure?
- classical data-flow analysis, dependence analysis
- *time frame*: seventies thru to now.

Interprocedural *(or whole program)*

Variables & constants accessed by more than one subprogram are allocated to registers. This can greatly reduce call/return overhead.

- analyze whole programs
- What do we need to optimize an entire program?
- analysis and representation still not clear
- *time frame*: late seventies thru to now.

Ottimizzazione del codice: perché?

Why Optimizations

Compiler optimization is essential for modern computer architectures.

- ✓ Without optimization, most applications would perform very poorly on modern architectures
- ✓ Even with optimization, most applications do not get a high fraction of peak performance
- ✓ Optimization techniques are also basis for exploiting SIMD components, vector units, hyperthreading, other forms of multithreading

OTTIMIZZAZIONE DEL CODICE: definizione

Definition

An *optimization* is a transformation that is *expected* to:
1. improve the running time of a program, ***or***
2. decrease its space requirements

Classical optimizations

reduce number of instructions ➔ reduce cost of instructions

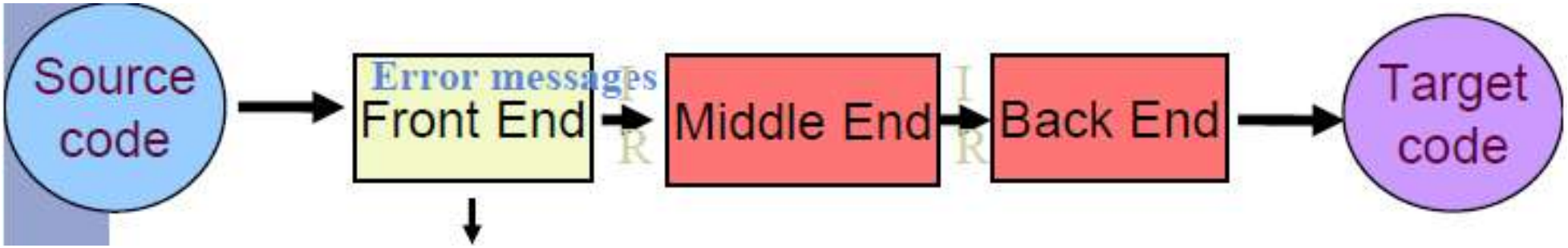
Propagazione di costanti

X:=3; ➔ X:=3;
A:=B+X; ➔ A:=B+3;
evitando un accesso alla memoria

Eliminazione di sottoespressioni comuni

A:=B*C; T:=B*C;
D:=B*C; ⇒ A:=T;
D:=T;

Il middle end



Middle end is dedicated to code improvement

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is assumed to be to reduce running time of the compiled code
- May also improve space, power consumption, …
- Improvements must provably preserve "meaning" of the code

The goal : produce fast code

Some people distinguish middle end from back end, some don't

Generatore di codice: principali task

Un generatore di codice (sia esso in IR o codice di macchina) prevede ***sempre*** tre task principali, :

- **instruction selection**, che consiste nel mapping tra programma in IR e sequenza di istruzioni che possono essere eseguite dal calcolatore target; la complessità di tale mapping è determinata da fattori quali il livello della IR, la natura dell'instruction set del calcolatore target, la qualità desiderata di codice generato.
- **register allocation and assignement**, è un requisito chiave della generazione di codice; un registro è l'unità di memoria ad accesso più veloce di una architettura, ma sfortunatamente non se ne possono avere quanti in realtà servirebbero. Perciò alcuni valori che non possono trovarsi nei registri devono risiedere in memoria e devono essere elaborati con istruzioni diverse, che occupano più spazio e sono decisamente più lente.
- **instruction ordering**, anche esso un problema NP-completo, che implica la scelta dell'ordine in cui le computazioni saranno effettute: alcune computazioni, infatti, comportano l'uso di un minor numero di registri, influenzando così l'efficienza del codice.

Instruction selection

Livello della IR

Se la IR è di alto livello, il generatore di codice può tradurre ogni statement della IR in una sequenza di istruzioni di macchina mediante predefiniti schemi di codifica (code template), che, però, richiedono una successiva fase di ottimizzazione.

Se, invece, la IR riflette alcuni dei dettagli di basso livello del calcolatore target, allora il generatore di codice può far uso di tale informazione per generare sequenze più efficienti di istruzioni di macchina.

Natura dell'instruction set

La natura del set d'istruzioni del calcolatore target ha una notevole influenza sulla difficoltà dell'*instruction selection*; per esempio l'uniformità e completezza dell'instruction set è importante, in quanto la sua mancanza può richiedere laboriose operazioni sostitutive: si pensi al caso in cui le operazioni in virgola mobile facciano uso di registri diversi da quelli adoperati dalle operazioni fra interi.

La qualità del codice

La qualità del codice generato è determinata dalla sua velocità e dimensione. Un programma in una IR può essere realizzato con differenti sequenze d'istruzioni, aventi significative differenze di costi: una traduzione approssimativa della IR può portare ad un codice corretto ma inaccettabilmente inefficiente.

Instruction selection (1 di 2)

Se non ci si cura dell'efficienza del programma target, l'instruction selection può essere diretta: per ciascuna quadrupla, ad esempio, si può progettare uno schema del codice target che deve essere generato per il costrutto.

Per esempio, uno statement

`x = y + z`, equivalente ad una quadrupla della forma ***`op`*** **`y,z,x`**,

dove **`x`**, **`y`** e **`z`** sono staticamente allocate,

può essere tradotto nella sequenza di istruzioni

```
LD    R0,   y          load y into register R0
ADD   R0,   R0   z     add z to R0
ST    x,    R0         store R0 into x
```

Questa strategia spesso determina operazioni "load" e "store" ridondanti.

Instruction selection (2 di 2)

Infatti la sequenza di istruzioni

```
x = y + z
w = x + v
```

sarà tradotta in

```
LD  R0, y
ADD R0, R0  z
ST  x,  R0
LD  R0, x
ADD R0, R0  v
ST  w,  R0
```

Come si può notare, la quarta istruzione è ridondante, perché carica un valore che è stato scaricato dall'istruzione immediatamente precedente. Analogamente è ridondante la terza istruzione se il valore di `x` non è usato successivamente.

Analogamente, se la macchina target disponesse di un'istruzione d'incremento `INC`, l'istruzione a 3 indirizzi `a = a + 1` potrebbe essere realizzata molto più efficientemente dalla singola istruzione `INC a` piuttosto che dalla sequenza

```
LD   R0, a
ADD  R0, R0  #1
ST   a,  R0
```

Strength Reduction: Replace an expensive instruction with an equivalent but cheaper alternative. For example a division may be replaced by multiplication of a reciprocal, or a list append may be replaced by cons operations.

Register allocation and assignement

L'uso dei registri comporta due ordini di problemi:

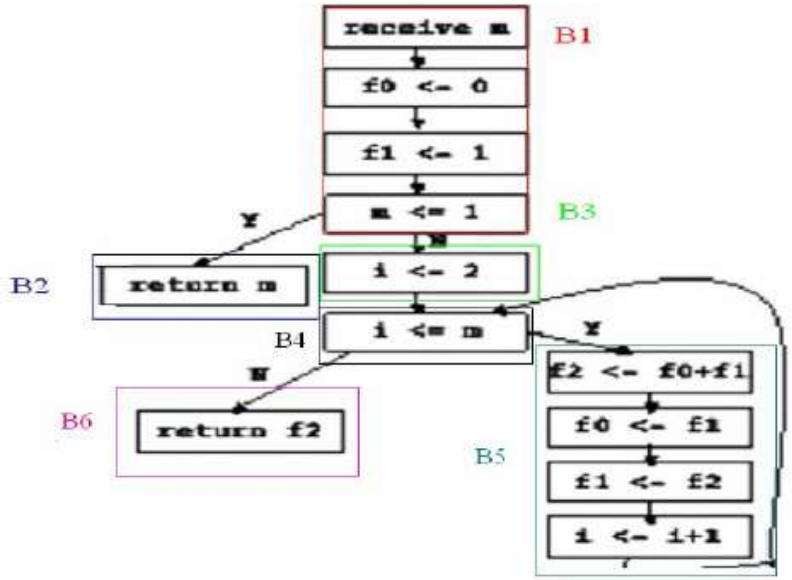
- **register allocation**, con il quale si sceglie l'insieme delle variabili che risiederanno nei registri nelle varie parti del programma;
- **register assignement**, con il quale si decide il particolare registro in cui una variabile risiederà, un problema NP-completo, difficile da risolvere anche nel caso di macchine single-register, a causa della "collisione" con il sistema operativo.

Machine independent optimization: diagrammi di flusso

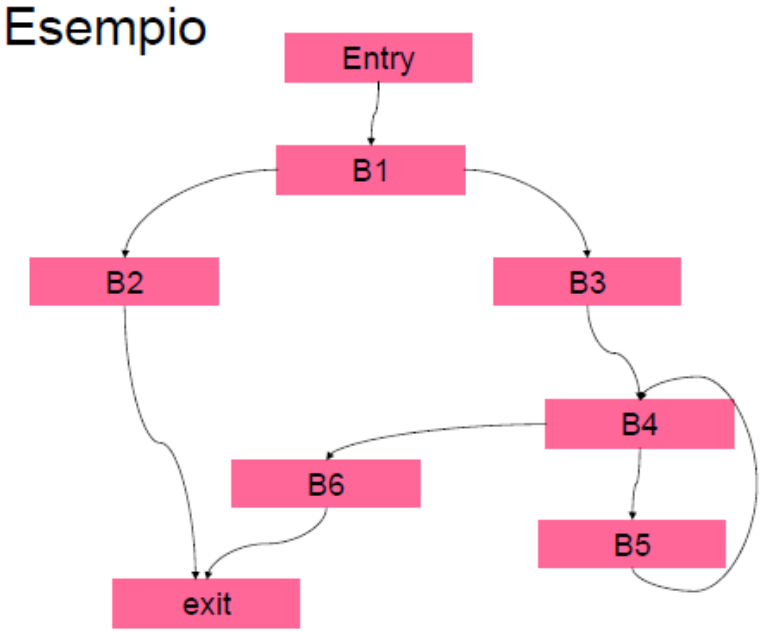
Una particolare IR, basata essenzialmente su grafi aciclici, è molto utile per affrontare diversi aspetti per migliorare la generazione del codice. Tale IR tende a costruire un diagramma di flusso (o *flow graph*) a partire dalle istruzioni a 3 indirizzi (quadruple). Come un albero sintattico di un'espressione, un DAG (*Direct Acyclic Graph*) relativo a un'espressione identifica le sue sottoespressioni comuni, che appaiono, cioè, più di una volta.

Esempio

```
        receive m
        f0 <- 0
        f1 <- 1
        if m <= 1 goto L3
        i <- 2
L1:     if i<= m goto L2
        return f2
L2:     f2 <- f0 + f1
        f0 <- f1
        f1 <- f2
        i <- i+1
        goto L1
L3:     return m
```



Esempio



Dalle istruzioni di un programma al DAG.
(con i simboli B_i sono evidenziati i *basic block*)

Machine independent optimization: ottimizzazione locale

Molti generatori di codice, al fine di procedere all'ottimizzazione locale (ottenendo un significativo miglioramento del tempo di esecuzione di un codice effettuando solamente interventi all'interno di singole parti del programma), partizionano le istruzioni in IR in "**basic block**" (sequenze massime di istruzioni che vengono eseguite sempre assieme, in quanto il flusso di controllo entra solo in corrispondenza della prima istruzione ed esce dal blocco solo dalla sua ultima istruzione, senza operazioni di `halt` o di `jump`, se non, eventualmente, all'ultima istruzione del blocco) che vengono sottoposti a modificazioni locali che tengono conto dell'individuazione, nella IR, di particolari sottoespressioni comuni.
L'individuazione di sottoespressioni comuni consente di realizzare operazioni di copia tra basic block, piuttosto che più laboriose operazioni di traduzione.
Tale ottimizzazione a volte riguarda anche il puro ambito di un basic block.

Una rappresentazione basata su DAG di un blocco base consente di effettuare trasformazioni del codice che ne migliorano la qualità, come:

- ***eliminazione delle sottoespressioni comuni locali***, ovvero istruzioni per calcolare un valore che è stato già ricavato;
- ***codice morto*** (dead code), cioè istruzioni che calcolano un valore mai utilizzato;
- ***riordino di istruzioni con reciproche dipendenze***, per ridurre il tempo d'impegno di un registro;
- ***riordino degli operandi delle istruzioni*** per semplificare il calcolo complessivo.

Machine independent optimization: ottimizzazione globale

L'ottimizzazione globale dei basic block, invece, si occupa di studiare come fluisce l'informazione (*data flow*) tra i vari blocchi di un programma.

Analogamente a quanto avviene nell'ottimizzazione locale, l'ottimizzazione globale provvede a:

- ***individuare sottoespressioni comuni a più blocchi***, così che il risultato ottenuto la prima volta venga memorizzato e usato nelle successive occorrenze;
- ***propagare le operazioni di copia*** (`u = v`), sostituendo al posto di `u` e delle operazioni di assegnazione il valore di `v`;
- ***muovere una computazione al di fuori di un ciclo*** (Loop Invariant Code Motion) se la computazione produce sempre lo stesso valore nel ciclo;
- ***eliminare le "variabili indotte" di un ciclo***, che hanno una sequenza lineare di valori.

OTTIMIZZAZIONE DEL CODICE: compilatori moderni

Complexity of Modern Compilers

A commercial compiler is usually a very large and sophisticated piece of software developed over years or even decades.

A complete industrial compiler may have some millions of lines of code

E.g. Sun's Fortran compiler has ca. 4 Million Lines Of Code (MLOC)

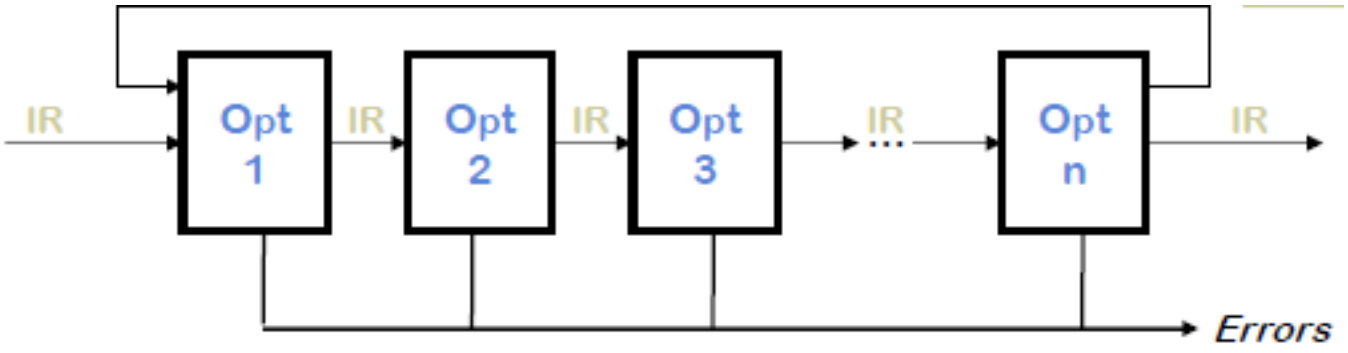
Open64 Fortran/C/C++ has ca. 7 MLOC

No one person understands the complete system.

Analysis versus Optimization

knowledge doesn't make code run faster; changing the code can *sometimes* make it run faster.

We use analysis to transform code.



Modern optimizers are structured as a series of passes

There are a lot of common transformations

Discover and propagate some constant value
Move a computation to a less frequently executed place
Specialize some computation based on context
Discover and remove a redundant computation
Remove useless or unreachable code
Encode an idiom in some particularly efficient form

What does an Optimizing Compiler do?

Performs a good deal of **program analysis**

- Gathers facts about program and its data
- Analyzes individual procedures
- Studies use of data between procedures
- Examines references to arrays
- Compares symbolic expressions

Applies **transformations**

- Provably correct program modifications with a specific purpose
- Rewrite regions of IR if appropriate and legal
- Analysis must prove that application of transformation is legal

We want to improve the code some how.

Improvements should be *objective*, easy to quantify, concrete, *i. e.,* measurements seem easy to take (*Code either gets faster or slower!*)

But,

linear time heuristics for hard problems

unforeseen consequences

multiple ways to achieve the same end

Experimental science takes a lot of work.

Optimization is . . .

Optimization is not a single process or procedure . . .

Rather it is a collection of strategies for program improvement that may be applied at various stages during compilation.

CONCLUDING

Linguaggi Formali e Compilatori ????

They are rather more complex than introductory courses suggest.