

Compilatori

**Programmi che manipolano
Programmi**

Programmi che manipolano Programmi

Cosa significano queste scritture?

- `cc -w -o code.exe ex.yy.c`
- `code.exe A.file B.file <In >Out`
- `javac -classpath /docu/XML/Xerces/Xparse.jar -v MyDocument.xml`
- **Cosa sono** `ex.yy.c`, `MyDocument.xml` ?
- **Cosa sono** `cc`, `javac` ?
- **Questi strumenti sono programmi:**
 - **Come si costruiscono ?**
 - **In quale linguaggio sono scritti?**

Quando il programma usa un'interfaccia grafica per gestire I/O

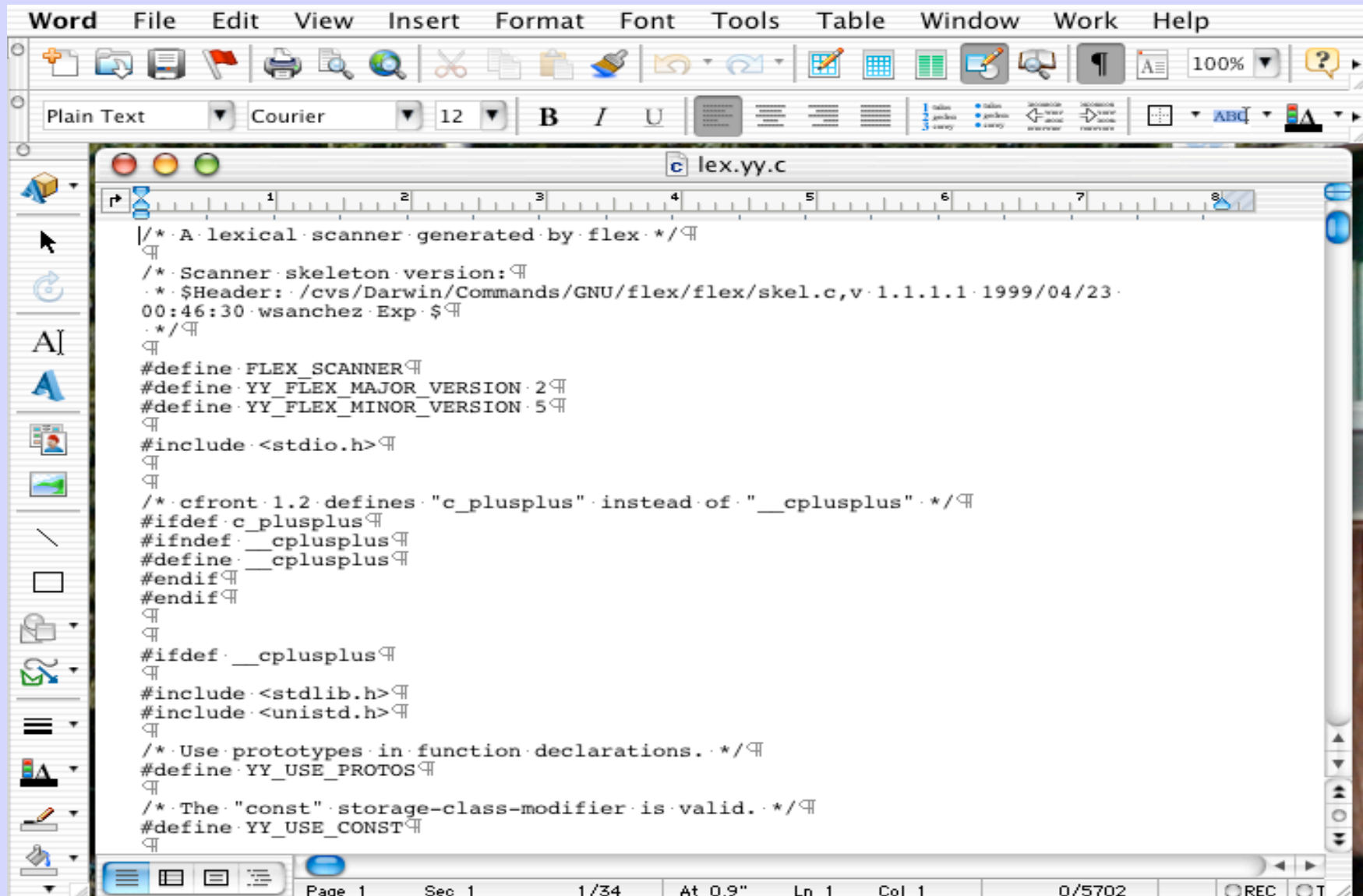
The image shows a screenshot of an IDE with two windows. The top window, titled 'Build: Parse1 -', shows the build process with the command: `oper/Makefiles/pbx_jamfiles /ProjectBuilderJambase JAMFI LF=- build_ACTION=build_D`. The bottom window, titled 'Run: Parse1 -', shows a runtime error: `Exception in thread "main" java.lang.NoClassDefFoundError: /Users/marcobel/Desktop/XML/contents/xml`. The main editor window, titled 'Parse1 - Parse1.java', shows the source code for `Parse1.java`. The code includes imports for `java.util.*`, `java.io.*`, `java.io.IOException`, `org.xml.sax.XMLReader`, `org.apache.xerces.parsers.SAXParser`, and `org.xml.sax.SAXException`. It also contains a Javadoc comment for `SaxParserDemo` and a reference to Brett McLaughlin.

```
//
import java.util.*;
import java.io.*;
import java.io.IOException;
import org.xml.sax.XMLReader;

//importa l'implementazione di
XMLReader accessibile
import org.apache.xerces.parsers.
SAXParser;
import org.xml.sax.SAXException;;

/**
 * <b><code>SaxParserDemo</code></b>
 * prende e parsea un file XML usando
SAX e mostrando
 * le callbacks nel ciclo di parsing
 *
 * @author
 * <a
href="mailto:brettmclaughlin@earthlink.
net">Brett McLaughlin</a>
```

Quando il programma fornisce un insieme di comandi per gestire I/O



```
Word  File  Edit  View  Insert  Format  Font  Tools  Table  Window  Work  Help
Plain Text  Courier  12  B  I  U
lex.yy.c
/* A lexical scanner generated by flex. */
Scanner skeleton version:
$Header: /cvs/Darwin/Commands/GNU/flex/flex/skel.c,v 1.1.1.1 1999/04/23
00:46:30 wsanchez Exp $
*/
#define FLEX_SCANNER
#define YY_FLEX_MAJOR_VERSION 2
#define YY_FLEX_MINOR_VERSION 5
#include <stdio.h>
/* cfront 1.2 defines "c_plusplus" instead of "__cplusplus" */
#ifdef c_plusplus
#ifndef __cplusplus
#define __cplusplus
#endif
#endif
#ifdef __cplusplus
#include <stdlib.h>
#include <unistd.h>
/* Use prototypes in function declarations. */
#define YY_USE_PROTOS
/* The "const" storage-class-modifier is valid. */
#define YY_USE_CONST
```

Page 1 Sec 1 1/34 At 0.9" Ln 1 Col 1 0/5702 REC OT

In questo corso: le tecniche

tecniche di base per la costruzione di strumenti per macchine astratte

Tecniche: automi (a stati finiti, pushdown)
syntax-driven translations
attribute grammars
schemi di traduzione
tecniche di attraversamento
trasformazioni invarianti (ottimizzazioni)
...

In questo corso: metodologie, strumenti

Metodologie: semantic attachment
abstract interpretation
metavalutazione
valutazione parziale
sintesi

...

Strumenti: analizzatori sintattici
analizzatori semantici
text formatters
editor sintattici
generatori di strumenti di analisi
interpreti
debuggers
compilatori

...

Generalità

- **Linguaggio e Macchina Astratta**
- **M.A.: struttura e stati dell'esecutore**
- **Costruire M.A.: Interprete, Compilatore**
- **Interprete: dentro**
- **Compilatore: il supporto RTS**
- **Compilatore: Macchine di sviluppo e Gerarchia**
- **Macchine Intermedie: costruzioni miste**

Definizioni:

linguaggio e formalismo

linguaggio di (programmazione) =

= formalismo per esprimere
(applicazioni di funzioni calcolabili)

formalismo = **sintassi** (*forma* delle costruzioni permesse)
+
semantica (*significato* loro associato)

Esempio: Linguaggio di Programmazione

$L = \langle S, SEM \rangle$ e' un linguaggio di progr.

1) per ogni $P \in S$, $SEM(P) \in \{N \rightarrow N\}$

2) per ogni $f \in \{N \rightarrow N\}$, esiste $P \in S$, tale che:
per ogni $n \in N$, $f(n) = SEM(P)(n)$

[dove: $\{N \rightarrow N\} =$ funzioni calcolabili]

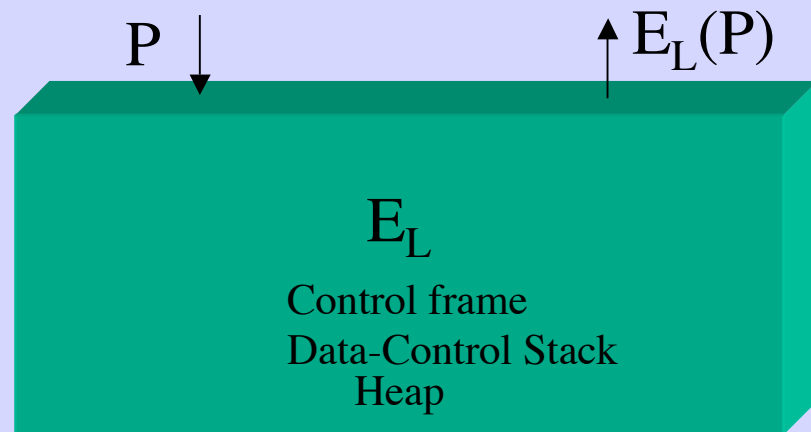
Definizioni:

Macchina Astratta

Macchina astratta =
Linguaggio ($L = \langle S, SEM \rangle$)
+
Esecutore (E_L)

Per ogni $P \in S$,
 $SEM(P) \approx E_L(P)$

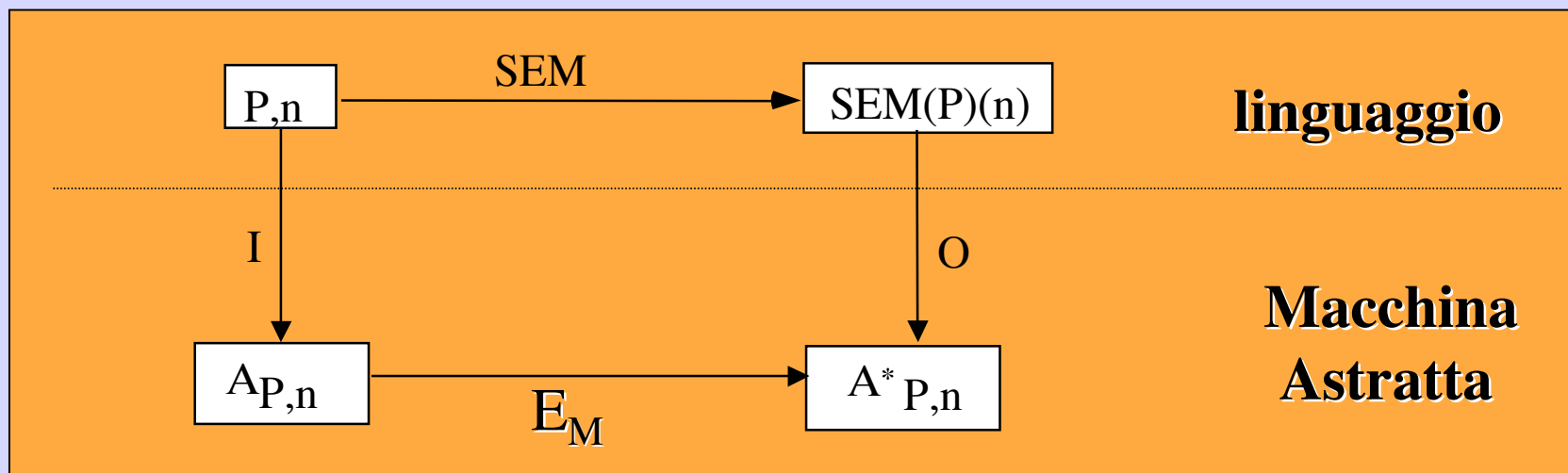
Java Virtual Machine
Landin's SECD



Esempio: Macchina Astratta, Linguaggio ed Esecutore

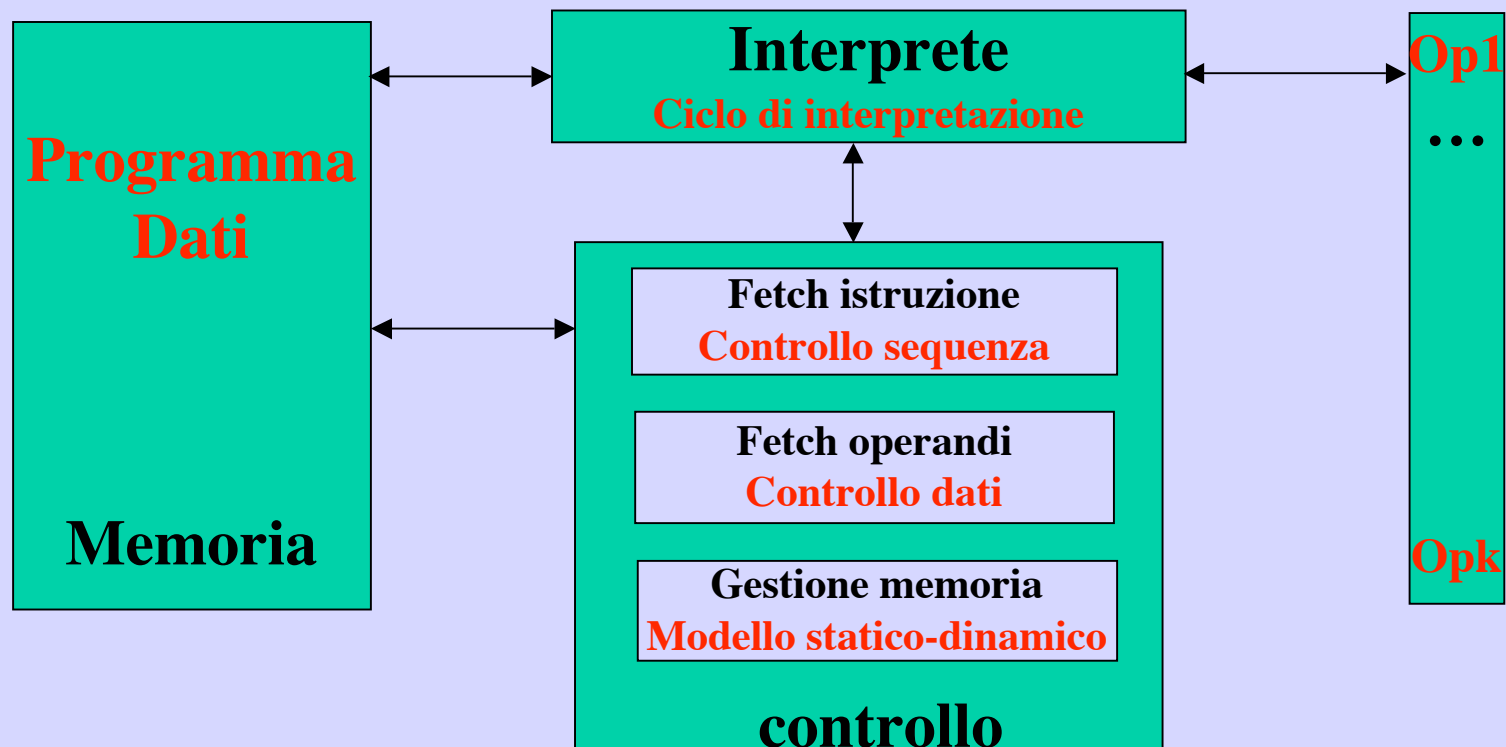
- $L_M = \langle S, SEM \rangle$ è un linguaggio di programmazione
- E_M è un esecutore di applicazioni di programmi
- $M = \langle L_M, E_M \rangle$

$E_M: A \rightarrow A^*$
per ogni $P \in S, n \in \mathbb{N}$



[dove I,O funzioni da A e A* risp.]

Macchina Astratta: Struttura e Stati dell'Esecutore



Macchina astratta - Esecutore

Macchina Astratta: Memoria, Controllo

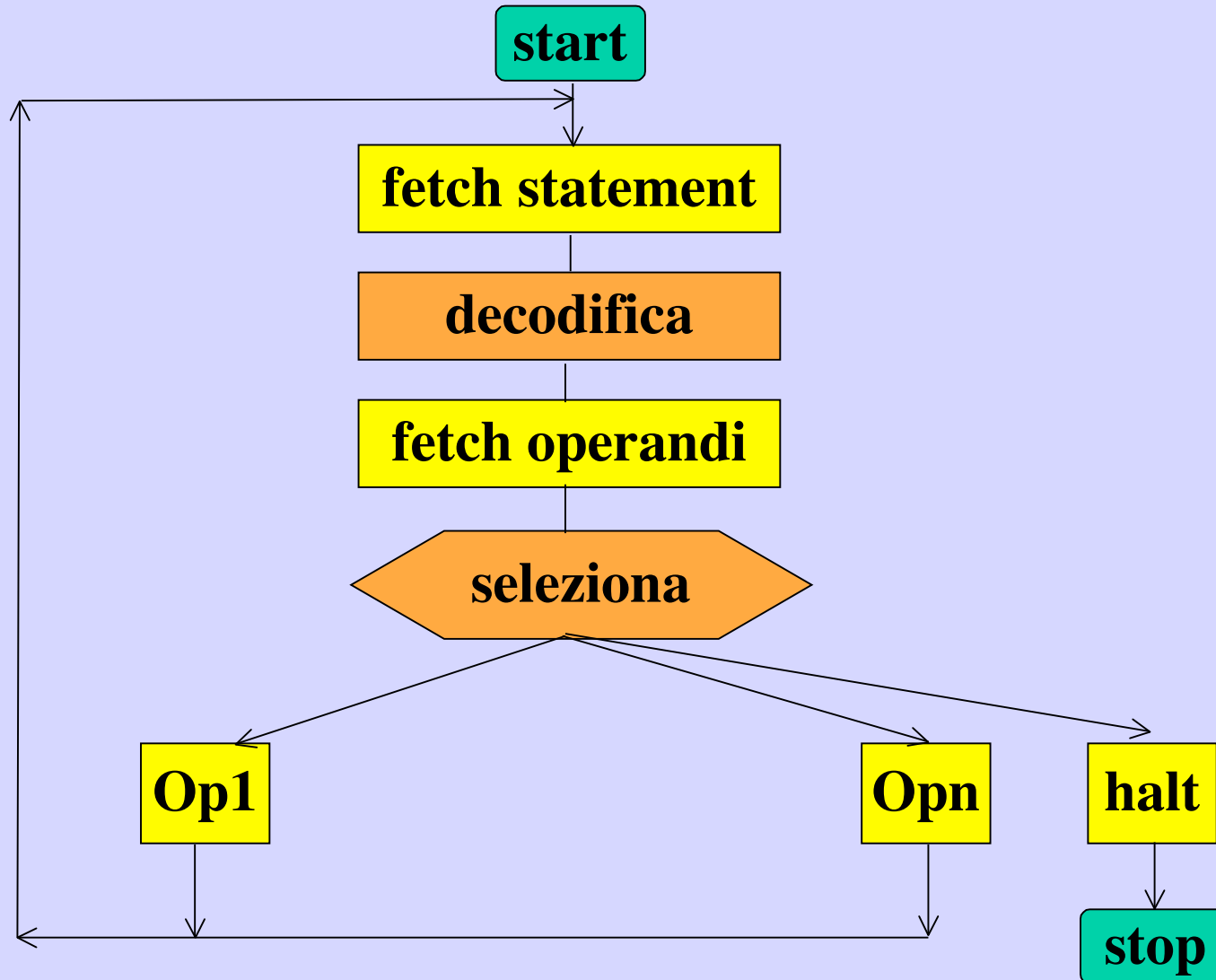
Memoria: strutturata secondo un modello che dipende dal linguaggio della macchina

- *array di parole, registri, stack*
- **heap** - L. con allocazione dinamica (Pascal, C, C++, ..., Java,
- **grafo** - L. con condivisione di strutture (*funzionali*)

Controllo: gestisce lo stato della macchina

- trova il successivo *statement* o *espressione*
- trova i dati su cui tale *stat.* o *espr.* opera
- gestisce la memoria

Macchina Astratta: Interprete - ciclo di interpretazione



Costruire Macchine Astratte

Utilizziamo macchine astratte già definite

- Sia $L_0=(S_0,SEM_0)$ il nostro linguaggio
- Sia $M_1=(L_1=<S_1,SEM_1>,E_{L_1})$ una macchina

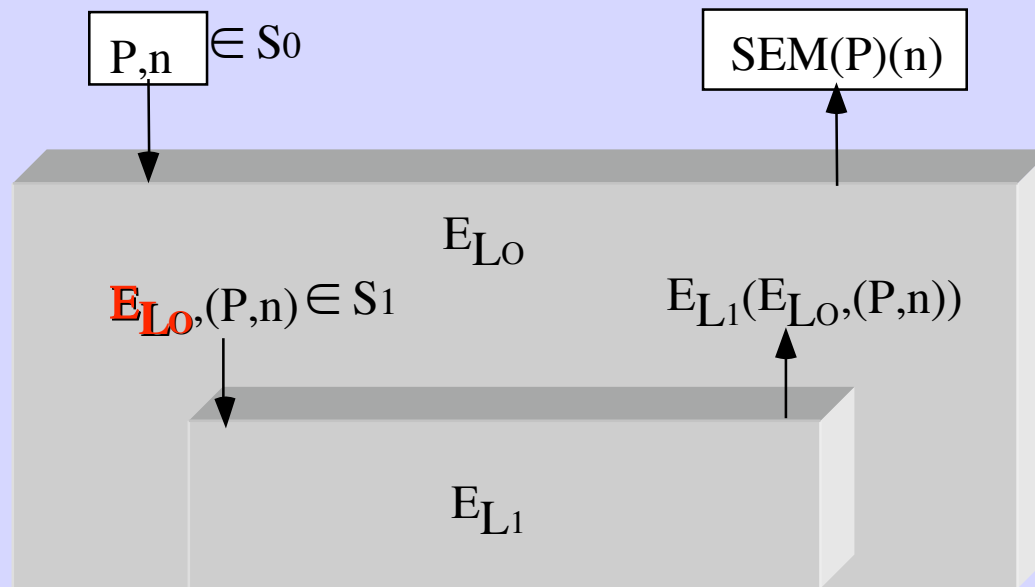
interprete

definiamo l'esecutore E_{L_0} come programma di L_1 .

compilatore

trasformiamo ogni struttura (programma) di L_0
in una equivalente struttura (programma) di L_1 .

Interprete



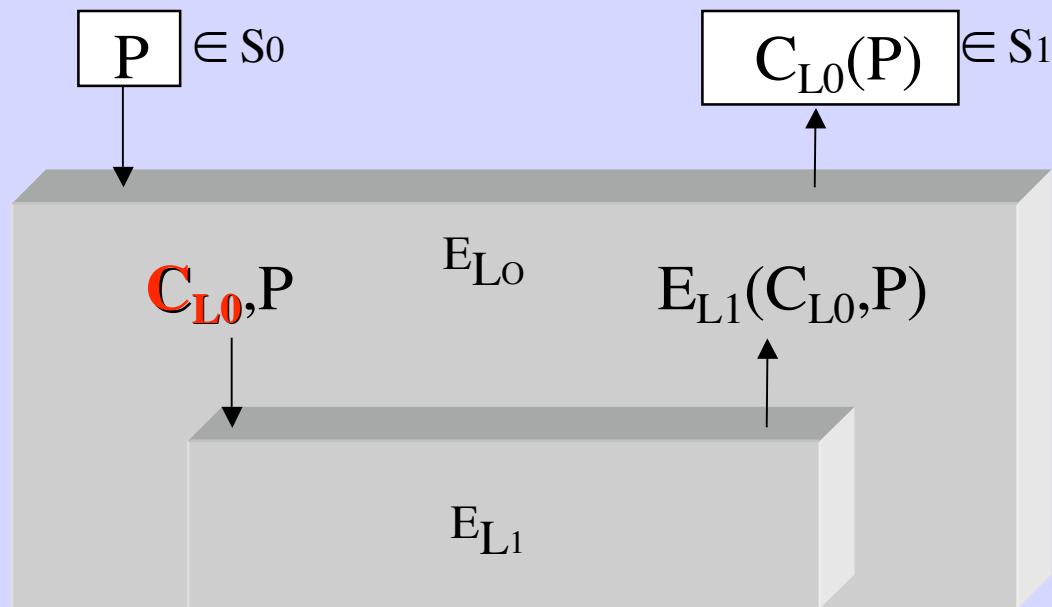
Eeguire una **APPLICAZIONE** (P, n) di L_0 , consiste:
nell'eseguire l'applicazione $E_{L_0}, (P, n)$ di L_1

Interprete: dentro E_{L0}

Una collezione di procedure che realizzano:

- i passi (*decodifica*) del ciclo di **interpretazione** per L0
- il modello di **memoria** di dati e programmi di L0
- l'unità di **controllo** per fetch di codice e di dati di L0
- un implementazione delle **primitive** di L0

Compilatore

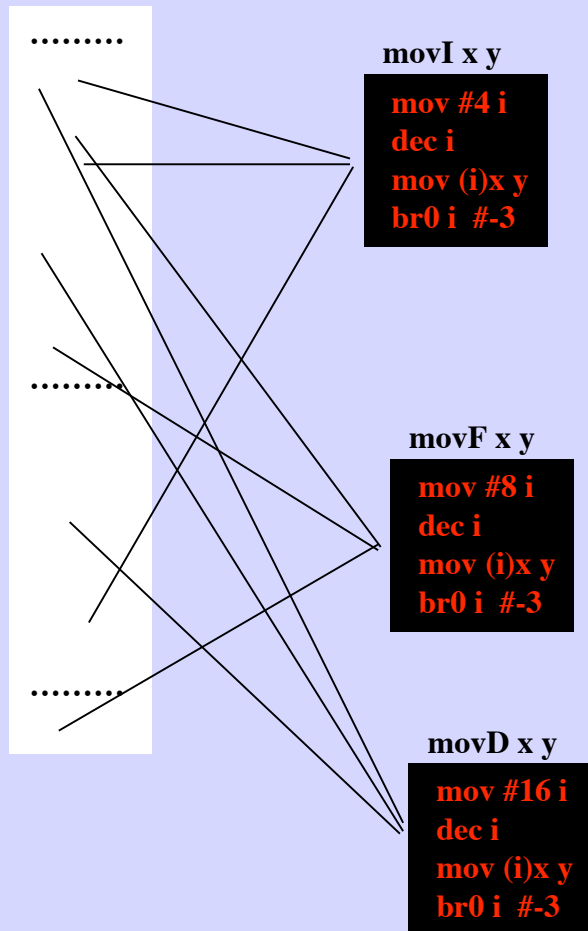


Il compilatore non opera su applicazioni bensì
su strutture (programmi: P)

C_{L0} preserva la semantica:

$$SEM_0(P) = SEM_1(C_{L0}(P))$$

Il Run Time Support un semplice esempio



Quando il modello della memoria di
L0 ha parole di 4, 8, 18 byte e
L1 ha solo parole di 1 byte

1000 assegnamenti richiedono
4x1000 righe di codice

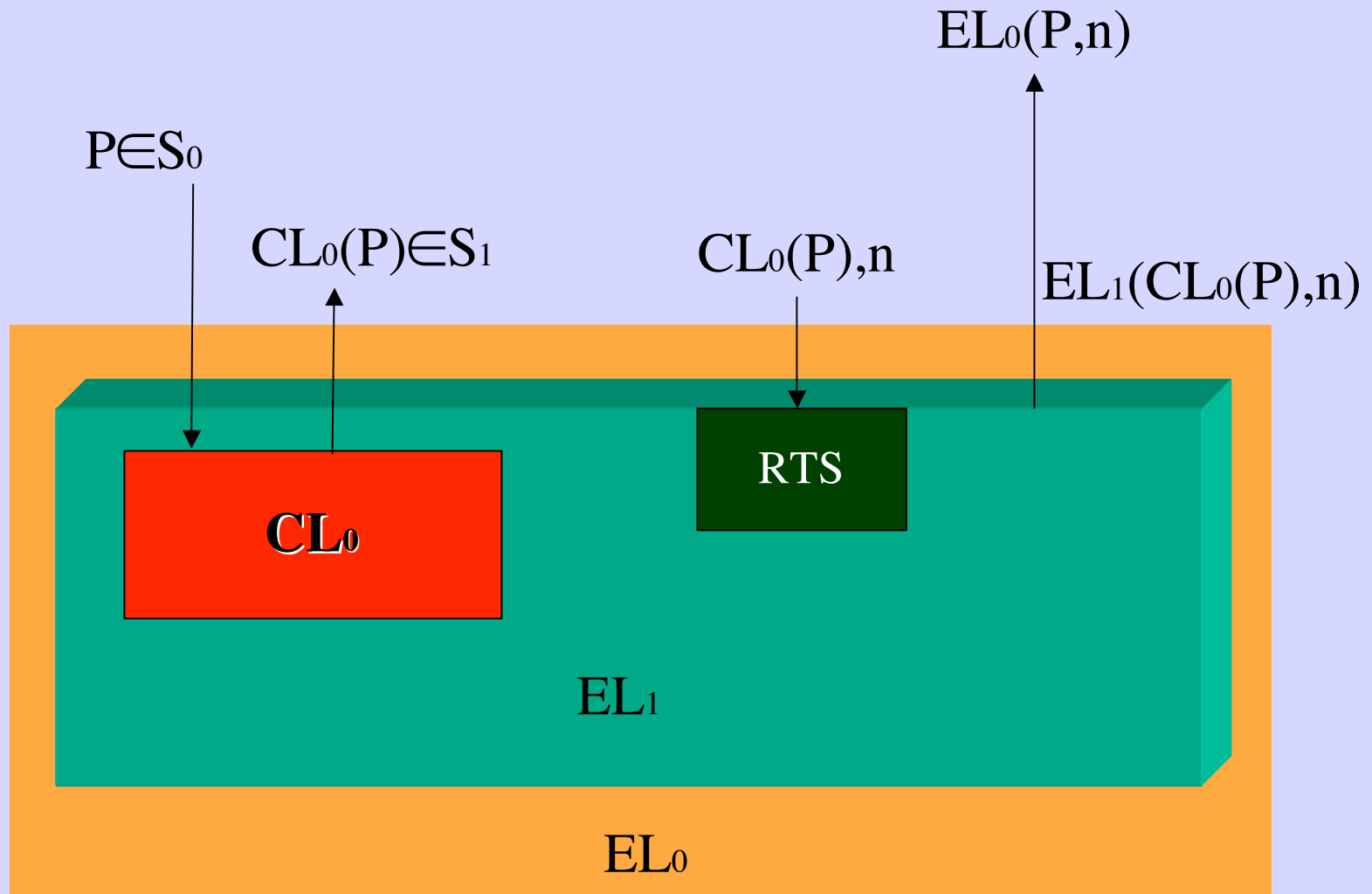
Compilatore: Il Run Time Support

- Non dipende dallo specifico programma compilato
- Utilizzabile dall'oggetto di ogni sorgente

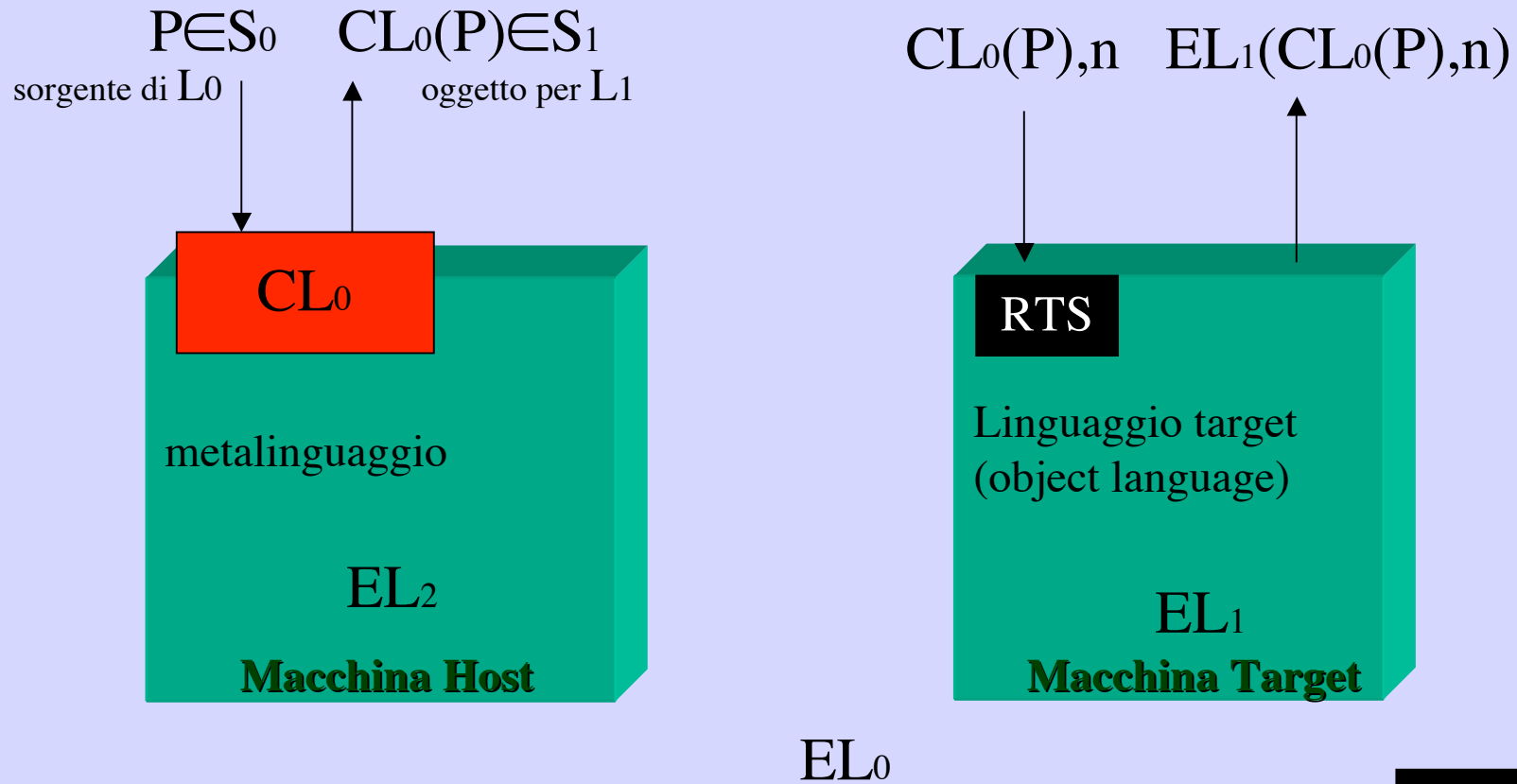
Una collezione di procedure che realizzano:

- il modello di **memoria** di dati e programmi di L0
- una implementazione delle **primitive** di L0

Compilatore: La Macchina Sottostante



Compilatore: La Macchina di Sviluppo



Gerarchia di Macchine

Gerarchia di Macchine

- riduce ad ogni livello l'espressività
- semplifica la costruzione della macchina di un linguaggio molto espressivo

Linguaggio meno espressivo ha:

- molti costrutti ma elementari
- un esecutore piú semplice

Quando la macchina target è concreta

- nessuna differenza concettuale tra astratta e concreta
- abbiamo però un esecutore effettivo per ogni macchina della gerarchia

Classi di Macchine Concrete

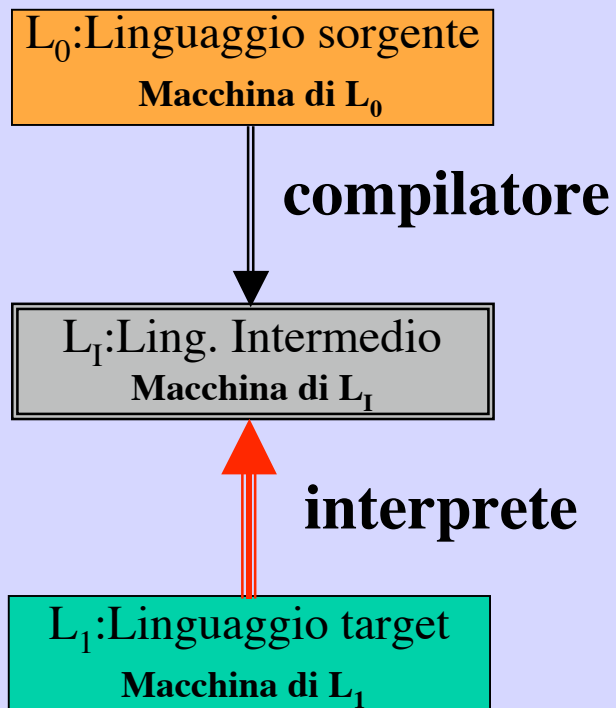
In corrispondenza alle molte classi di linguaggi

- Imperativi
- Applicativi
- Object oriented

Differiscono per il linguaggio e di conseguenza per:

- struttura dello stato, ovvero:
 - modello di memoria
 - metodi di fetch e di decodifica
 - operazioni primitive

Macchina Intermedia: Costruzioni Miste



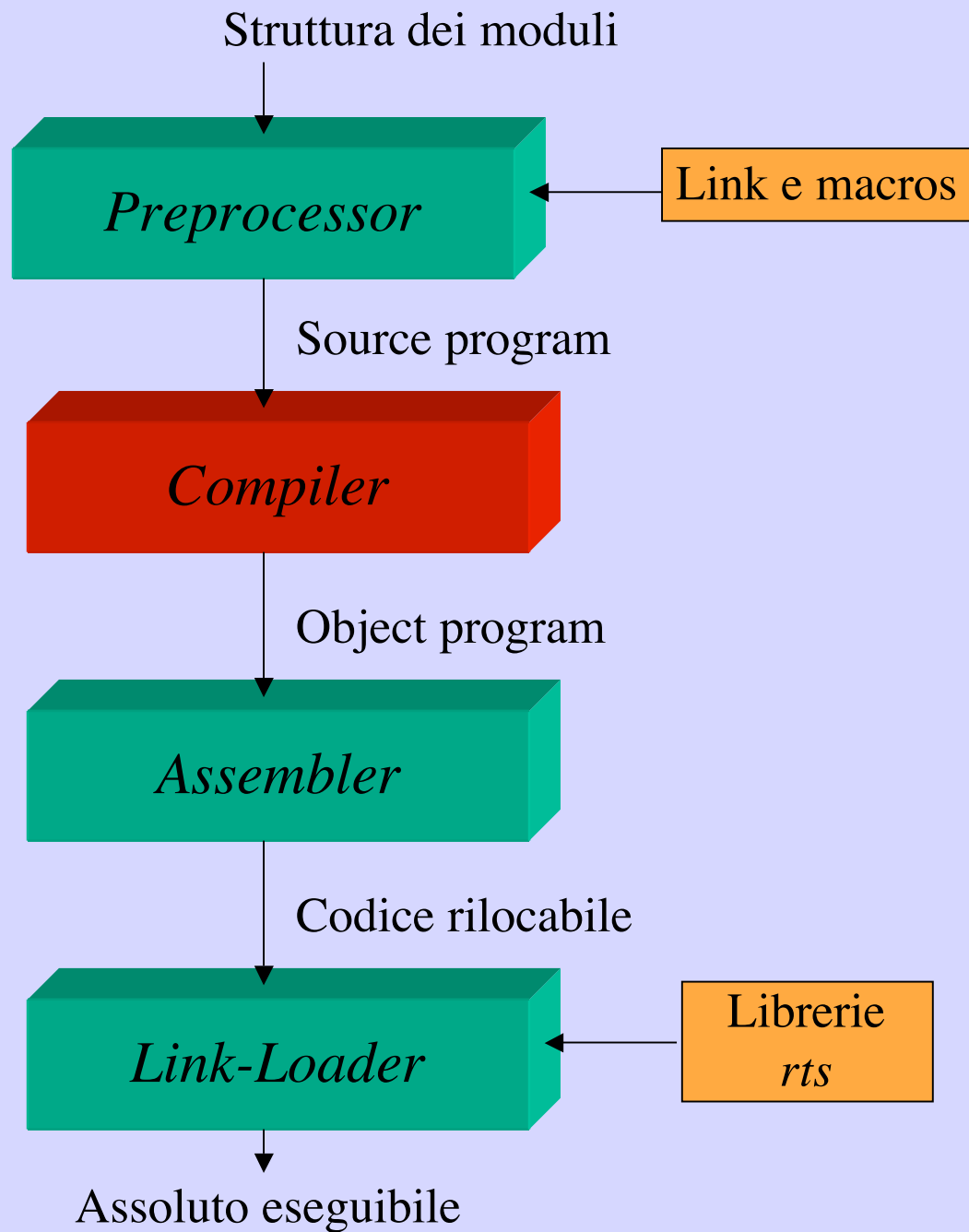
Vantaggi:

- **sviluppo ridotto**
- **portabilità aumentata**
- **dimensione c. oggetto:**
 - **occ. memoria**
 - **tempo esecuzione**

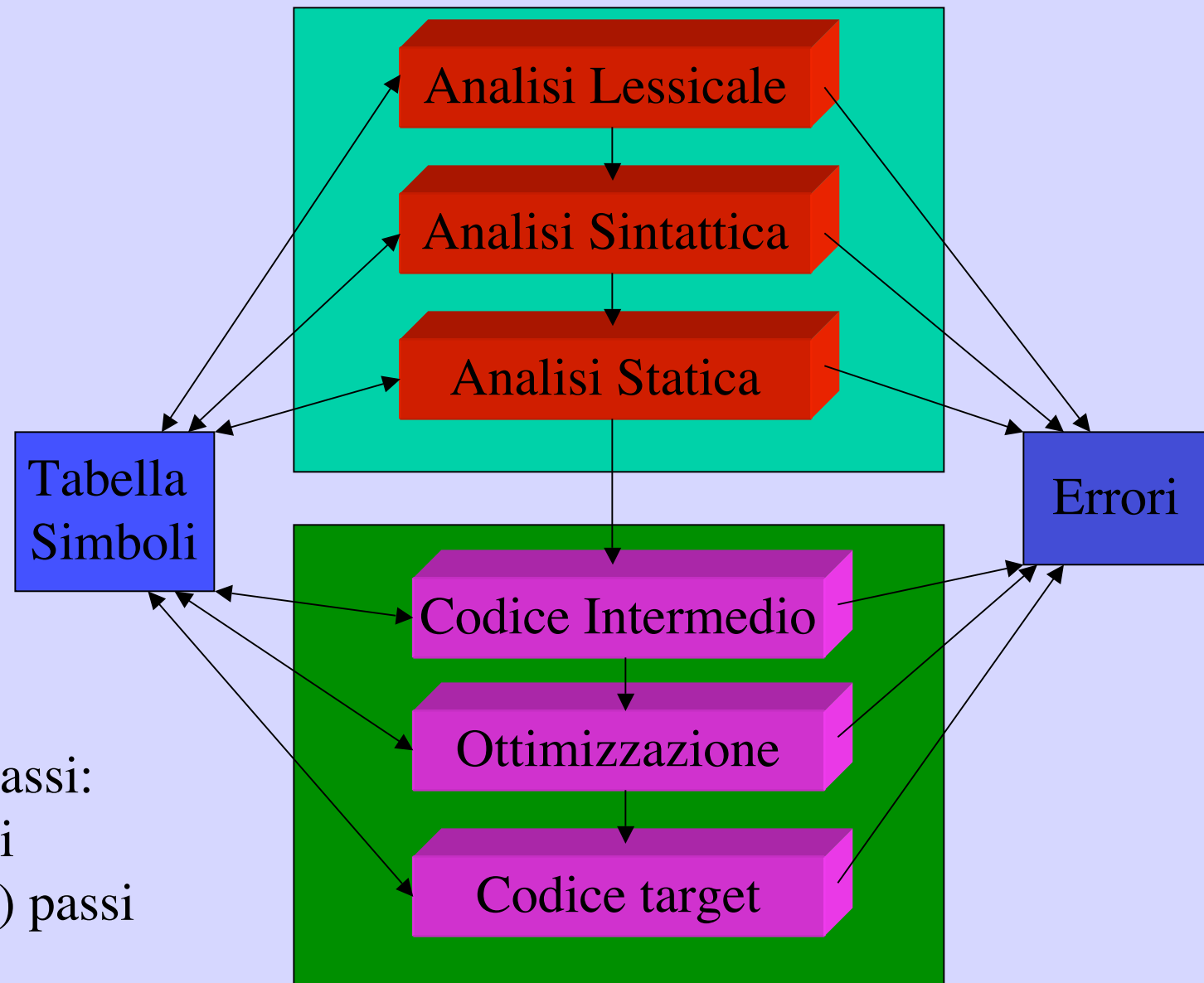
Compilatore, Interprete: contesto, struttura componenti

- **contesto operativo: preprocessing e loading**
- **Compilatore: Struttura, fasi e passi**
- **Interprete: Struttura standard**
- **Visitiamo le fasi: un esempio**
- **Compiler-Compiler: semplifichiamo la stesura**
- **Bootstrapping**

**Contesto del
Compilatore:
Font-end
Back-end**

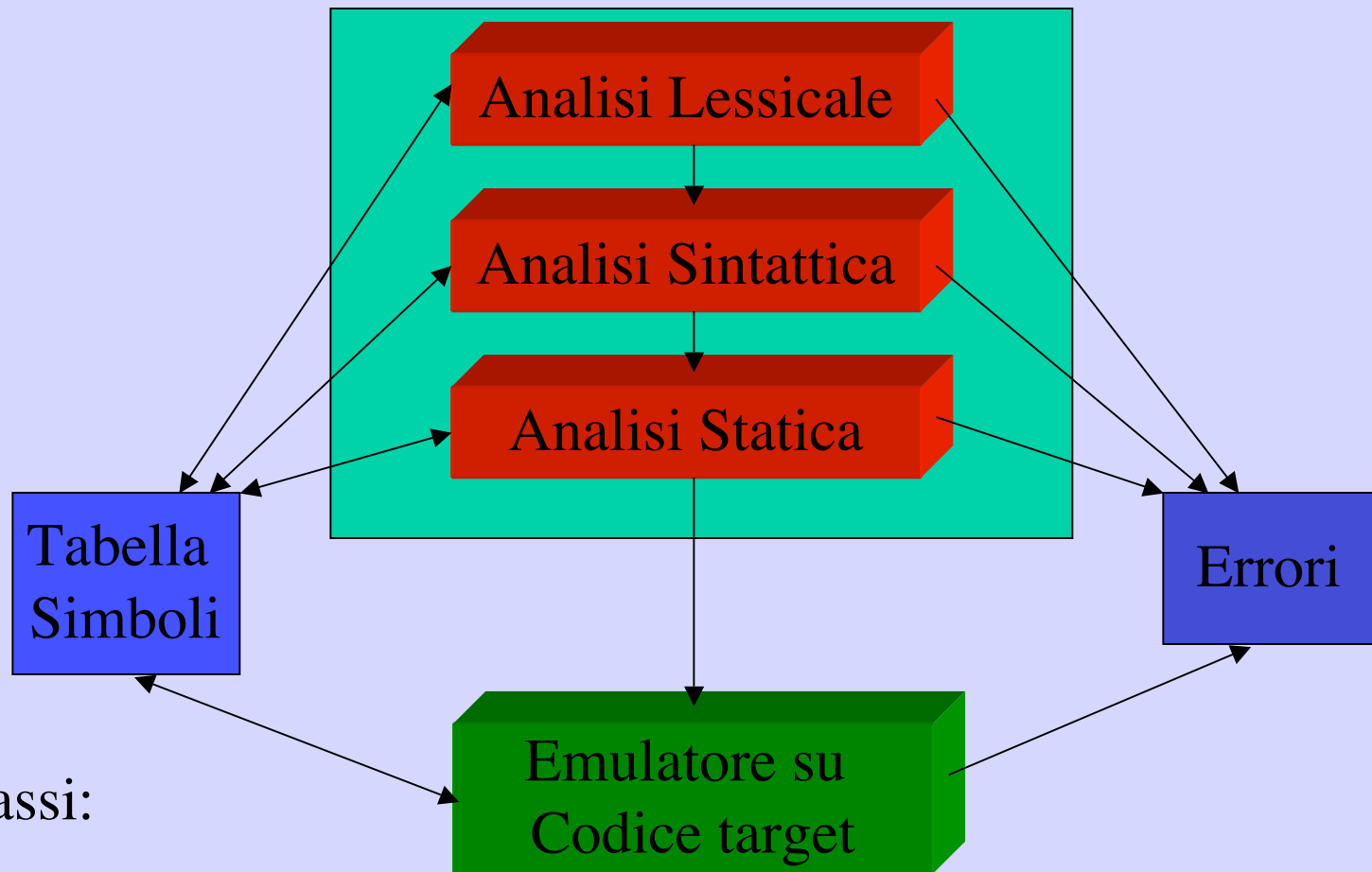


Compilatore: struttura, fasi e passi



Fasi e Passi:
6 fasi
 $k(\geq 1)$ passi

Interprete: La struttura standard



Fasi e Passi:

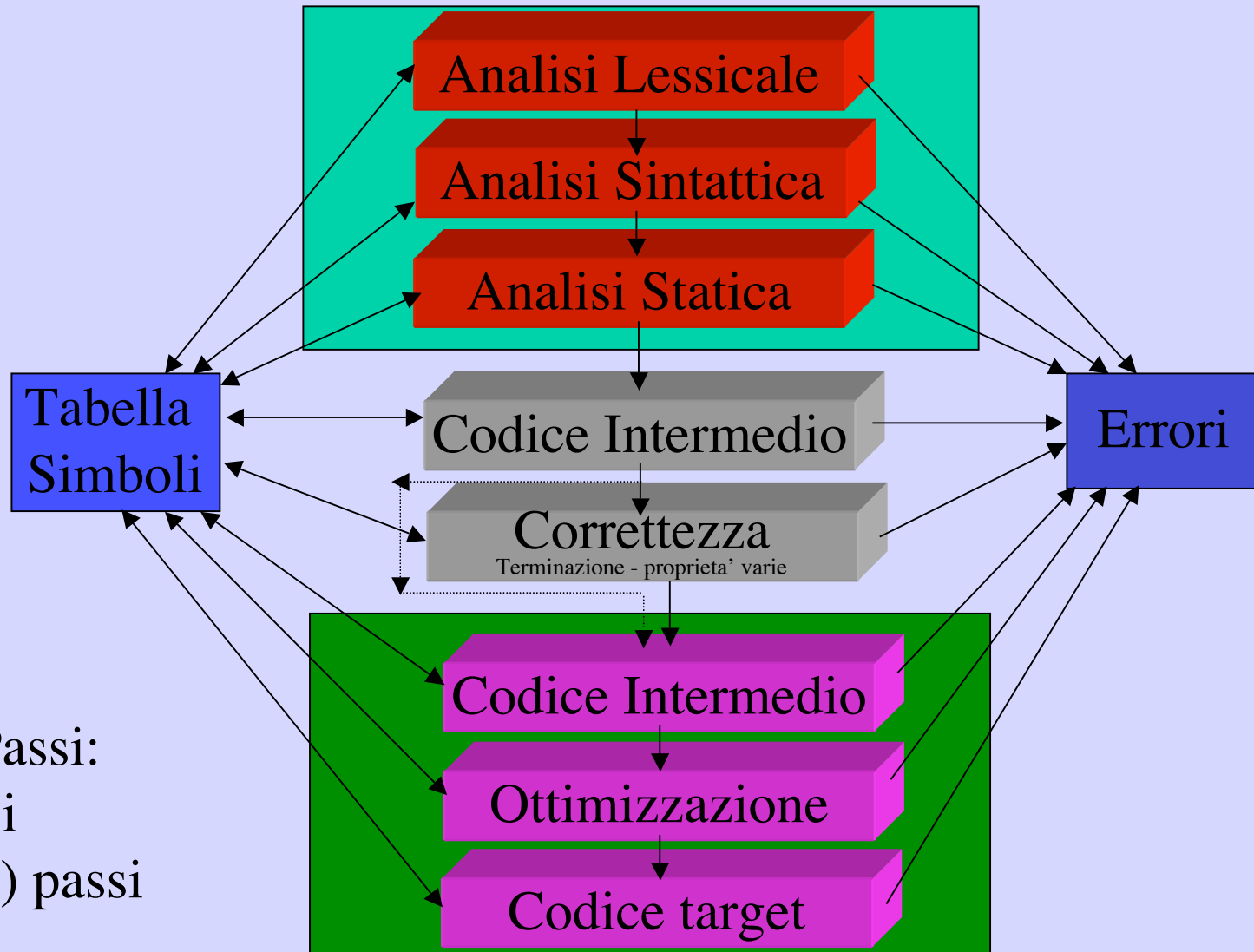
4 fasi

$k(\geq 1)$ passi

Visitiamo le fasi della compilazione attraverso un esempio

dal testo - fig. 1.10 pag. 13

Compilatore: Una struttura per analisi di correttezza avanzate



Fasi e Passi:
8 fasi
 $k(\geq 1)$ passi

Compiler-Compiler: ridurre i metalinguaggi e semplificare la stesura di un compilatore

Lo sviluppo di un compilatore (interprete), da un linguaggio L_0 a L_t , coinvolge altri linguaggi L_m .

I metalinguaggi sono utilizzati per esprimere le procedure di analisi e traduzione, e condizionano il compilatore che può essere eseguito solo sul meta scelto

Distinguere tra: $C_{0 \rightarrow t \downarrow m}$ e $C_{0 \rightarrow t \downarrow n}$

Combiniamo *interprete* e *compilatore*

Bootstrapping

- costruiamo un interprete $E_{0 \downarrow m}$ (che valuta programmi di L_0 su una macchina M_m): strumento di sviluppo
- costruiamo compilatore $C_{0 \rightarrow t \downarrow 0}$: Il compilatore è scritto nel linguaggio L_0 stesso.
- eseguiamo: $E_{0 \downarrow m}(C_{0 \rightarrow t \downarrow 0})(C_{0 \rightarrow t \downarrow 0})$ otteniamo $C_{0 \rightarrow t \downarrow t}$

Bootstrapping ($m \leq t$)

Il prodotto è ora indipendente dal meta