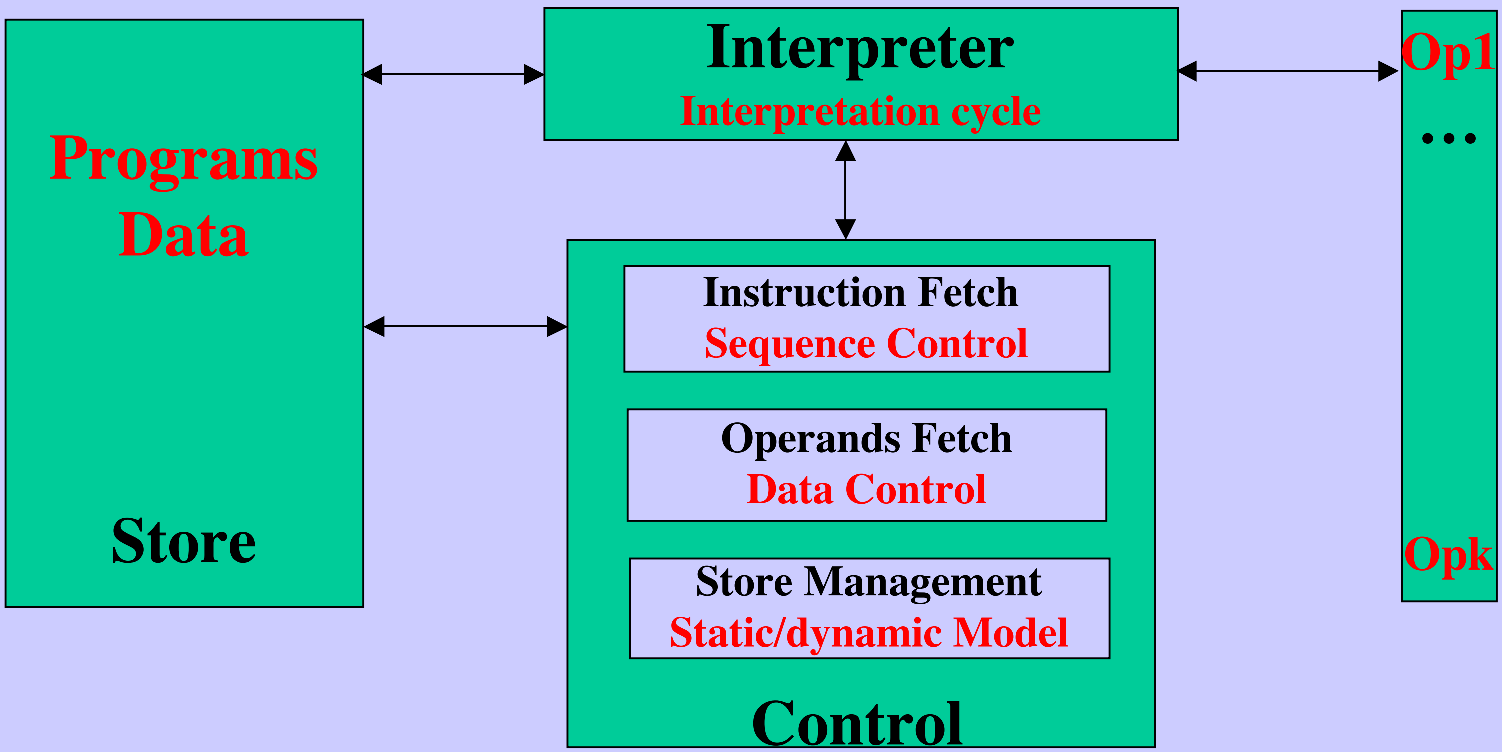
MA: Structure e Executor States



Abstract Machine - Machine Structure

MA:
Store, Control

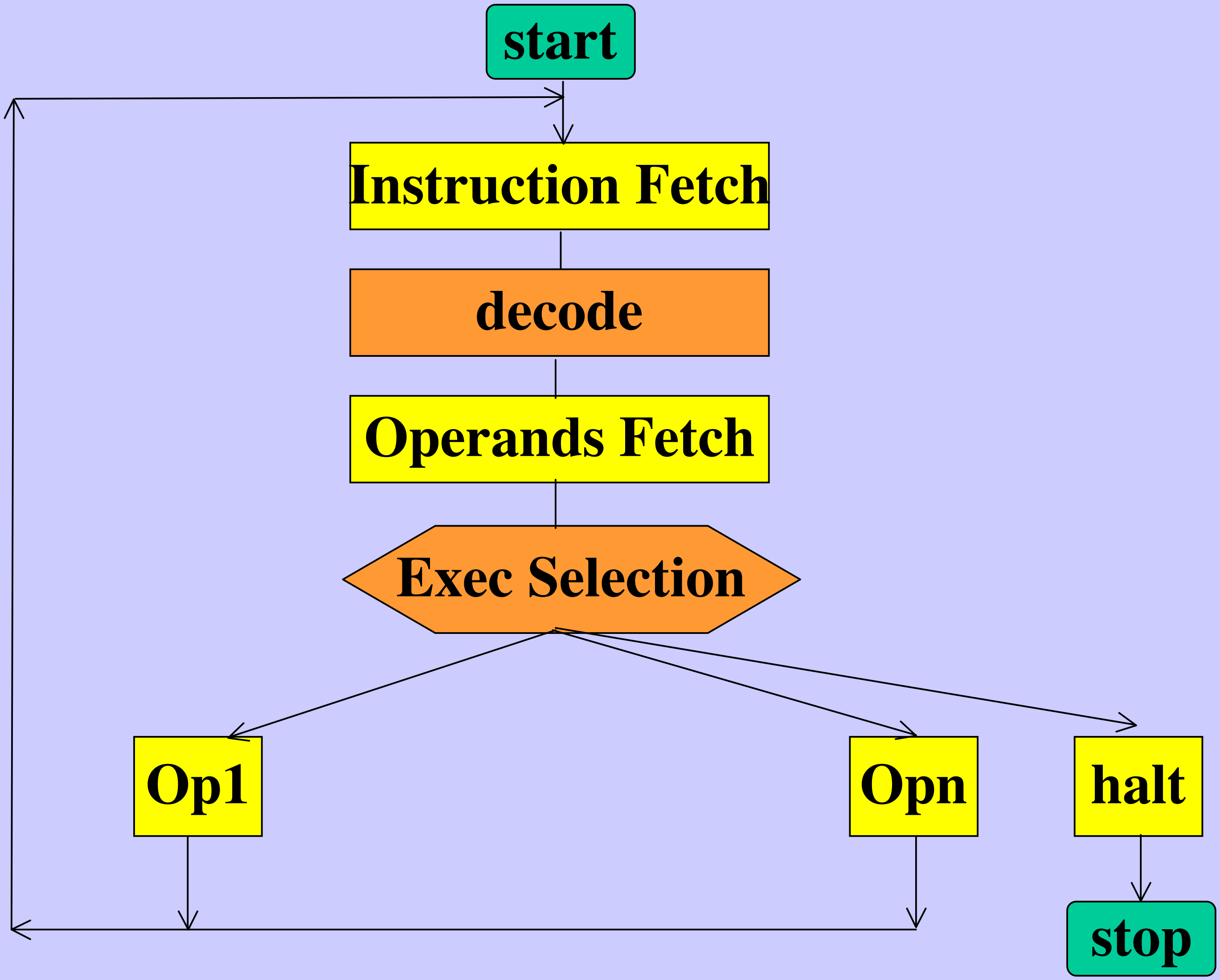
Store: **It is structured according to a model that relies on the specific features of the Machine Language**

- ***arrays of words, registers, stacks***
- **heap -** for dynamic allocation (Pascal, C, C++, …,Java,
- ***graph -*** for structure sharing (*functional languages*)

Control: **It handles the Executor States:**

- **finds the next *statement* or expression**
- **finds the *stat*. or *espr*. data**
- **updates store**

MA:
Elementary Execution Cycle



On Building MAs

The Problem: Given a new language, L0, how a MA for L0 can be built?

- Let $L_0=(S_0,SEM_0)$
- Let $M_1= (L_1=<S_1,SEM_1>,\mathbf{E_{L_1}})$ an available MA where (obviously) $L0 \neq L_1$

Two Main Approaches:

Intepreter

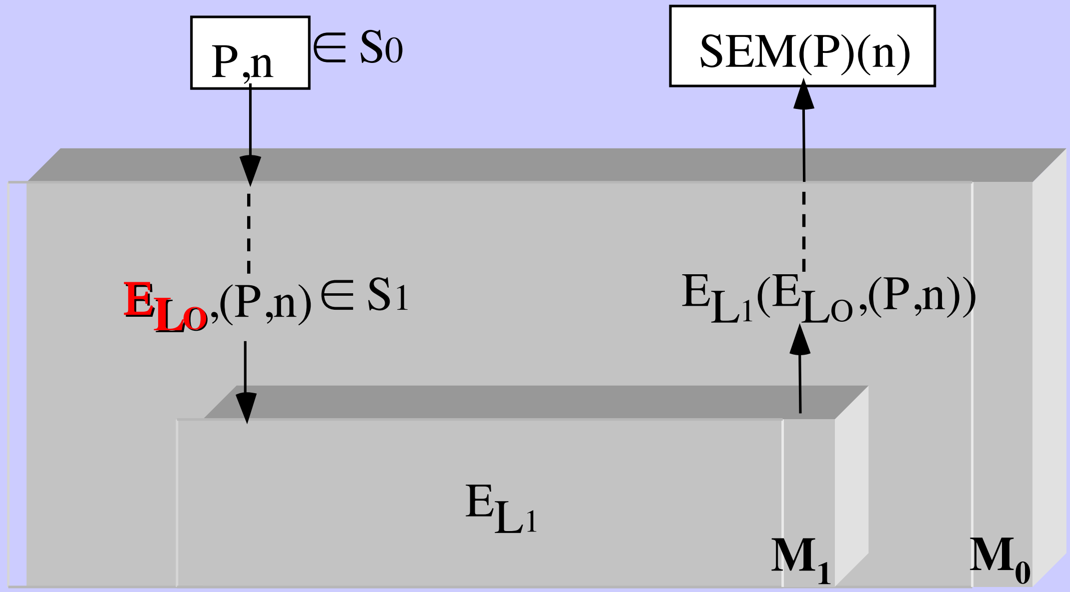
defines executor E_{L0} as a program of L_1 which describes the behaviour of the L0 structures using E_{L1}

Compiler

Maps each structure (program) of L_0 into an equivalent structure (program) of L_1.

L0=(S0,SEM0)
M_1= (L_1=<S_1,SEM_1>,E_{L_1})

Interpreter



Running **Application** (P,n) of L0 consists in

Running **Application** (**E_{LO}**,(P,n))

provided:

+ **E_{LO}** is an interpreter, in L1, of L0
+ (P,n) has a suitable representation datum in L1

Interpreter: Example

- $L_0=(S_0,SEM_0)$ is *C-like*
- Let $M_1= (L_1=<S_1,SEM_1>,\mathbf{E_{L_1}})$ an available
  where L_1 is a *3-address code Language*

The interpretation sequence provided by E_{L_0} in M_1 for:

while x {x=x+y*z}

could be expressed in a form like:

call E_{L_0} (while x {x=x+y*z})

and generates an *execution step sequence* like:

find locx
br @locx …
find *valy*
find *valz*
find *freeR0*
put* *valy valz* into *freeR0*
put+ @*locx* @*freeR0* into *locx*
call E_{L_0} (while x {x=x+y*z})

Black steps are machine language statements. Red colored steps are meta-code which may describe machine states/action or lead to generation of new execution steps

Interpreter: Inside E_{L0}

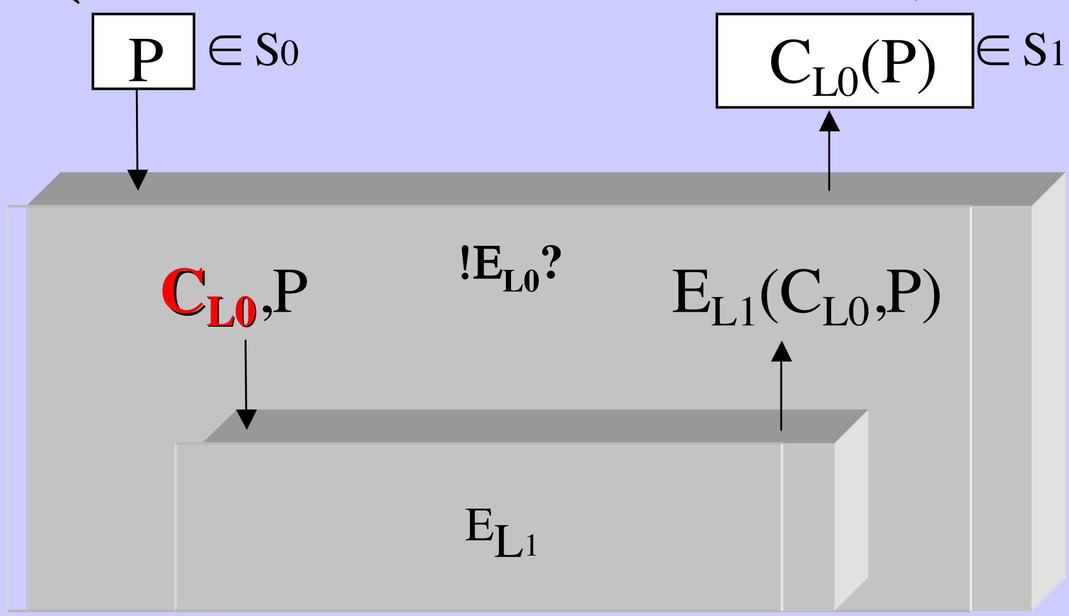
A collection of, suitably correlated, procedures (and supporting structures) that implement:

- **The Steps (*Fetch-Decode-Execute*) of the Interpretation Cycle of the MA for L0**
- **The Store Model of data and programs of L0**
- **The Control Unit for data and code access of L0**
- **The Primitive Operators and Data of L0**

It is called Source program

Compiler

It is called Object program



- A compiler do not apply to **application (P,n)**
- Instead, it deals directly with **programs P**

C_{L_O} preserves the semantics:

$$SEM_O(P) = SEM_1(C_{L_O}(P))$$

Proving Compiler Correctness is clearer and more evident than Proving Interpreter Correcteness

$C_{0\to1\downarrow1}$

Compiler: Example

- L_0=(S_0,SEM_0) is *C*
- Let M_1= (L_1=<S_1,SEM_1>,$\mathbf{E_{L_1}}$) an available
  where L_1 is a *3-address code Language*

The compilation provided by E_{L_0} in M_1 for:

while x {x=x+y*z}

generates a L_1 code like:

find locx
br @locx 7
find *valy*
find *valz*
find *freeR0*
put* *valy valz* into *freeR0*
put+ *@locx @freeR0* into *locx*
jmp- 7

Compiled code is more efficient and less time consuming than Intepreted code

Compiler: Run Time Support

Properties

- It does not depend from the form of the *source* to be compiled
- It may be used from the *object* of possibly, any *source*

Characteristics

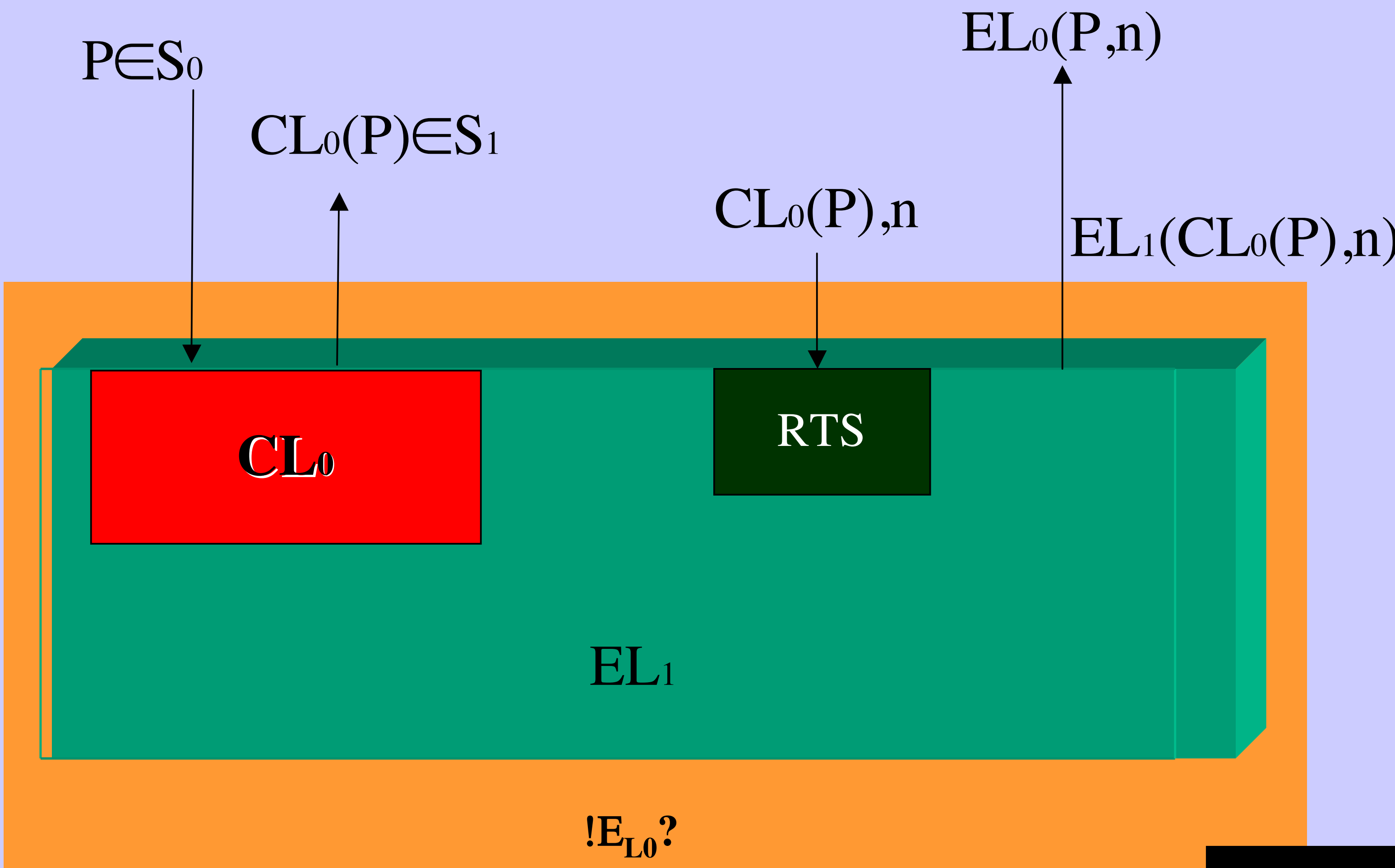
RTS = *Collection of data structures and procedures* which are written in the object language and implement:

- **Store** Model for *data and programs*
- **Primitives** *data and operations*
- **Control** Model for *Activation Record and Control Transfer*

of the source language

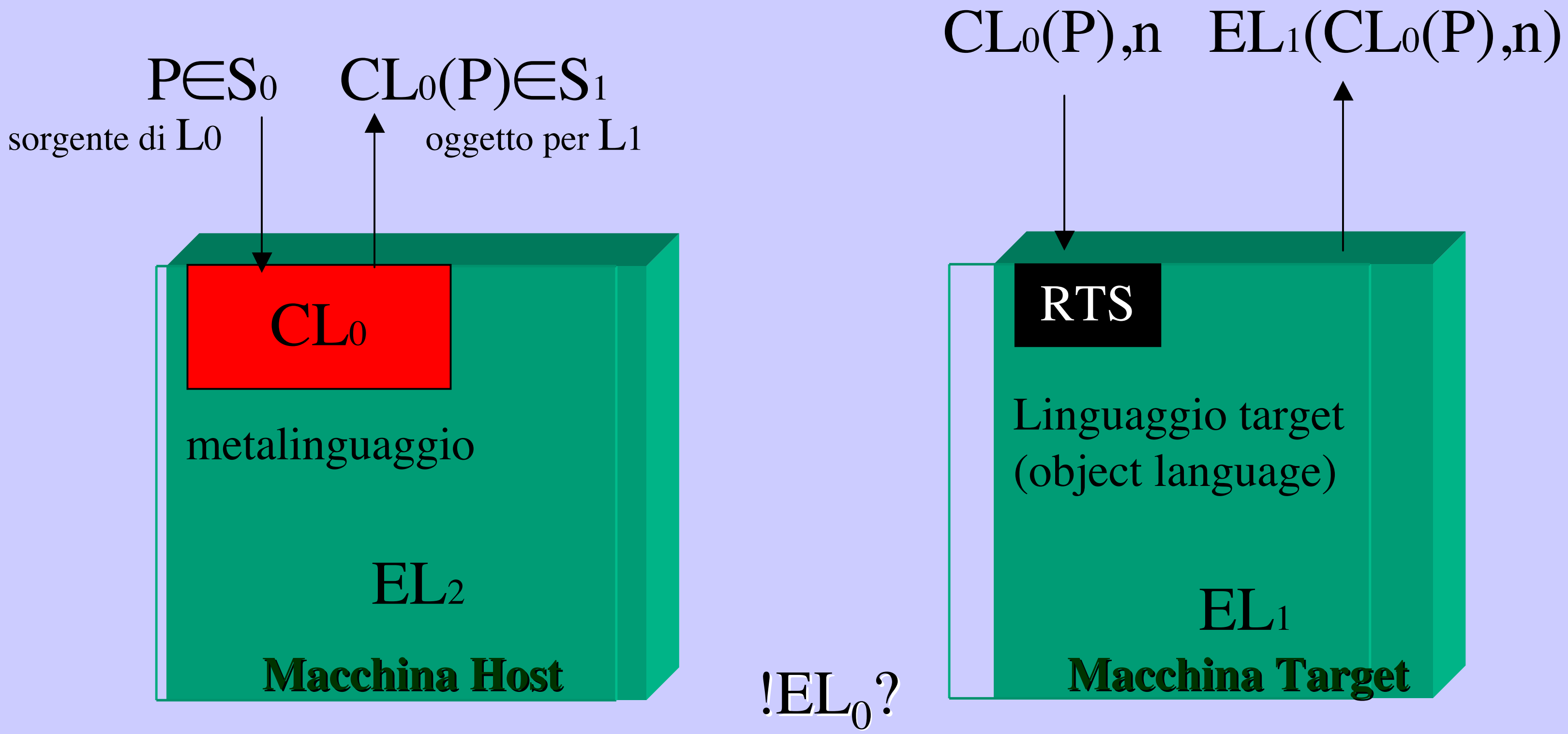
RTS is conceptually the same to that present in Interpreters for modeling Store, Control, and Primitives

Compiler:
The Underlying Machine



$C_{0 \to 1 \downarrow 1}$

Compiler:
Development Machine



Development Techniques and Use **of compilers are much versatile and flexible compared to those of interpreters**

$C_{0 \rightarrow 1 \downarrow 2}$

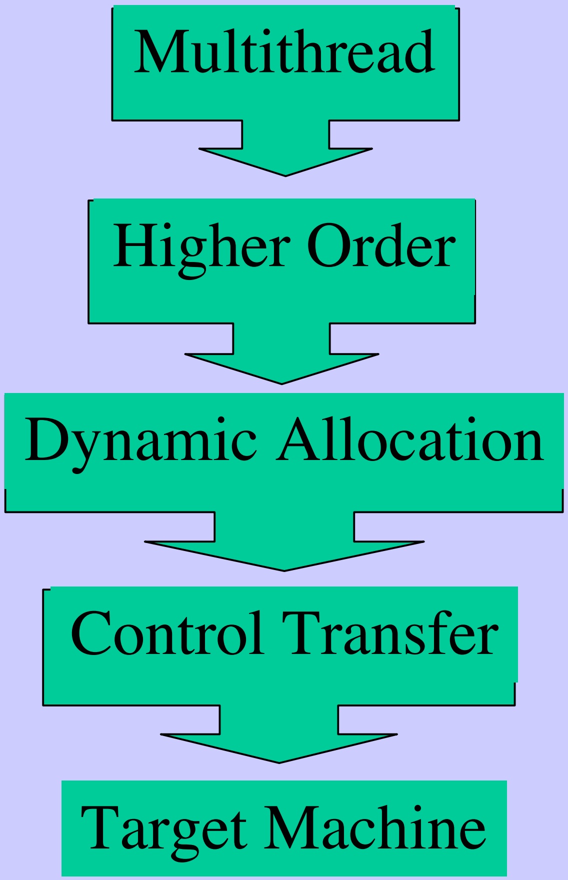
Hierachy of Development Machines

A Hierarchy of Machines

- reduces language expressiveness at each level
- simplifies the compilation of High Level language

As *higher is the expressiveness* of a language as *higher is the complexity of the interpretation cycle* of (some of) its structures and instructions
- harder is the construction of an executor of the language

Machine Hierachy: Example



When the Target Machine is a Concrete Machine

- No conceptual difference
- but Executor is effective

Classes of Macchines

Classes of Machines exist in correspondence to the different Programming Language Paradigms

- Imperative
- Functional
- Logic
- Object oriented

They differ for the supporting structures:

- store
- control
- decode (machine intepretation cycle)
- primitive data and operations

Compiler vs. Interpreter

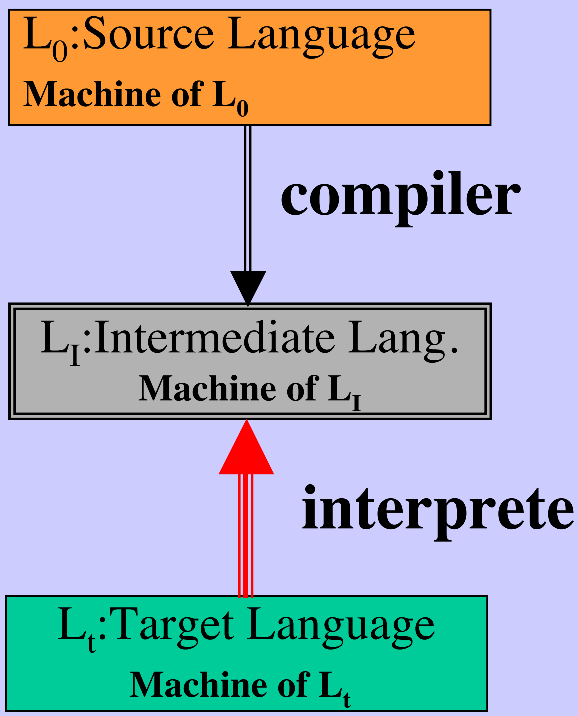
Proving *Compiler Correctness is clearer* and more evident than Proving Interpreter Correcteness

***Compiled code is more efficient* and less time consuming than Intepreted code**

***RTS is conceptually the same* to that present in Interpreters for modeling Store, Control, and Primitives**

***Development Techniques and Use of compilers* are much *versatile and flexible* compared to those of interpreters**

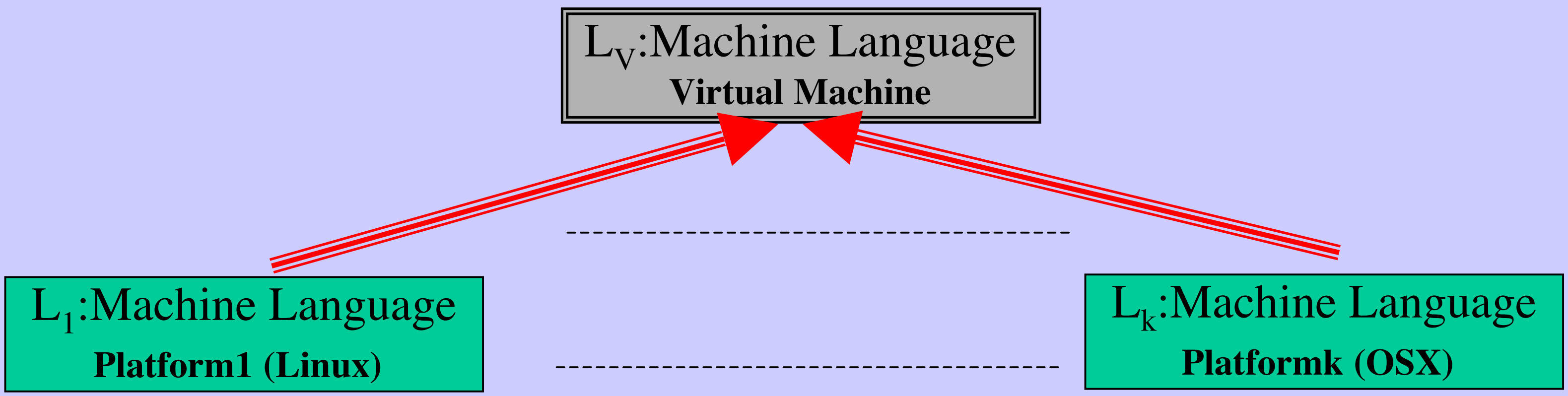
Intermediate Machine: Mixed Construction



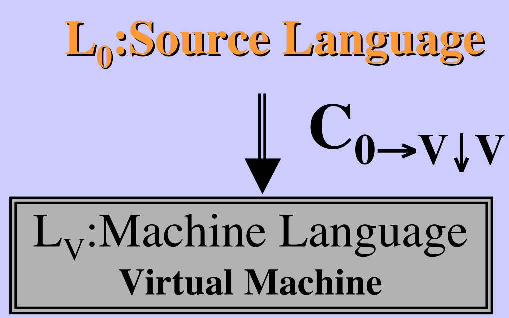
Pro:
- **contained development cost**
- **higher portability**
- **compact object code:**
  - **memory space**
  - **run time**

Virtual Machine

A *unique machine* with *many implementations*: One for each *different computer platform*



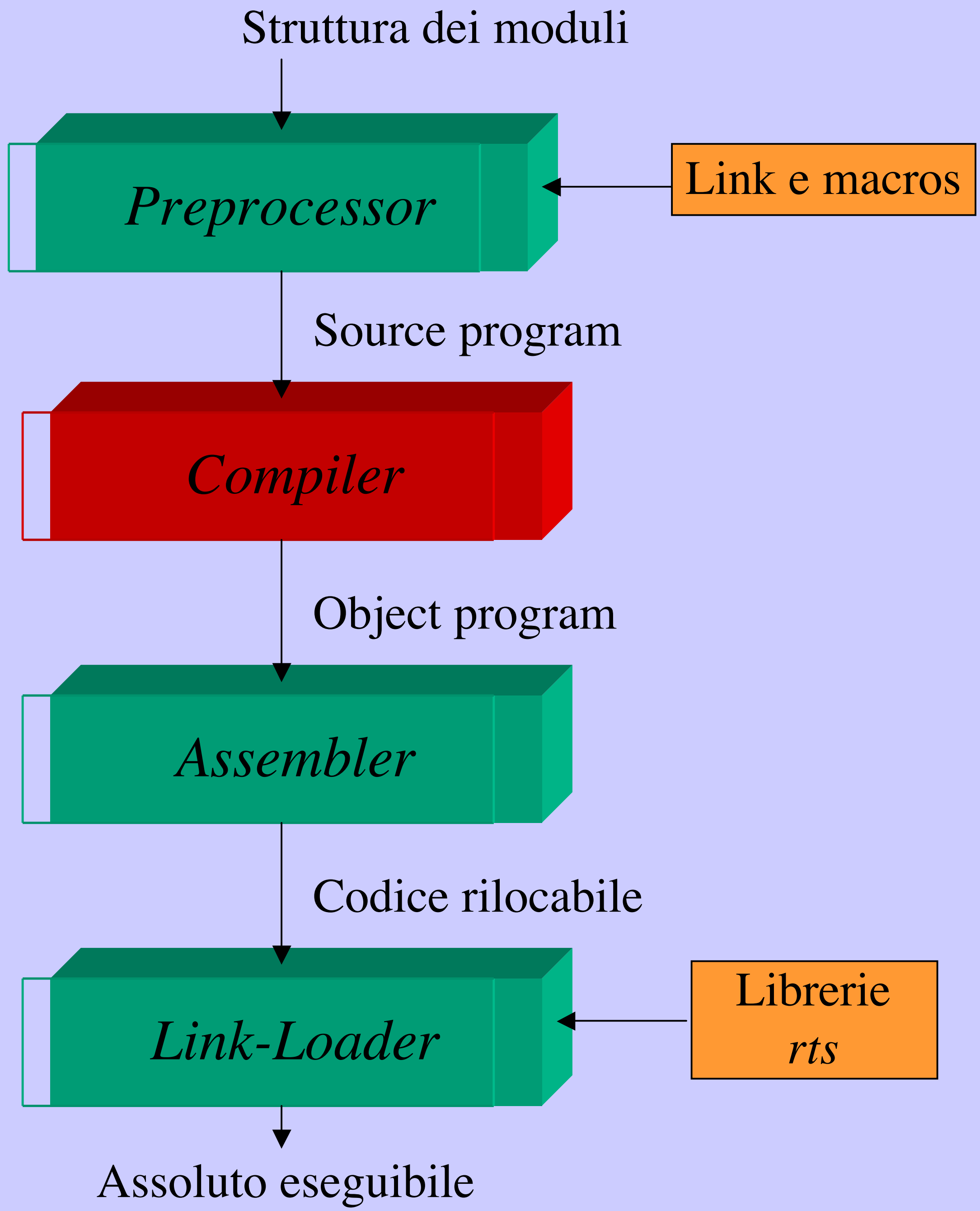
A *unique compiler* for *each Language*



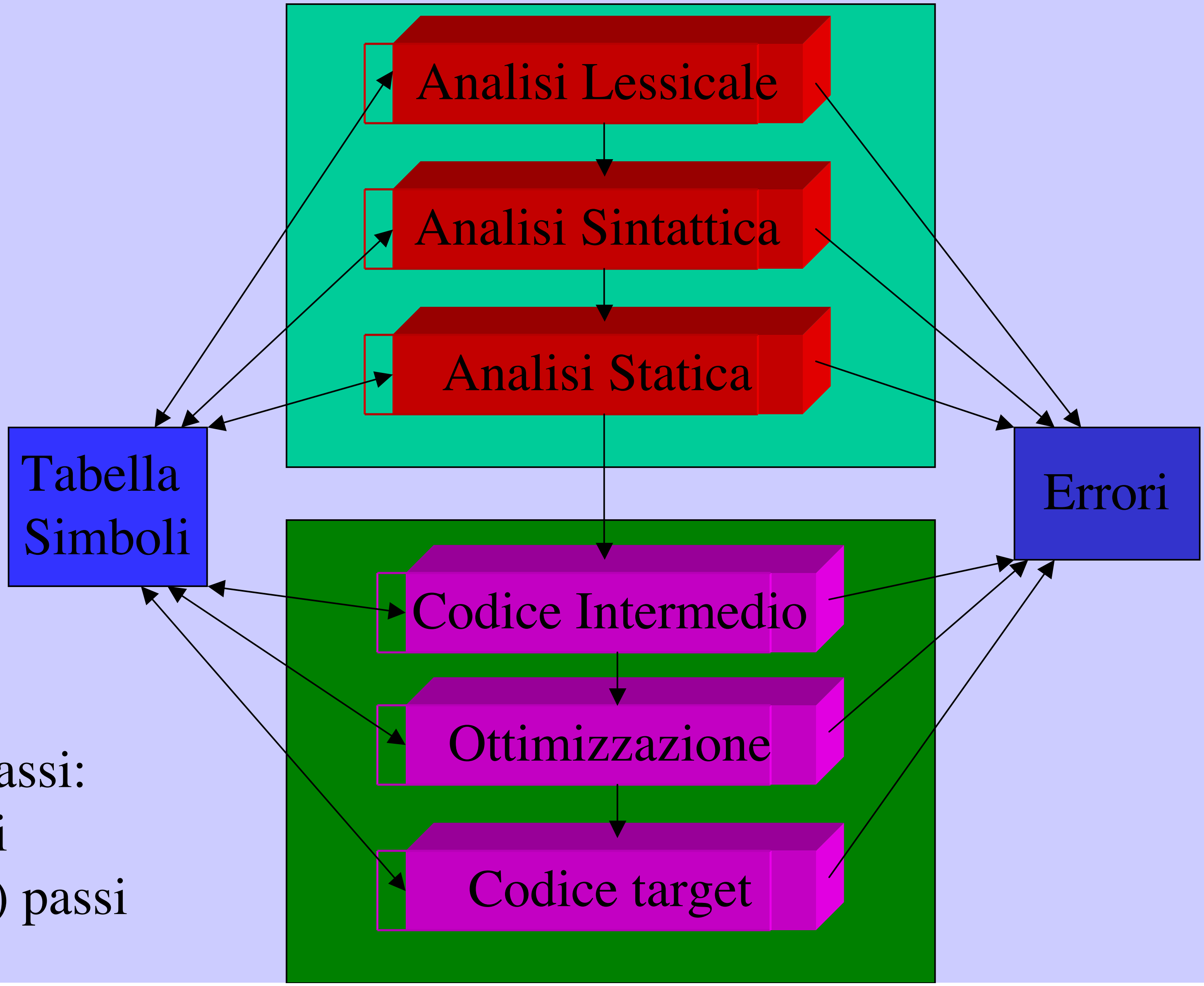
Compiler, Interpreter: contexts, structures components

- Working Context: preprocessing and loading
- Compiler: Structure, phases and steps
- Interpreter: Structure
- Compiler-Compiler: How really do it !
- Bootstrapping
- A view of the phases: Example

Contesto del Compilatore: Font-end Back-end

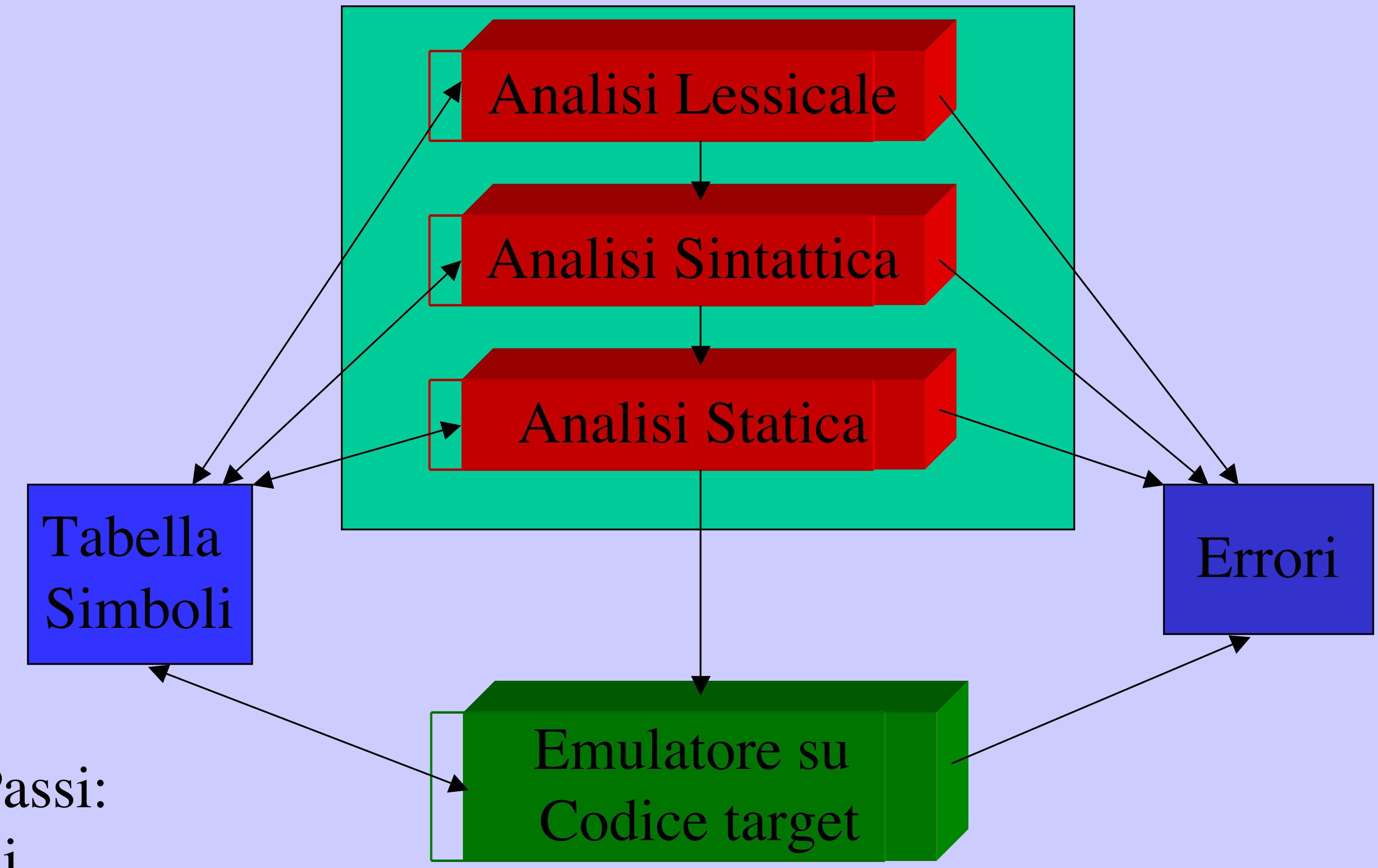


Compilatore: struttura, fasi e passi



Fasi e Passi:
- 6 fasi
- k(≥1) passi

Interprete: La struttura standard



Fasi e Passi:

- 4 fasi
- k(≥1) passi

Compiler-Compiler: How to limit metalanguage use and simplify compiler construction

The development of a compiler, from a language L_0 into a language L_t, may involve other languages, L_m, called meta-languages

Metalanguages are used to express data and procedure for analysis and translations. They affect the compiler performance which is forced to run on the chosen meta.

How much differ $C_{0\rightarrow t\downarrow m}$ and $C_{0\rightarrow t\downarrow n}$?

How to overcome this meta-language limitation?
Answer: Combining Intepreter and Compiler

Bootstrapping

- Construct an interpreter $E_{0\downarrow m}$ (It runs L_0 programs on a machinea M_m). It is a development tool

- Construct a compiler $C_{0\rightarrow t\downarrow 0}$: Noting that C is now written in source language L_0. Hence, no metas are used.

- Run: $E_{0\downarrow m}(C_{0\rightarrow t\downarrow 0})(C_{0\rightarrow t\downarrow 0})$ obtaining $C_{0\rightarrow t\downarrow t}$

$C_{0\rightarrow t\downarrow 0}$ does not use meta-languages

Compiler-Compiler & Bootstrapping: Example

Development of a Compiler for Java in 3-address PDP/11 code:

L_0=Java L_t= PDP/11 code

A time consuming, experimental, correct interpreter J_{C++} of Java is available. It is running on C++. Then, we use it:

$E_{0\downarrow m} = J_{C++}$

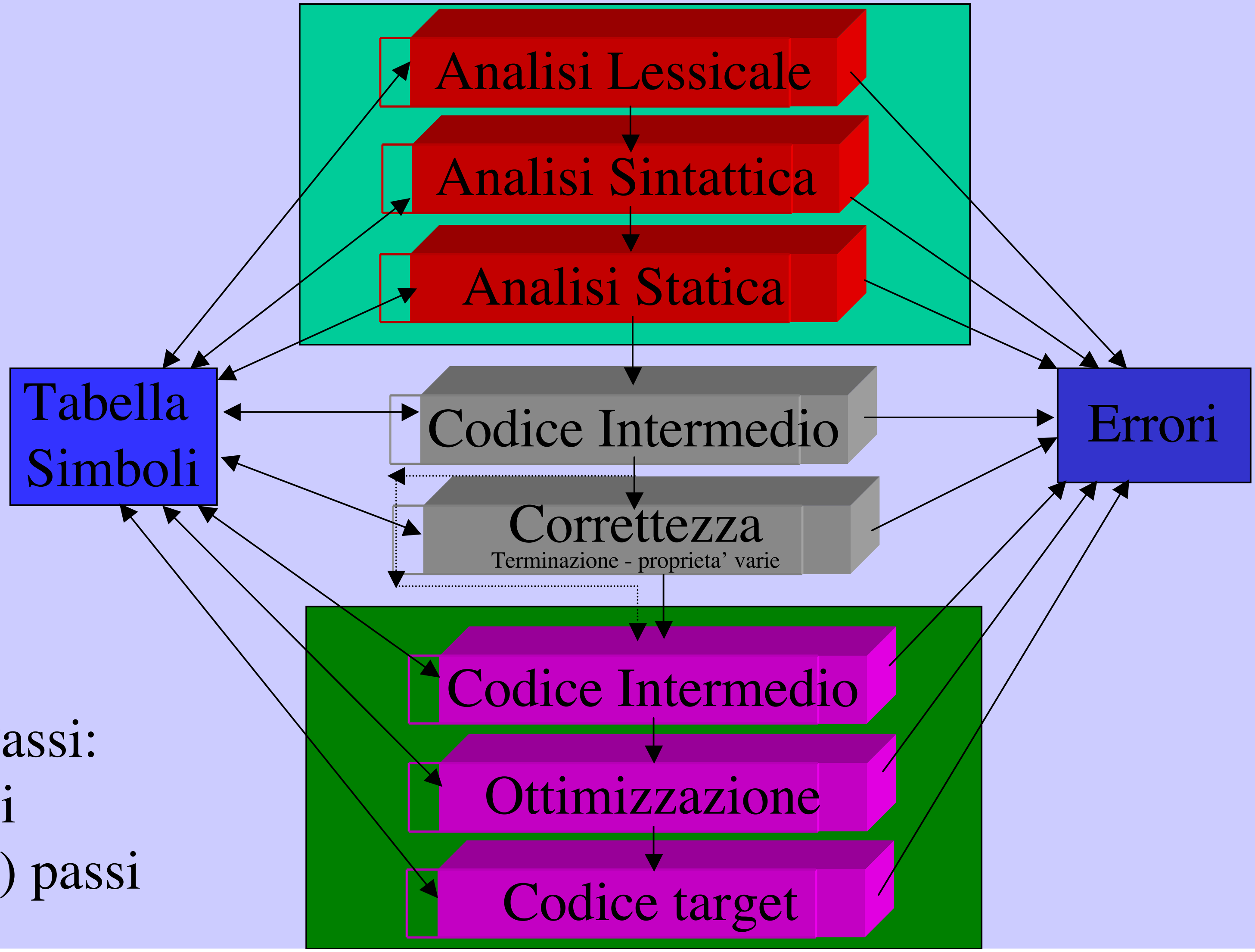
We use Java for writing down the classes and methods implementing each phase (Lexical,…,Target Code) of

$C_{Java \rightarrow PDP/11 \downarrow Java}$

A view on the Compiler phases through an Example

fig. 1.10 pag. 13 [Aho]

Compilatore: Una struttura per analisi di correttezza avanzate



Fasi e Passi:

8 fasi

k(≥1) passi