La struttura di un compilatore

Enea Zaffanella

enea.zaffanella@unipr.it

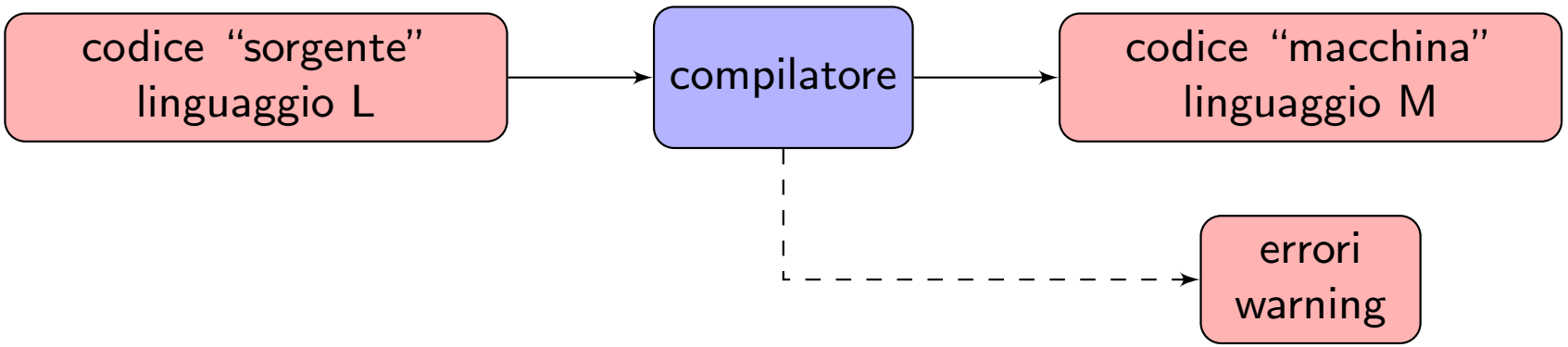
21 settembre 2020

Linguaggi, interpreti e compilatori
Laurea Magistrale in Scienze informatiche

Sommario

Sommario

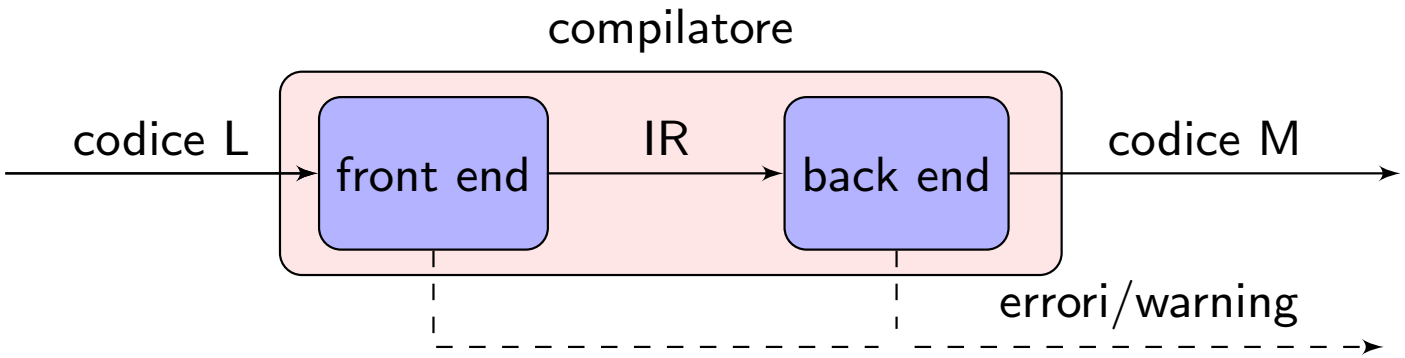
La visuale ad alto livello di astrazione



Requisiti

- Riconoscimento programmi validi/invalidi
- Generazione codice corretto
- Gestione risorse (allocazione e deallocazione memoria)
- Interazione con sistema operativo (e.g., linker dinamico)

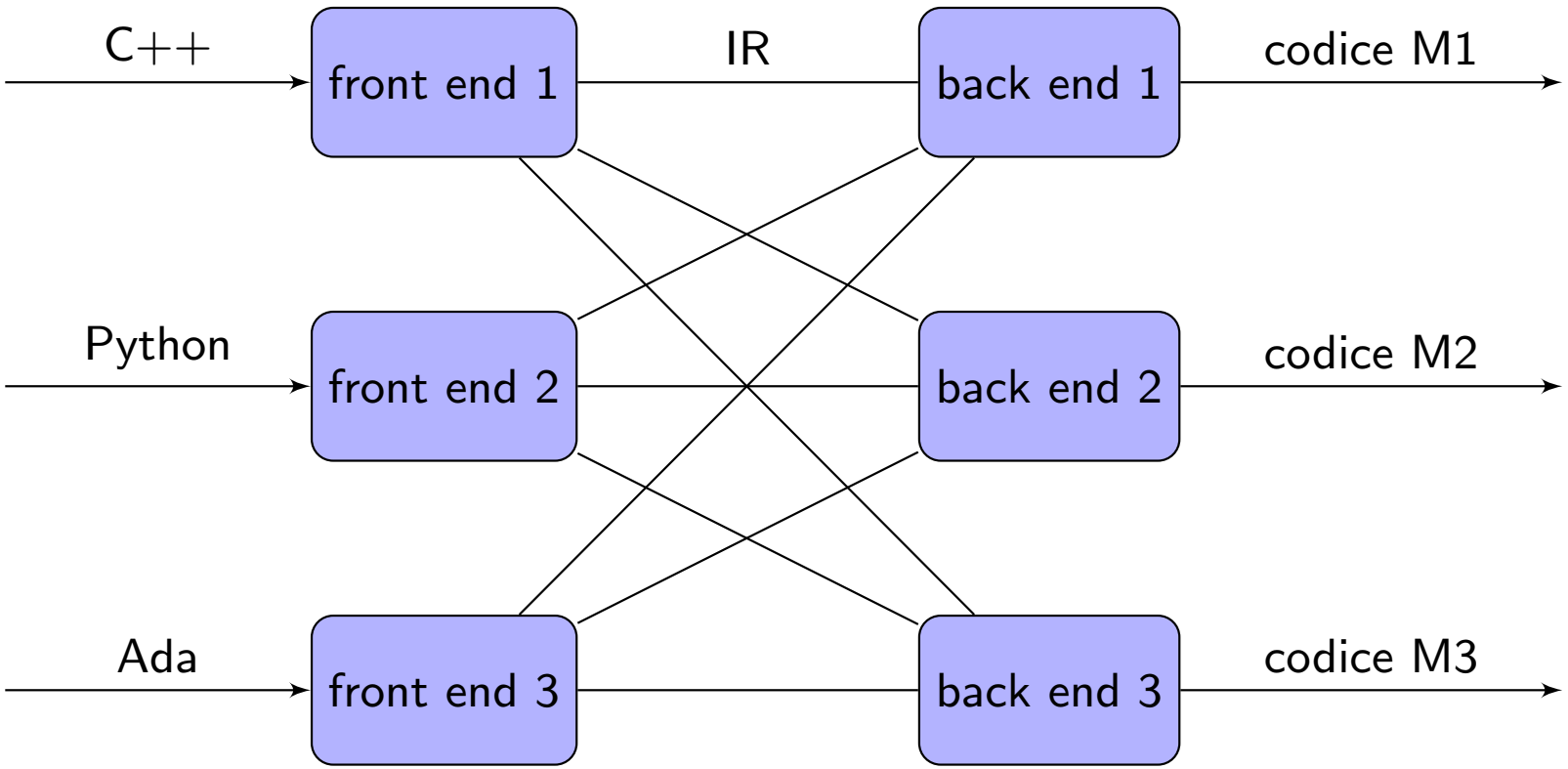
Tradizionale compilatore a due passi



Separazione responsabilità

- Uso di una rappresentazione intermedia (IR)
- Front end: dipendenze da linguaggio sorgente L
- Back end: dipendenze da macchina target (per M)
- Possibile avere più front end e/o più back end

Più front end e/o più back end

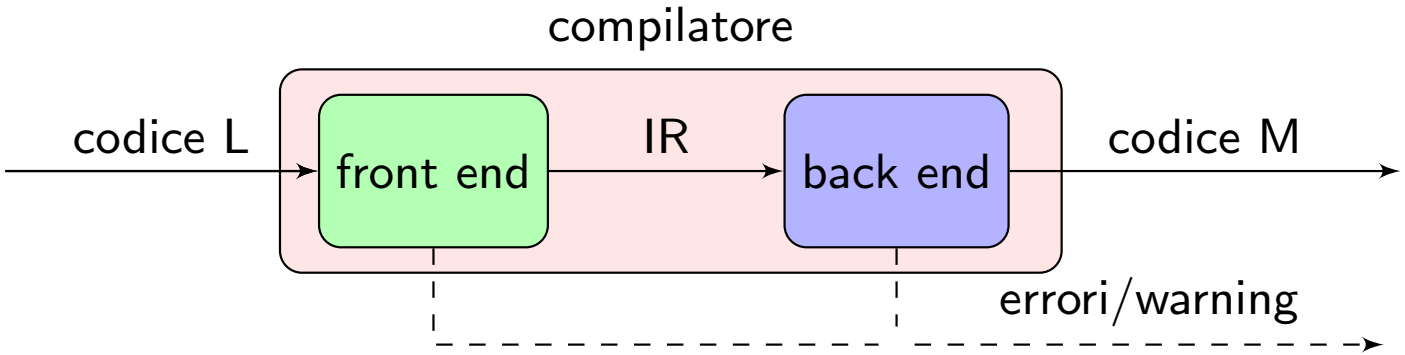


Linguaggio IR

- Deve poter rappresentare le informazioni raccolte dal compilatore
- Varie tipologie (più o meno specializzate)
  - strutturali: alberi, grafi, DAG
  - lineari: 3-address code, stack-machine code
  - ibridi: CFG (control flow graph) per BB (basic block)
- Esempi:
  - GCC $\Longrightarrow$ RTL (Register Transfer Language)
  - LLVM $\Longrightarrow$ LLVM bitcode

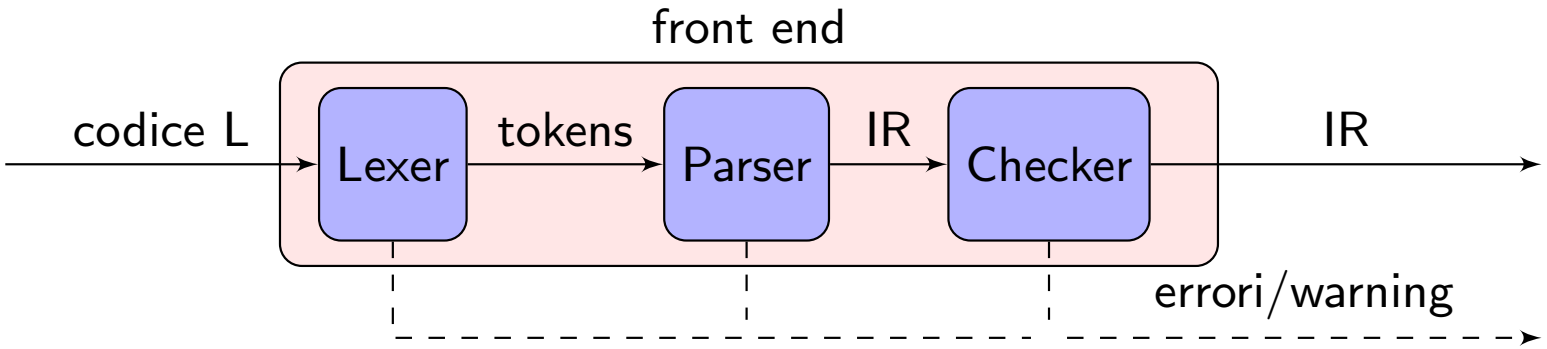
Sommario

Il front end del compilatore



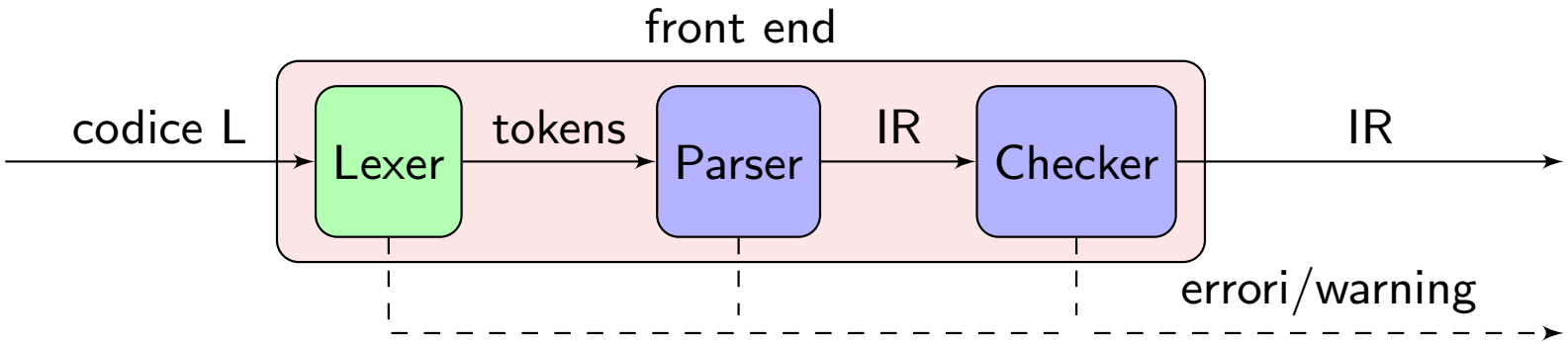
- Riconoscere programmi validi (e invalidi)
- Segnalare errori e warning *facilmente leggibili*
- Produrre codice IR (e strutture dati ausiliarie)

Decomposizione del front end



- Lexer: analisi lessicale
- Parser: analisi sintattica (libera da contesto)
- Checker: analisi di semantica statica (dipendente da contesto)

Componenti front end: il lexer



- Analisi lessicale
- Input: sequenza di caratteri
- Output: sequenza di *token*
- $\mathit{token} = \langle \mathit{part\_of\_speech}, \mathit{lexeme} \rangle$
- Esempi: $\langle \mathrm{KWD}, \texttt{while} \rangle$, $\langle \mathrm{IDENT}, \texttt{somma} \rangle$, $\langle \mathrm{INT}, 42 \rangle$, $\langle \mathrm{FLOAT}, \texttt{3.1415} \rangle$, $\langle \mathrm{STR}, \texttt{"Hello"} \rangle$, ...

Lexer: specifica vs implementazione

Specifica

- Come definire in modo rigoroso quali sono i token validi?
- Linguaggio adeguato: **RE** (espressioni regolari)
- Comprensibile dal *progettista* (essere umano)

Implementazione

- Fattore critico: efficienza
- Lexer detti anche "scanner"
- Riconoscitore: **DFSA** (automa a stati finiti deterministico)
- Spesso generato automaticamente partendo dalla specifica

Esempio: RE per identificatori

Specifica

- DIGIT = [**0**–**9**]
- LETTER = [**a**–**z**A–**Z**] | [**_**]
- ID = LETTER (LETTER | DIGIT)*

Note

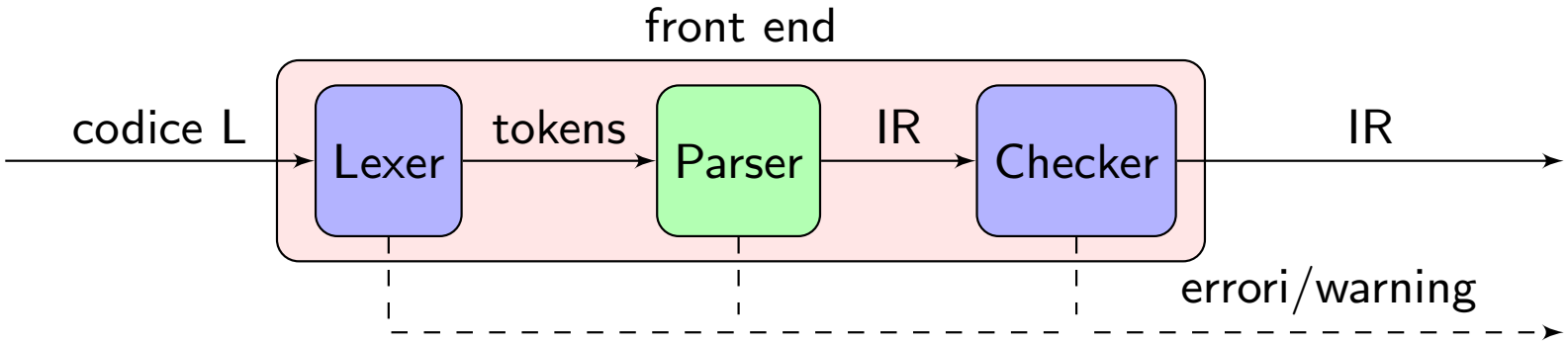
- Caratteri del linguaggio L scritti in **blu**
- Caratteri in nero sono *meta-sintassi*
- [**0**–**9**] abbrevia **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
- La meta-sintassi ammette varie forme di abbreviazione (iterazione positiva, complemento, . . .)

Non solo compilatori

L'uso delle espressioni regolari è pervasivo

- Comandi shell con wildcard (`ls *.pdf`)
- Ricerche di stringhe in file di testo (`grep`) o in database (SQL)
- Query&Replace in editori testuali (`emacs`)
- Uso di wildcard in file di configurazione servizi (firewall, routing, DBMS, ...)
- Librerie per supporto espressioni regolari nei linguaggi di programmazione (`regex` in C++ 2011)

Componenti front end: il parser



- Analisi sintattica
- Input: sequenza di token
- Output: rappresentazione IR della struttura sintattica
- Output deve essere adeguato per fasi successive
  - parse tree (concrete syntax tree), usato raramente
  - AST (abstract syntax tree)

Parser: specifica vs implementazione

Specifica

- Linguaggio adeguato: **CFG** (context free grammar)
- Comprensibile dal *progettista* (essere umano)
- Problemi **non banali** (determinismo, efficienza, ambiguità)

Implementazione

- Riconoscitore: **PDA** (automa a pila non deterministico)
  - codificato direttamente (spesso implicitamente, usando la ricorsione e il backtracking)
  - generato automaticamente partendo dalla grammatica
  - tipologie diverse di generatori (per sottoclassi di grammatiche)
- A volte si applicano *sporchi trucchi*

CFG: context free grammar

$G = \langle S, N, T, P \rangle$

- N: simboli non terminali; T: simboli terminali
- $S \in N$: simbolo iniziale
- $P \subseteq N \times (N \cup T)^*$: produzioni della grammatica

Caratteristiche

- Grammatica libera dal contesto, grammatica acontestuale
- Produzioni: un solo simbolo (di N) a sinistra
- N: categorie sintattiche
- T: categorie lessicali (token prodotti dal lexer)

Esempio di grammatica libera da contesto

Grammatica per espressioni additive

- $S = Expr$
- $N = \{Expr, Op, Term\}$
- $T = \{\mathrm{id}, \mathrm{num}, +, -\}$
- $P = \left\{ \begin{array}{l} S \rightarrow Expr, \\ Expr \rightarrow Term \mid Expr\ Op\ Term, \\ Op \rightarrow + \mid -, \\ Term \rightarrow \mathrm{id} \mid \mathrm{num} \end{array} \right\}$

Abbreviazioni sintattiche

- $(N \rightarrow \gamma) \equiv (N, \gamma) \in P$
- $(N \rightarrow \gamma_1 \mid \gamma_2) \equiv (N \rightarrow \gamma_1, N \rightarrow \gamma_2)$

Esempio di grammatica libera da contesto

Ulteriori abbreviazioni

- N, T e S impliciti
- Numerazione produzioni:

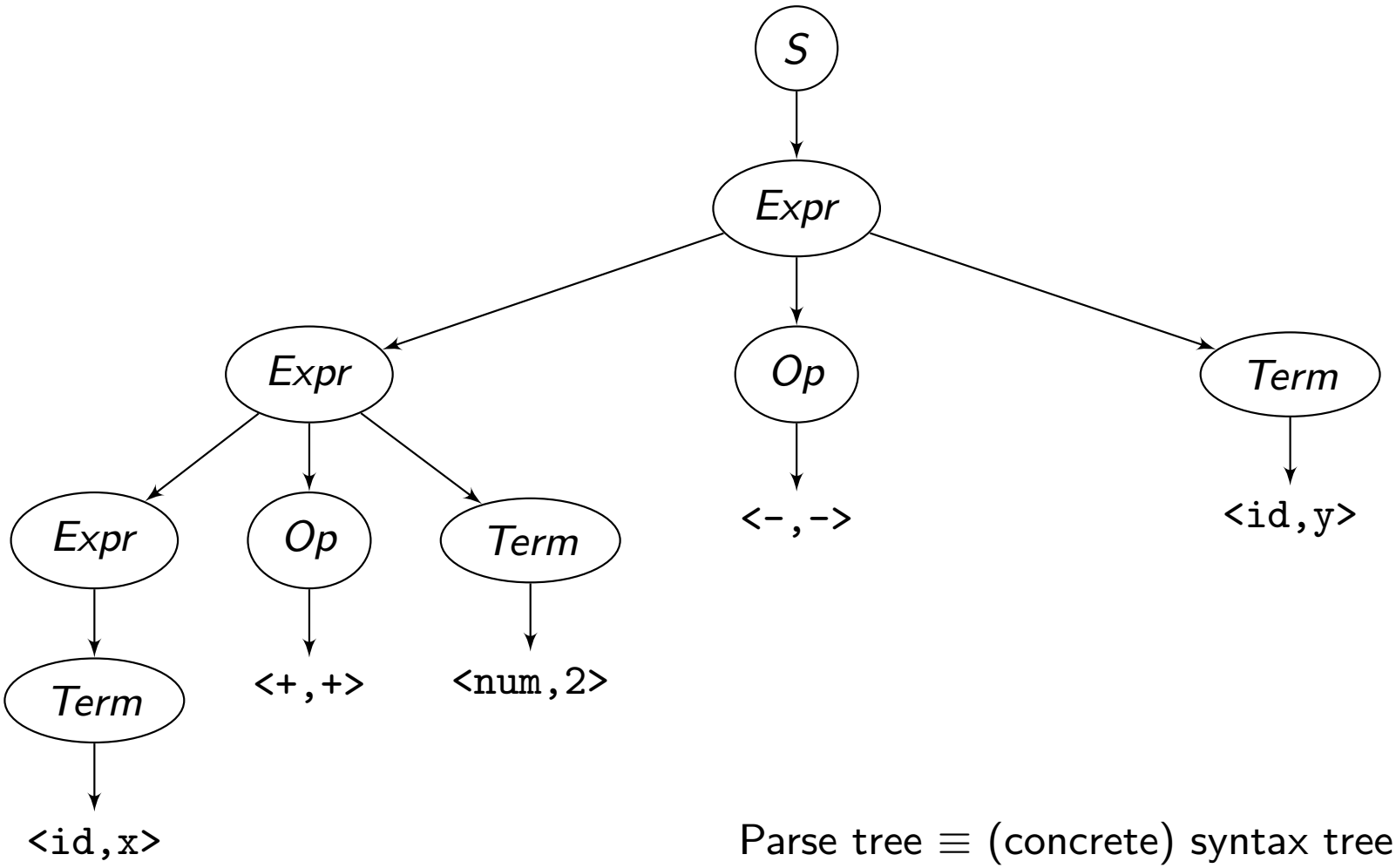
1	S	$\rightarrow$	$Expr$
2	$Expr$	$\rightarrow$	$Term$
3		$\mid$	$Expr\ Op\ Term$
4	Op	$\rightarrow$	+
5		$\mid$	-
6	$Term$	$\rightarrow$	id
7		$\mid$	num

Esempio: derivazione di `x + 2 - y` (top down)

prod	risultato
	S
1	*Expr*
3	*Expr Op Term*
6	*Expr Op* `y`
5	*Expr* `- y`
3	*Expr Op Term* `- y`
7	*Expr Op* `2 - y`
4	*Expr* `+ 2 - y`
2	*Term* `+ 2 - y`
6	`x + 2 - y`

1	*S*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Term*
3		$\mid$	*Expr Op Term*
4	*Op*	$\rightarrow$	+
5		$\mid$	-
6	*Term*	$\rightarrow$	id
7		$\mid$	num

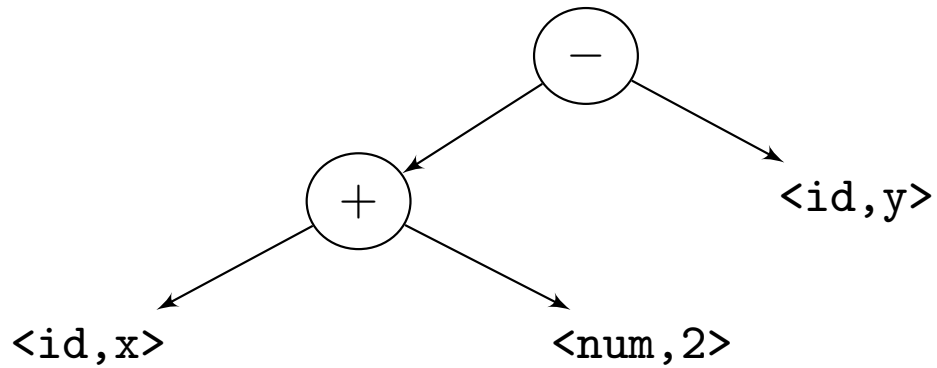
Esempio: parsing di `x + 2 - y` (bottom up)



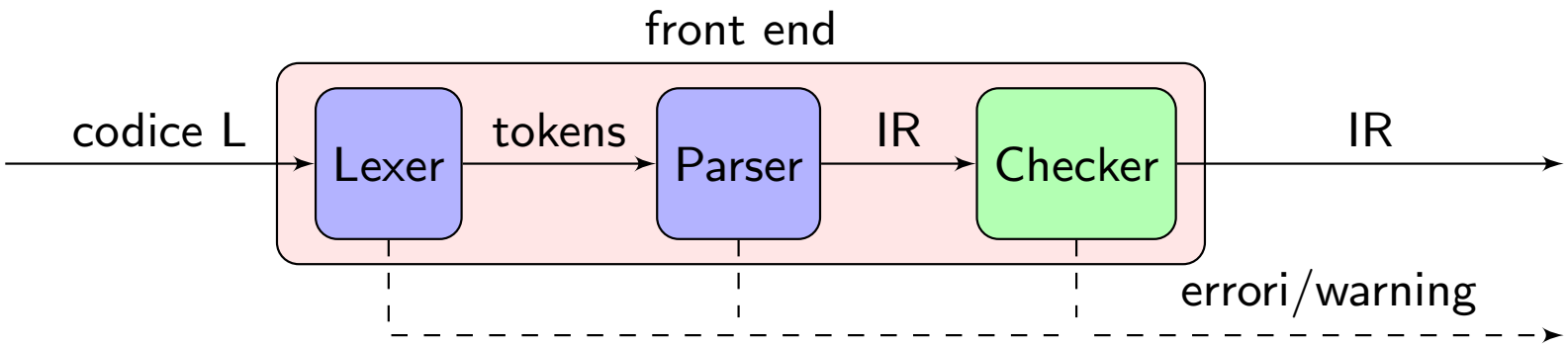
Parse tree $\equiv$ (concrete) syntax tree

Esempio: parse tree $\Longrightarrow$ AST

- Il parse tree contiene troppa informazione ridondante
- Astrazione $\Longrightarrow$ AST (abstract syntax tree)
- AST non rappresenta informazione sul parsing in quanto tale
- AST (+ tabelle supporto) a volte usato come IR anche nelle fasi successive



Componenti front end: il checker



- CSA (context-sensitive analysis): analisi di semantica statica
- Input: un AST "grezzo"
- Output: un AST arricchito di informazione dipendente dal contesto (tipi di dato, conversioni implicite, risoluzione overloading, ...)
- Nota Bene: spesso Checker e Parser sono fortemente integrati; l'AST "grezzo" non viene generato, si costruisce direttamente l'AST arricchito a partire dal parse tree.

Checker: specifica vs implementazione

Specifica

- Come definire in modo rigoroso quali sono i programmi validi?
- Linguaggio naturale (e.g., lo **standard** del linguaggio, manualistica, documentazione compilatore)
- Semantiche formali (sistemi di regole)
- Problema: spesso **non** facilmente comprensibili

Implementazione

- Fattore critico: correttezza
- In passato: grammatiche arricchite da *attributi* calcolati
- Più frequentemente: SDT (syntax directed translation), codifica di specifici algoritmi di visita

Esempi di specifica

Linguaggio naturale (?)

7.6 Integral promotions **[conv.prom]**

1 A prvalue of an integer type other than `bool`, `char16_t`, `char32_t`, or `wchar_t` whose integer conversion rank (6.7.4) is less than the rank of `int` can be converted to a prvalue of type `int` if `int` can represent all the values of the source type; otherwise, the source prvalue can be converted to a prvalue of type `unsigned int`.

Una regola semantica per il calcolo dei tipi

$$\frac{\beta \models \mathit{expr}_1 : \mathrm{int} \qquad \beta \models \mathit{expr}_2 : \mathrm{int}}{\beta \models \mathit{expr}_1 == \mathit{expr}_2 : \mathrm{bool}}$$

Esempio: AST di un programma C

```
$ cat a.c
int main(int argc, char* argv[]) {
  return argc==1;
}
```

Esempio: dump dell'AST prodotto da clang

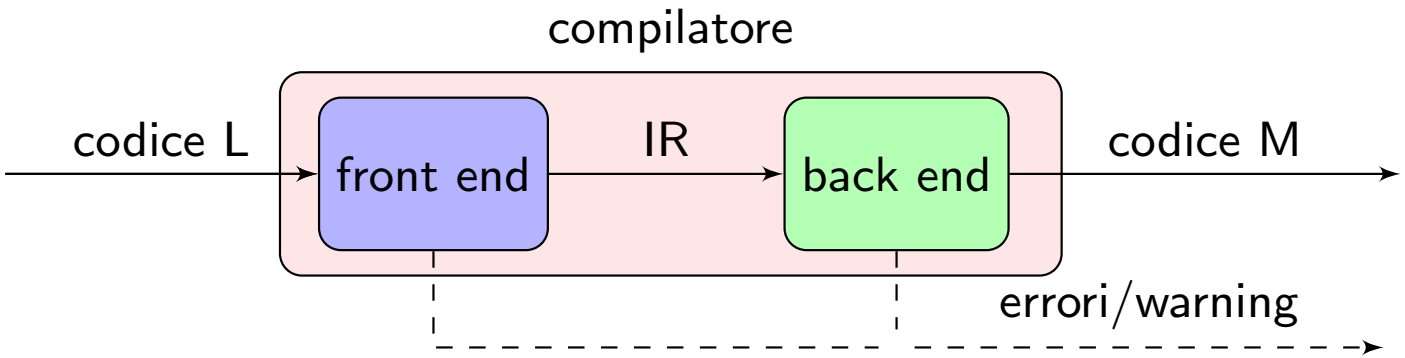
```
$ clang -Xclang -ast-dump -fsyntax-only a.c
TranslationUnitDecl 0x829008 <<invalid sloc>> <invalid sloc>
[ ... omissis ...]
`-FunctionDecl 0x887de0 <a.c:1:1, line:3:1> line:1:5 main 'int (int, char **)'
  |-ParmVarDecl 0x887c50 <col:10, col:14> col:14 used argc 'int'
  |-ParmVarDecl 0x887d00 <col:20, col:27> col:27 argv 'char **'
  `-CompoundStmt 0x887f60 <col:33, line:3:1>
    `-ReturnStmt 0x887f50 <line:2:3, col:18>
      `-BinaryOperator 0x887f30 <col:10, col:18> 'int' '=='
        |-ImplicitCastExpr 0x887f18 <col:10> 'int' <LValueToRValue>
        | `-DeclRefExpr 0x887ed8 <col:10> 'int' lvalue ParmVar 0x887c50 'argc'
        `-IntegerLiteral 0x887ef8 <col:18> 'int' 1
```

Stesso programma, ma compilato come C++

```
$ clang -xc++ -Xclang -ast-dump -fsyntax-only a.c
TranslationUnitDecl 0xd86e78 <<invalid sloc>> <invalid sloc>
[ ... omissis ...]
`-FunctionDecl 0xdc2c90 <a.c:1:1, line:3:1> line:1:5 main 'int (int, char **)'
  |-ParmVarDecl 0xdc2b08 <col:10, col:14> col:14 used argc 'int'
  |-ParmVarDecl 0xdc2bb0 <col:20, col:27> col:27 argv 'char **'
  `-CompoundStmt 0xdc2e28 <col:33, line:3:1>
    `-ReturnStmt 0xdc2e18 <line:2:3, col:18>
      `-ImplicitCastExpr 0xdc2e00 <col:10, col:18> 'int' <IntegralCast>
        `-BinaryOperator 0xdc2de0 <col:10, col:18> 'bool' '=='
          |-ImplicitCastExpr 0xdc2dc8 <col:10> 'int' <LValueToRValue>
          | `-DeclRefExpr 0xdc2d88 <col:10> 'int' lvalue ParmVar 0xdc2b08 'argc
          `-IntegerLiteral 0xdc2da8 <col:18> 'int' 1
```

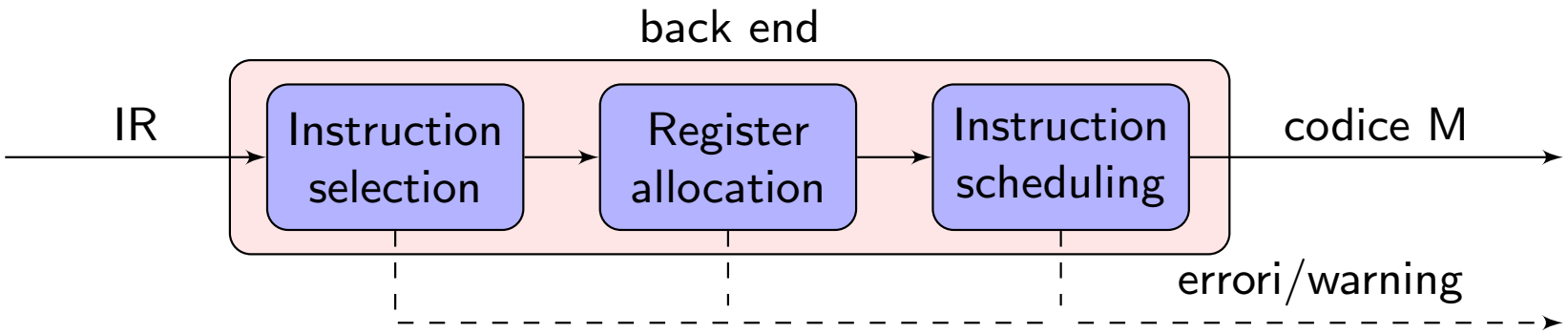
Sommario

Il back end del compilatore



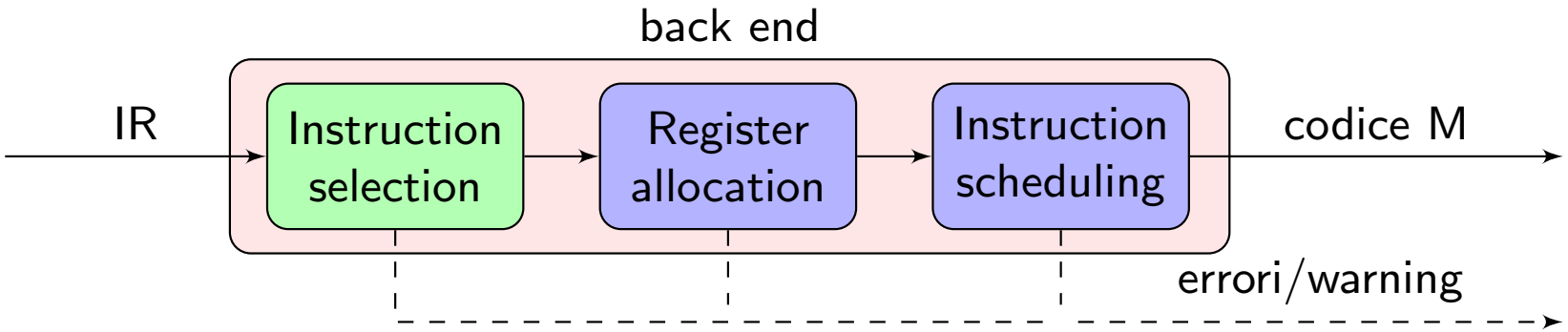
- Traduzione da IR a linguaggio M (codice “macchina”)
- Scelta delle istruzioni per implementare le operazioni
- Decidere quali valori mantenere nei registri
- Rispettare interfacce di sistema

Decomposizione del back end



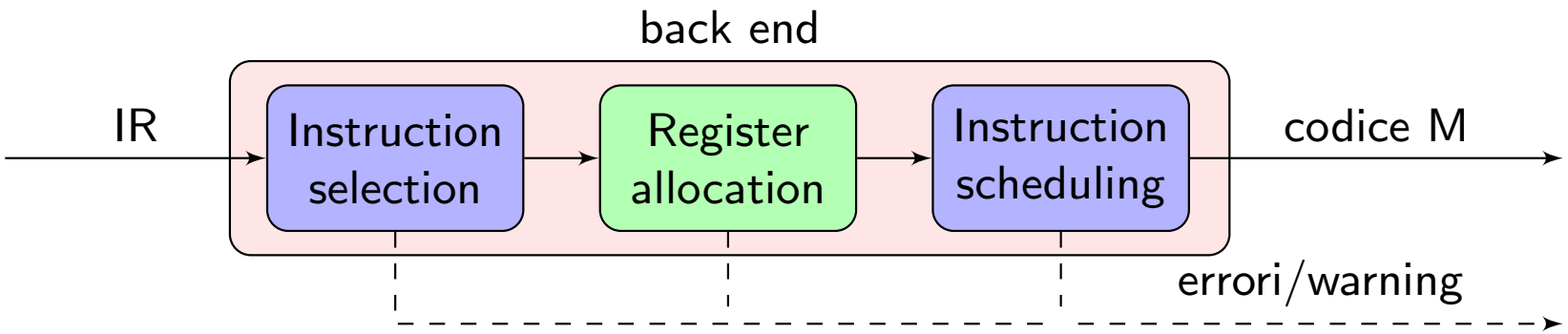
- Selezione istruzioni
- Allocazione registri
- Scheduling istruzioni
- Problemi inerentemente complicati (NPC)
- Applicazione di tecniche euristiche

Instruction selection



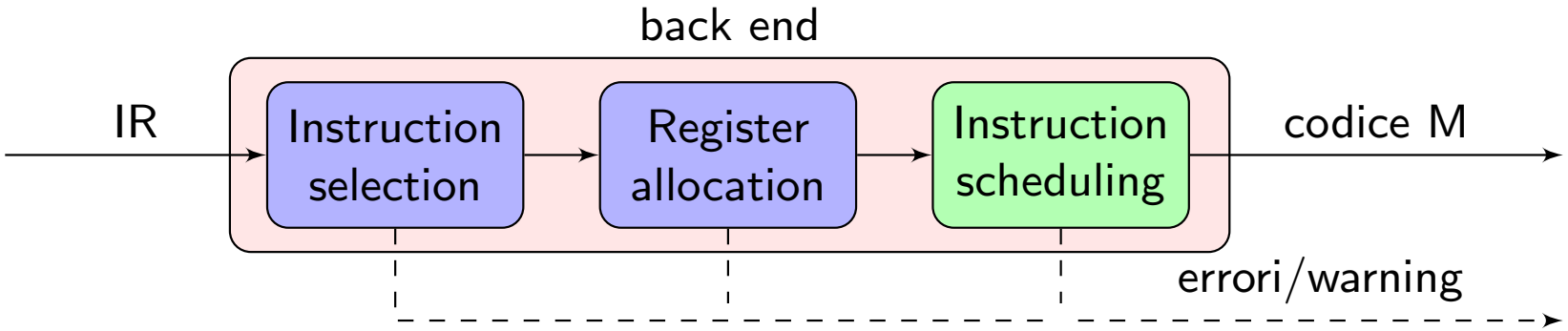
- Produzione di codice veloce (tempo) e compatto (spazio)
- Sfruttare caratteristiche della "macchina" per M
- Spesso visto come problema di *pattern matching*
- Ricerca di ottimi locali (approssimazione)

Register allocation



- Gestione di un insieme finito di risorse (registri CPU)
- **Splilling** dei registri: istruzioni di LOAD e STORE
- Colorazione di grafi

Instruction scheduling



- Gestione **dipendenze** a livello hardware: evitare le attese
- Ottimizzare l'uso delle unità funzionali della CPU
- Può modificare il tempo di vita dei valori nei registri (quindi influenzare la loro allocazione)

Un esempio

Codice IR

$a \leftarrow b \times c + d$
$e \leftarrow f + a$

Pseudo-codice macchina

unit 1
load @b ⇒ r1
load @c ⇒ r2
mult r1, r2 ⇒ r3
load @d ⇒ r4
add r3, r4 ⇒ r5
store r5 ⇒ @a
load @f ⇒ r6
add r5, r6 ⇒ r7
store r7 ⇒ @e

Assunzioni su latenza istruzioni
load, store: 2 cicli
altre istruzioni: 1 ciclo

Un esempio (cont.)

Codice IR

$a \leftarrow b \times c + d$
$e \leftarrow f + a$

Pseudo-codice macchina

unit 1
load @b ⇒ r1
load @c ⇒ r2
mult r1, r2 ⇒ r3
load @d ⇒ r4
add r3, r4 ⇒ r5
store r5 ⇒ @a
load @f ⇒ r6
add r5, r6 ⇒ r7
store r7 ⇒ @e

unit 1	unit 2
load @b ⇒ r1	load @c ⇒ r2
load @d ⇒ r4	load @f ⇒ r6
mult r1, r2 ⇒ r3	nop
add r3, r4 ⇒ r5	nop
store r5 ⇒ @a	nop
add r5, r6 ⇒ r7	nop
store r7 ⇒ @e	nop

Un esempio (cont.)

Codice IR

$a \leftarrow b \times c + d$

$e \leftarrow f + a$

Pseudo-codice macchina

unit 1	unit 2
load @b ⇒ r1	load @c ⇒ r2
load @d ⇒ r4	load @f ⇒ r6
mult r1, r2 ⇒ r3	nop
add r3, r4 ⇒ r5	nop
store r5 ⇒ @a	nop
add r5, r6 ⇒ r7	nop
store r7 ⇒ @e	nop

Un esempio (cont.)

Codice IR

$a \leftarrow b \times c + d$

$e \leftarrow f + a$

Pseudo-codice macchina

unit 1	unit 2
load @b ⇒ r1	load @c ⇒ r2
load @d ⇒ r4	load @f ⇒ r6
mult r1, r2 ⇒ r3	nop
add r3, r4 ⇒ r5	nop
store r5 ⇒ @a	nop
add r5, r6 ⇒ r7	nop
store r7 ⇒ @e	nop

unit 1	unit 2
load @b ⇒ r1	load @c ⇒ r2
load @d ⇒ r4	nop
mult r1, r2 ⇒ r3	nop
add r3, r4 ⇒ r5	load @f ⇒ r6
store r5 ⇒ @a	nop
add r5, r6 ⇒ r7	nop
store r7 ⇒ @e	nop

Un esempio (cont.)

Codice IR

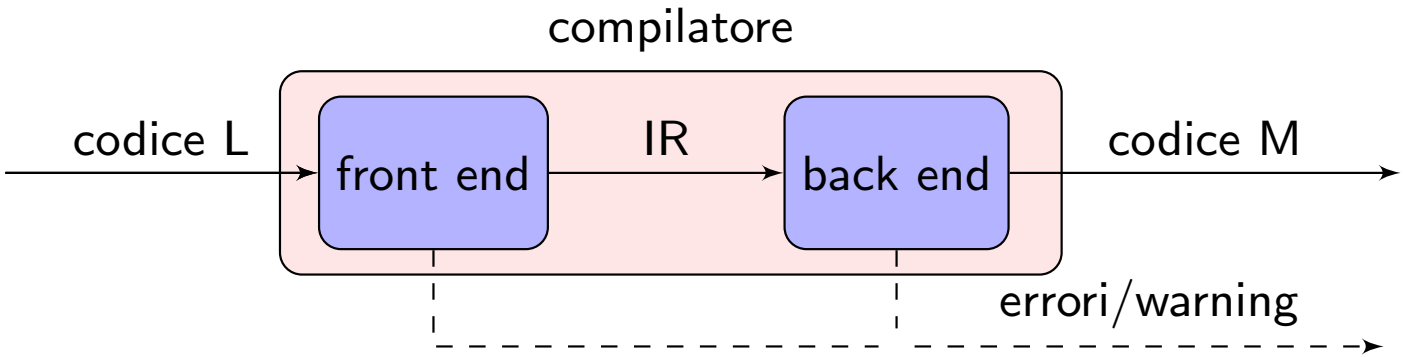
$a \leftarrow b \times c + d$
$e \leftarrow f + a$

Pseudo-codice macchina

unit 1	unit 2
load @b ⇒ r1	load @c ⇒ r2
load @d ⇒ r3	load @f ⇒ r4
mult r1, r2 ⇒ r1	nop
add r1, r3 ⇒ r1	nop
store r1 ⇒ @a	nop
add r1, r4 ⇒ r2	nop
store r2 ⇒ @e	nop

unit 1	unit 2
load @b ⇒ r1	load @c ⇒ r2
load @d ⇒ r3	nop
mult r1, r2 ⇒ r1	nop
add r1, r3 ⇒ r1	load @f ⇒ r2
store r1 ⇒ @a	nop
add r1, r2 ⇒ r2	nop
store r2 ⇒ @e	nop

Front end e back end

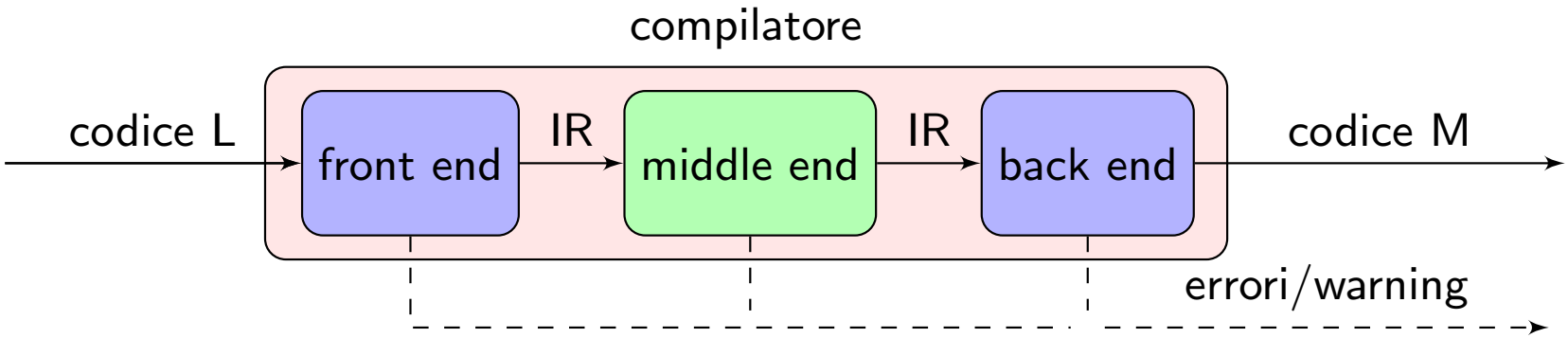


Separazione responsabilità e competenze

- Front end: linguaggi sorgente, generatori automatici, problema **"risolto"**
- Back end: macchine target, approcci *ad hoc*, problema **"in evoluzione"** (nuove CPU, GPU, . . .)

Sommario

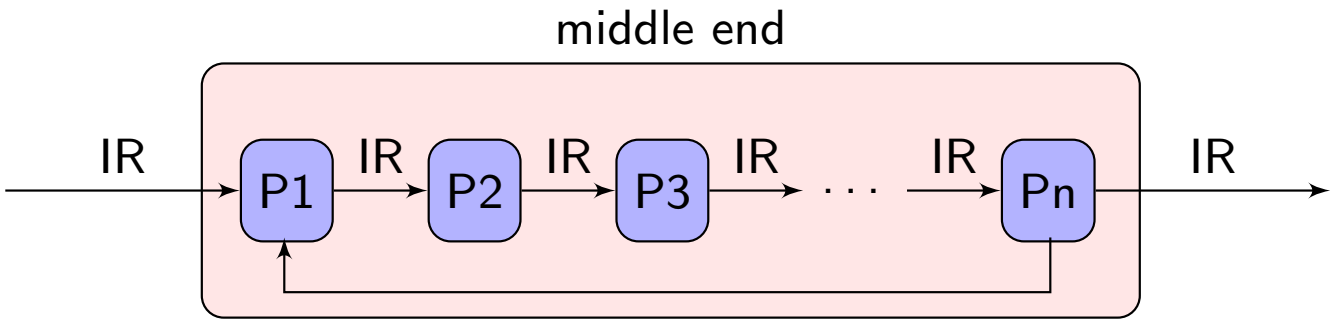
Middle end: ottimizzazione



Middle end

- **Analizza** e **trasforma** il codice IR
- Obiettivo: migliorare il codice
  - metriche classiche: tempo di esecuzione, spazio memoria
  - metriche recenti: consumo energia, risorse hardware, ...
- Deve preservare la **semantica** del programma

Decomposizione del middle end



- Suddivisione in **passi** (possibilmente ripetuti)
- Ogni passo mantiene la semantica del programma
- Un singolo passo implementa una analisi o trasformazione IR
- Un passo può **dipendendere** e/o **invalidare** altri passi
- Sequenza dei passi dipende dal compilatore (e opzioni)

Esempi di passi del middle end

Passi di analisi

- identificazione di valori costanti
- identificazione di codice o valori inutili
- analisi di aliasing

Passi di trasformazione

- propagazione di valori costanti
- rimozione di codice inutile
- inlining di chiamate a funzioni
- loop unrolling

Altri benefici strutturazione compilatore

Più agevole separazione e riuso componenti

- front end riutilizzabili per implementare **interpreti**
- compilazione JIT: passi middle end scelti a run-time
- semplificazione sviluppo di analisi ad hoc (**clang-tidy**)
- integrazione con IDE (editori, debugger, . . .)

Nella prossima lezione

L'analisi lessicale

- Richiami sui linguaggi formali
- Specifica: espressioni regolari
- Esempi di specifica di token per linguaggi di programmazione
- Implementazione: DFSA
- Automazione dell'implementazione: dalle RE ai DFSA
- Esercitazione: il tool **Flex**