Compilers

1	**Introduction**
2	**Lexical analysis**
3	**Syntax analysis**
4	**Semantic analysis**
5	**Code generation**
6	**Runtime environments**

- Aho, Lam, Sethi, Ullman "*Compilers. Principles, Techniques, and Tools*", Addison-Wesley, 2006
- Levine, Mason, Brown "*Lex & Yacc*", O'Reilly & Associates.
- Slides : `http://www.ing.unibs.it/lamperti`

History Background

- Link
  - Programming languages
  - Implementation technology

  ⇨ abstraction process possible with compilers

- Late 1940s: first computers based on the architecture of von Neumann
  → need for writing programs = [*statements*]

- Machine code: `C7 06 0000 0002` = hexadecimal statement of Intel 8x86 for inserting number 2 at address 0000

- Assembly language: `MOV X, 2` ⇨ translated into machine code by an assembler
  - difficult to understand
  - <u>not</u> portable

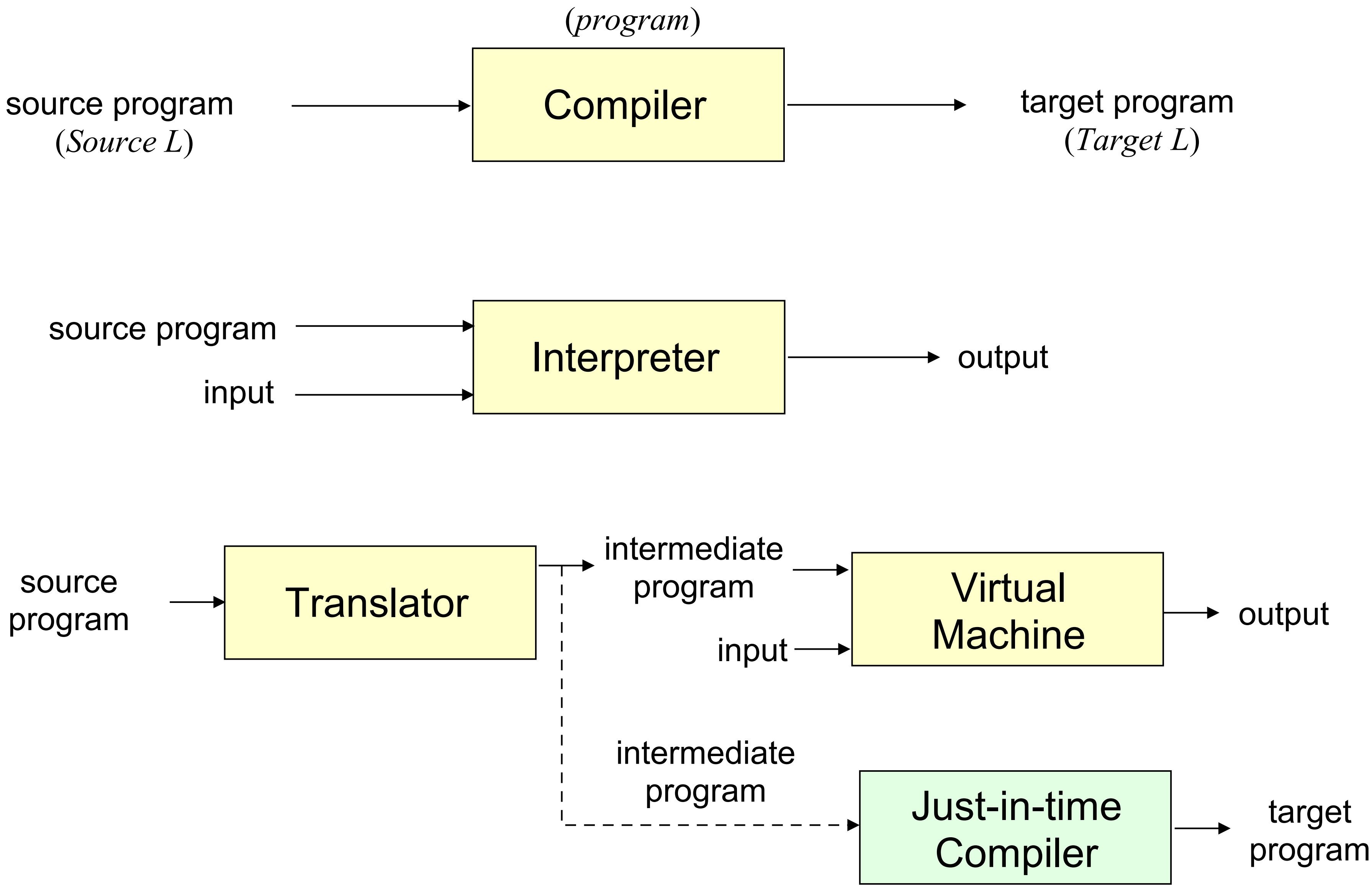
- "Ideally": specification of program operations in concise form
  - mathematics (expressions)
  - natural language (control)

  `X = 2`

- Historic objections
  - impossible
  - if possible → inefficient target code

  ⇨ contradicted by **FORTRAN**

Language Processors



- Also: source-to-source translators (e.g.: C++ $\rightarrow$ C)

Language Processors (ii)

- Apparent complexity factors in designing general techniques for compilers:
  1. Thousands of source L
  2. Thousands of target L — programming L (source-to-source) / machine L
  3. Various compiler typologies (single-pass, multiple passes, different grouping of phases ...)

- At a certain abstraction level → complexity reduced to a few essential procedures
  → make use of the same techniques

- Techniques for compilers → reusable in other contexts of computer science

- Realms of interest:
  - Artificial-language theory
  - Programming languages
  - Web
  - Hardware architectures
  - Algorithms
  - Software engineering

Implementation of High-Level Programming Languages

- High-level programming languages: easier to program but less efficient

- Low-level programming languages: more efficient but harder to write, less portable,
  more prone to errors, harder to maintain

- Optimizing compilers → techniques to improve efficiency of generated code

- Example: `register` keyword in C: considered necessary (in mid 1970s) to
  control which variables reside in registers

  → No longer necessary with effective register-allocation techniques
  Hardwiring register allocation may hurt performances!

Optimizations for Computer Architectures

- Rapid evolution of computer architecture → demand for new compiler technology

- High-performance systems take advantage of
  - *parallelism*
  - *memory hierarchies*

- Parallelism:

  - Compiler can rearrange instructions to make instruction-level parallelism by hardware more effective

  - If instruction-level parallelism in the instruction set (instructions with multiple operations in parallel) → compiler techniques to generate code for such machines from sequential programs

  - Parallelization techniques to translate sequential programs into *multiprocessor* code

- Memory hierarchies: registers + caches + physical memory + secondary storage

  - Cache-management by hw: not effective in scientific code operating on large arrays → compiler techniques for changing layout of data or order of instructions accessing data → improvement of effectiveness of memory hierarchy

Program Translations

- Translation between different kinds of languages

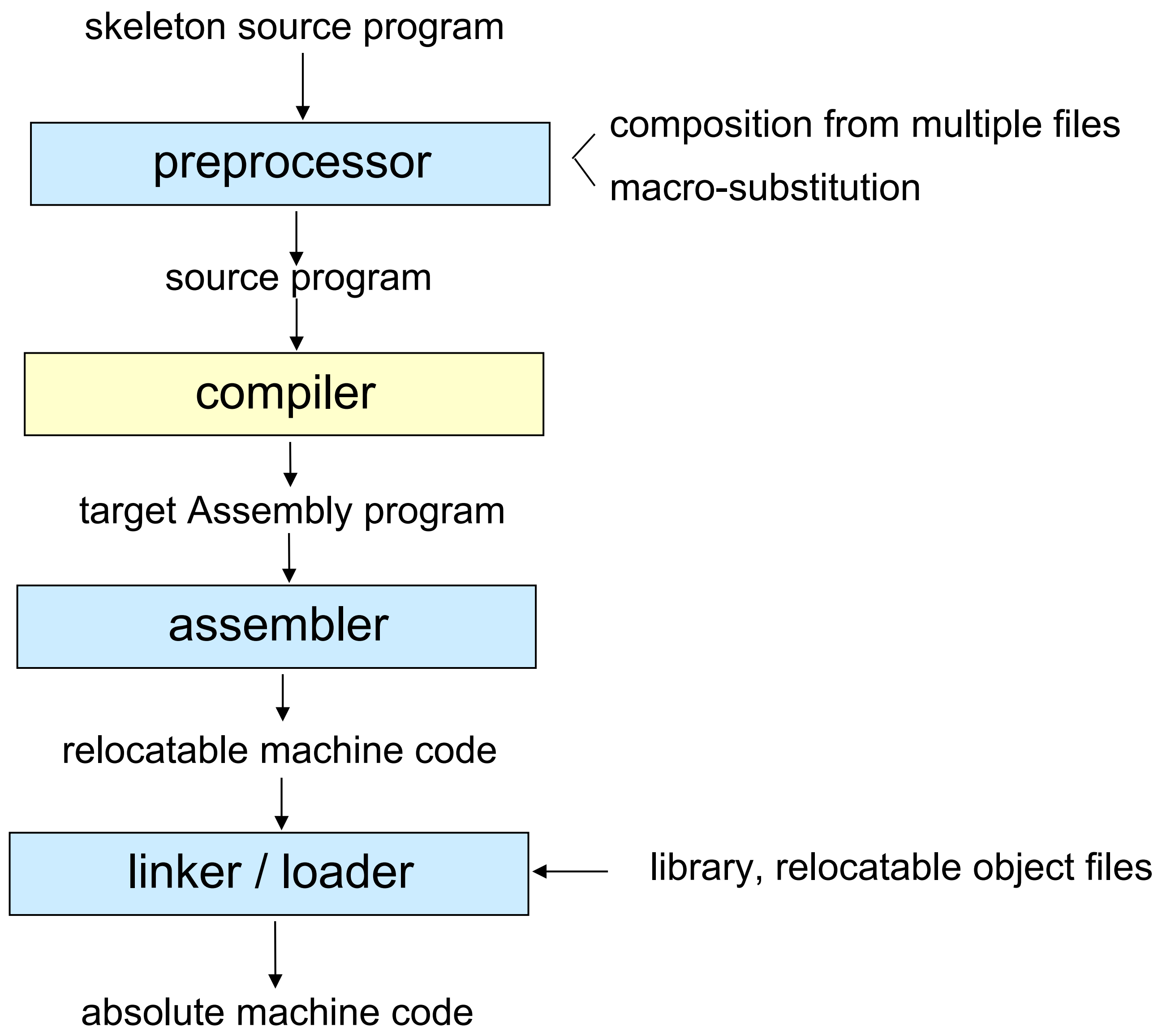
- Binary Translation: from binary code of one machine to binary code of another one (typically, to increase availability of software by computer companies)

- Hardware Synthesis: hw designs described in high-level hw description languages (VHDL, RTL) → translation by hw-synthesis tools into detailed hw schemas

- Database Query Translators: high-level query translated into actions for record search (SQL → Relational Algebra → Actions for physical search)

- Compiled Simulation: instead of writing a simulator interpreting the design (very expensive), better compiling the design into machine code that simulates the particular design

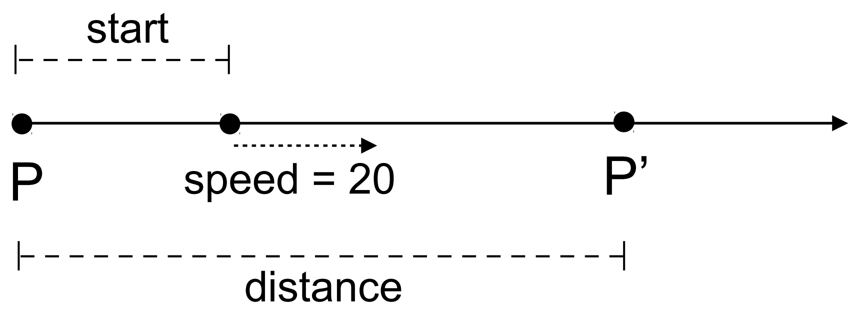
Context of a Compiler



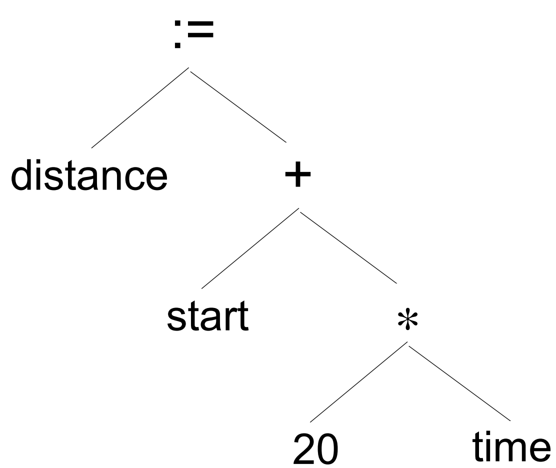
Compilation Model

- Separation
  - **analysis**: word recognition + internal representation of source (syntax tree)
  - **synthesis**: construction of target program → more specialized (and complex)

- Analysis → operations in source program mapped to hierarchical structure → **syntax tree**




```
distance := start + 20 * time
```



Analysis of Source Program

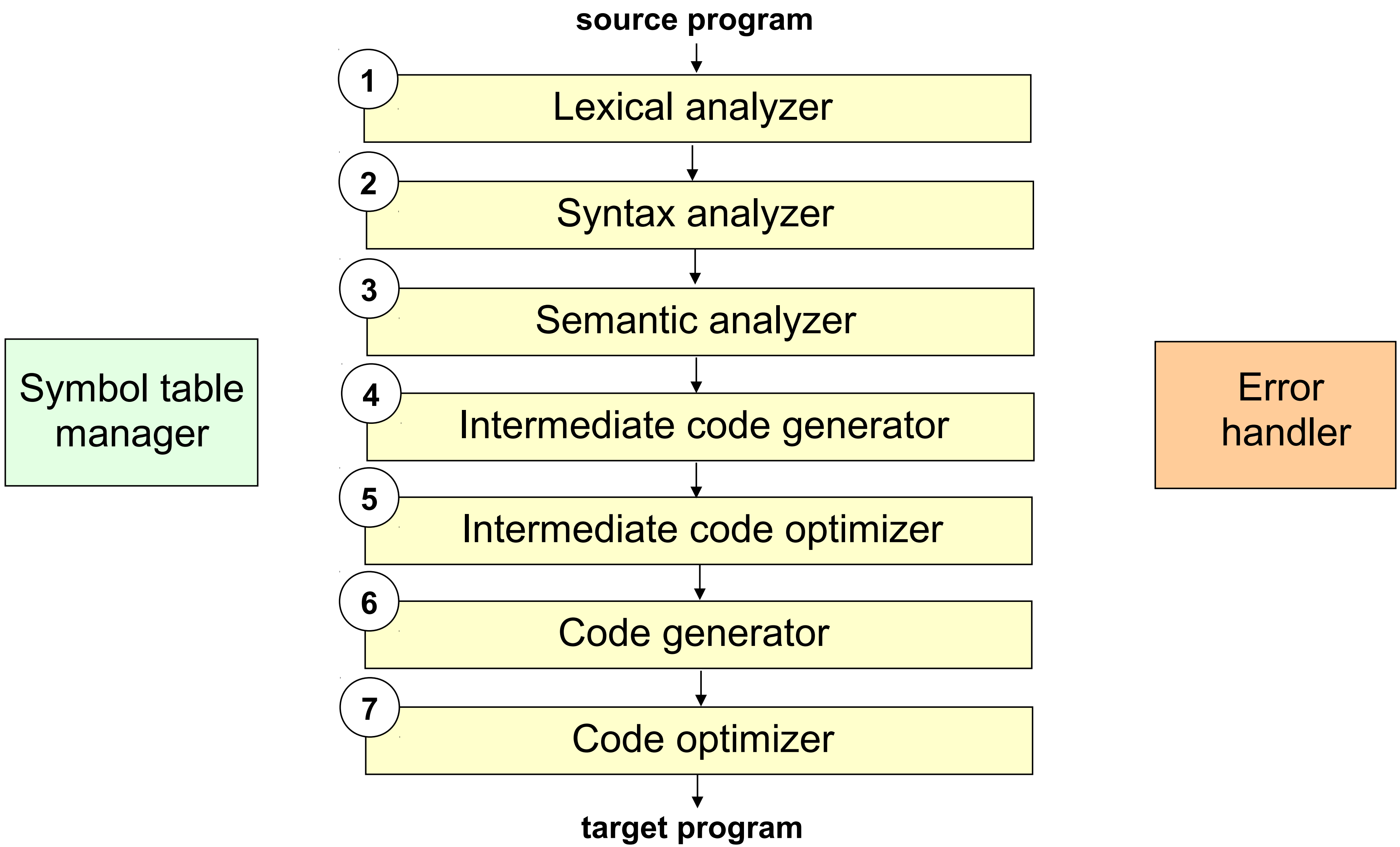
- Lexical analysis (linear) $\rightarrow$ grouping of characters into symbols $\rightarrow$ [token]

- Syntax analysis (hierarchical) $\rightarrow$ grammar symbols grouped hierarchically $\rightarrow$ syntax tree

- Semantic analysis $\rightarrow$ consistency checking
  - type checking
  - insertion of information into symbol table

Phases of a Compiler

- Phase = conceptual unit in which a compiler operates: Rep(source P) $\rightarrow$ Rep'(source P)



Symbol Table

- Data structure containing information on <u>identifiers</u> (catalog)

attributes (relevant information)

id					
distance					
start					
time					

- allocated space
- type
- scope
- if procedure name
  - number of parameters
  - type of parameters
  - method of passing param.
  - [type of output parameter]

- Requirements
  - efficient access to identifier attributes
    - read
    - write
  - <u>incremental</u> update of attributes

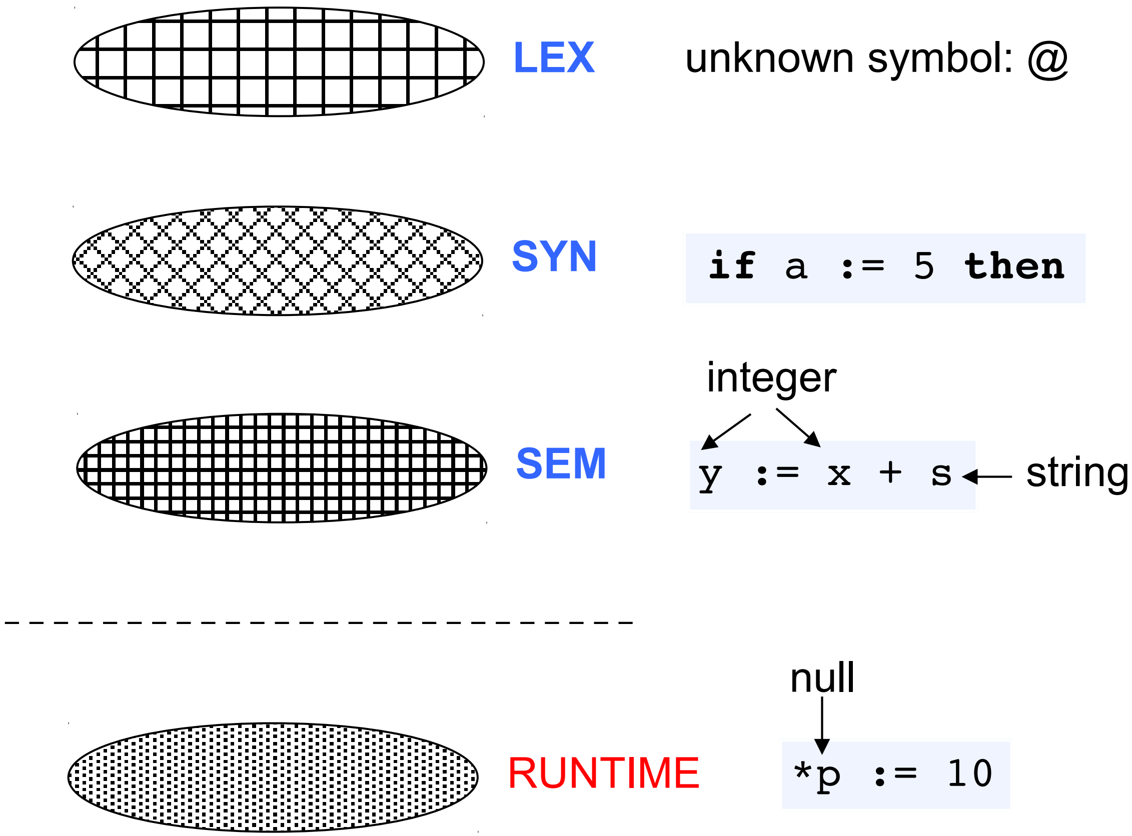
```
var distance, start, time: real;
```

Lexer → inserts identifiers, but type known only later (sem)

- Example of use:
  - SEM: type checking
  - GEN: space allocated in memory

Error Handling

- ∀ phase F → handling of errors <u>pertinent</u> to F (separation of concerns)



Transformation of Source Program

- Same computation expressed by different abstraction levels

```
distance := start + 20 * time
```

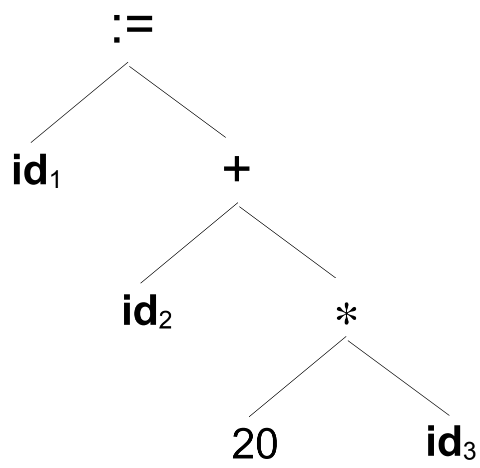
1. Lexical analysis   **id**$_1$:= **id**$_2$ + 20 * **id**$_3$

   - Recognition of (grammar) terminals → pairs (symbol, value)
   - Removal of spacing / comments
   - Encoding of each symbol (**:=** → `#define ASSIGN 257`)
   - Some symbols enriched by a lexical value (**id** → "`distance`", or pointer to symbol table)

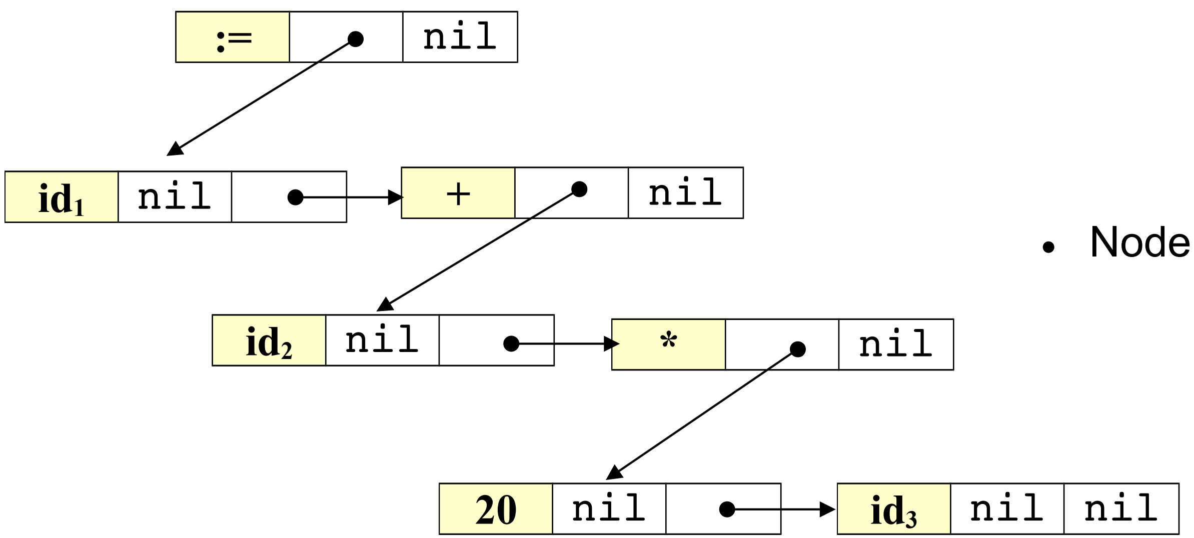
Transformation of Source Program (ii)

2. Syntax analysis

id$_1$:= **id**$_2$ + 20 * **id**$_3$



(token, child, brother)



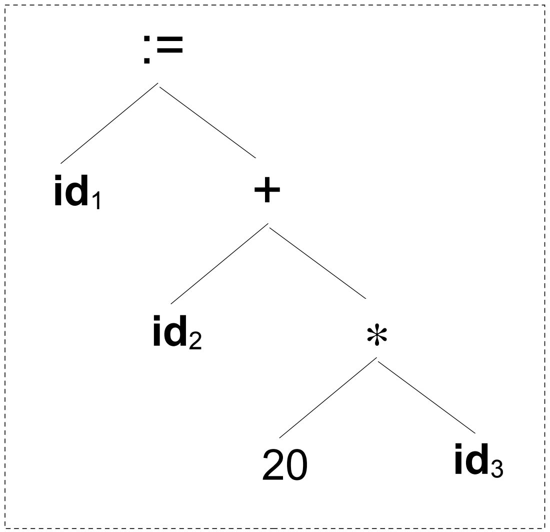
union

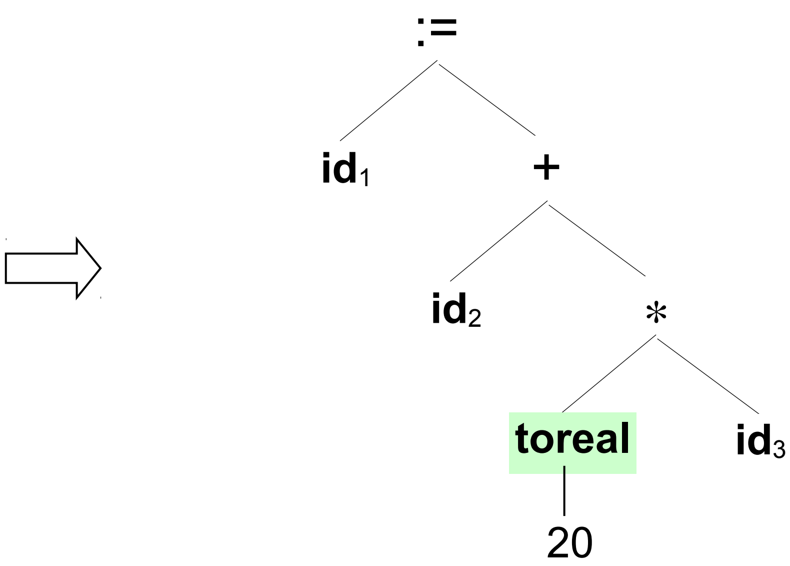
- Node uniformity → pb
  - inhomogeneity
  - child cardinality

binarization

Transformation of Source Program (iii)

3. Semantic analysis

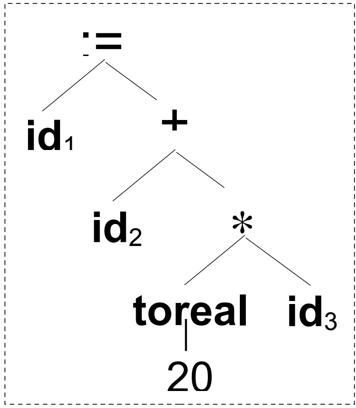




- Review of the tree for checking semantic constraints (types consistent with operations)
- Possible decoration/alteration of the tree

Transformation of Source Program (iv)

4. Intermediate code generation $\rightarrow \exists$ one statement for each operator of the syntax tree



t_1 := toreal(20)
t_2 := t_1 * id_3
t_3 := id_2 + t_2
id_1 := t_3

$\approx$ operational semantics

- Intermediate code = program written in a language of an abstract (virtual) machine
- Properties: easy to
  - generate
  - translate into target code
- Advantages
  - implementation simplification
  - portability (reusability)
- Nature: varying, typically: three-address code (quadruples) $\approx$ Assembly where memory locations viewed as registers

operator $addr_1$ $addr_2$ $addr_3$ $\Rightarrow$ * $addr_1$ $addr_2$ $addr_3$ = $addr_3$:= $addr_1$ * $addr_2$

- $\exists$ at most one <u>explicit</u> operator (in addition to assignment) $\rightarrow$ operation linearization
- Generation of temporaries for intermediate results
- Not necessarily three operands (possibly less, e.g. **toreal**)

Transformation of Source Program (v)

5. Intermediate code optimization $\rightarrow$ reduction of number of instructions

t_1 := toreal(20)
t_2 := t_1 * id_3
t_3 := id_2 + t_2
id_1 := t_3

$\Rightarrow$

t := 20.0 * id_3
id_1 := id_2 + t

- Intermediate code more efficient (e.g.: conversion 20 $\rightarrow$ 20.0 can be performed statically!)
- Problem: slowdown of compiling (for advanced optimizations)
- Possible different optimization levels
- Better not to optimize during coding (time, debugging)
- After optimization $\rightarrow$ redo testing

Transformation of Source Program (vi)

6. Code generation $\rightarrow$ need for register-loading to perform operations

$t := 20.0 * id_3$	
$id_1 := id_2 + t$	

$\Longrightarrow$

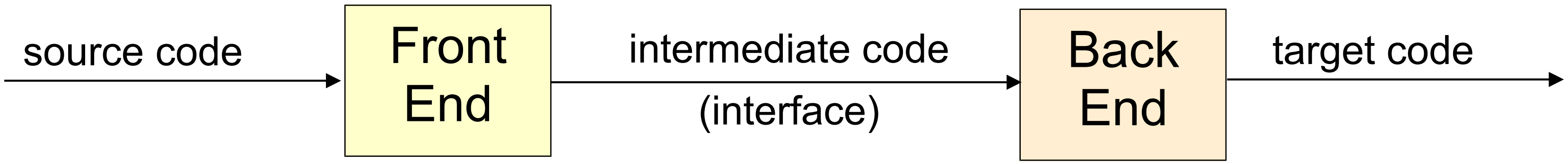
LDF	R_2,	id_3	
MULF	R_2,	R_2,	#20.0
LDF	R_1,	id_2	
ADDF	R_1,	R_1,	R_2
STF	id_1,	R_1	

- Type of target code
  - Assembly (typically)
  - Machine, relocatable $\rightarrow$ variables transformed into memory locations

- In general: mapping
  - statements $\rightarrow$ equivalent machine statements
  - variables $\rightarrow$ registers (for operations)

Grouping of Compiler Phases

- Logical organization ≠ physical organization (like in DB)



- Front-End: depends only on source → LEX, SYN, SEM, GEN-I [, part of OPT-I]
- Back-End: depends only on target → OPT-I, GEN, OPT

- Macro-modularization: useful for the porting of compiler (reusable macro-modules)

⇓

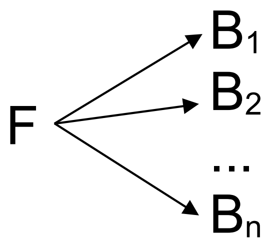
Ideally: changing
- Source L → change of Front-End
- Target L → change of Back-End

⇐ intermediate code: invariant

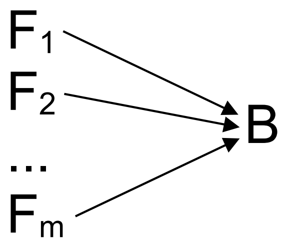
Grouping of Compiler Phases (ii)

- Possible scenarios:

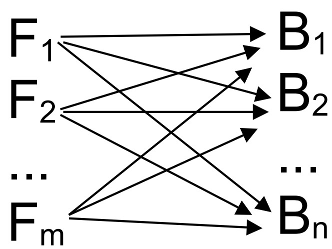
1. Same L on ≠ platforms (realistic):



2. ≠ L on same platform (ideal):



3. ≠ L on ≠ platforms:



⇐ (n+m) modules rather than (m∗n) compilers

Bootstrapping & Porting

- L involved in compiler construction
  - source
  - implementation (host)
  - target

Machine language in first compilers

(bootstrapping)

Compiler of language α, written in β → Compiler of β → Compiler (executable) of α

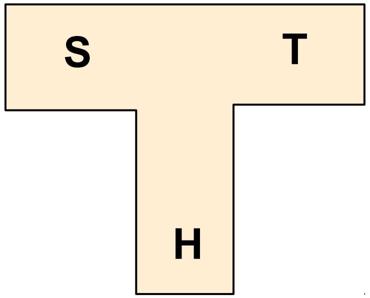
α = Ada
β = C

- **Cross-compiler**: when Comp(α) runs on a machine $\neq$ target
  (necessary when target machine with limited resources)
  $\rightarrow$ target code not executable on the same machine

target = Intel
Comp(α) on AMD

Bootstrapping & Porting (ii)

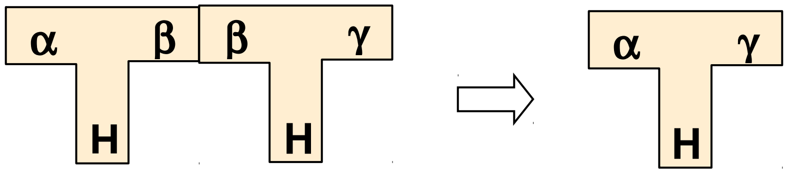
- T-diagram: to schematize a compiler (by means of the 3 involved languages)



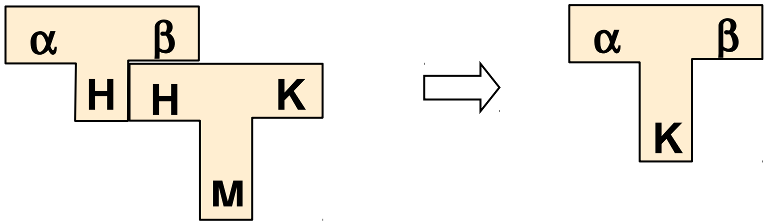
- **H** = Machine language of executable
- **H** ≠ **T** → cross-compiler

- Diagram composition (to generate new compilers):

1. **Concatenation**
   (*Change of **target** language*)



2. **Joint**
   (*Change of **host** language*)

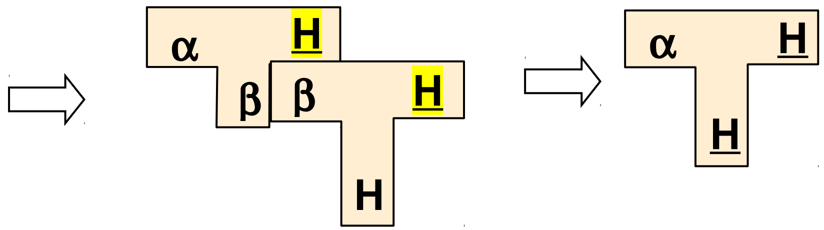


α = Ada
β = C
H = Pascal
K = Fortran

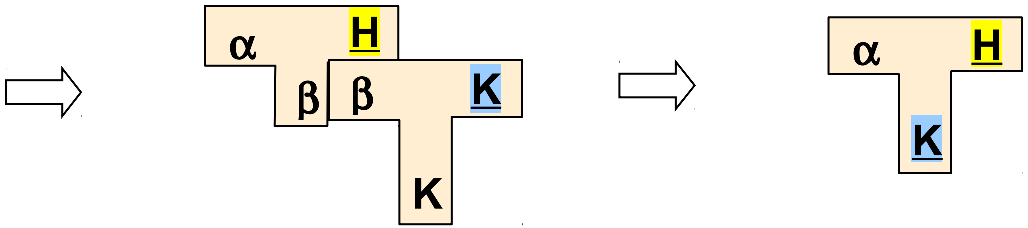
Bootstrapping & Porting (iii)

Notes (3 kinds of bootstrapping):

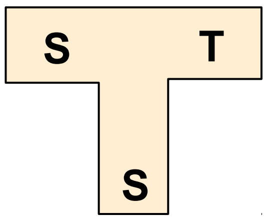
- Compiler of language α, written in β → (bootstrapping) Compiler of β → Compiler (executable) of α



- Bootstrapping of cross-compiler



- **Compiler written in the source language to compile** ! → Pb of **bootstrapping** !



Bootstrapping & Porting (iv)

- Solution of the circularity:
  - **GOOD** = compiler of S written in S
    - efficient
    - generates efficient target code
  - <u>**BAD**</u> = compiler of S’ $\subset$ S written in Assembly

- Limits of <u>BAD</u>:
  - Compiles only a subset of S (relevant to implementation of GOOD)
  - Inefficient at runtime
  - Generates inefficient target code
    - space
    - time

- Unique requirement for <u>BAD</u>: *functional correctness of the generated code*

GOOD
+
<u>BAD</u>

<u>GOOD</u>

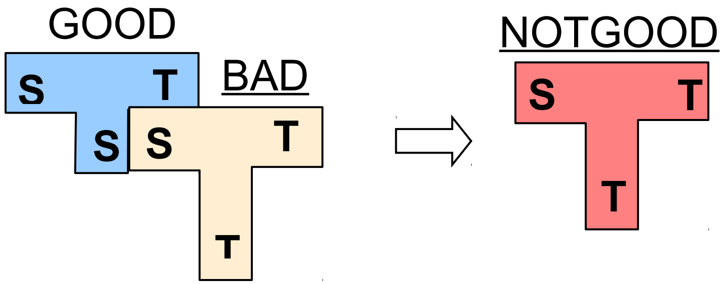
Bootstrapping & Porting (v)

Bootstrapping (alchemy):

1. Writing in Assembly of a compiler "*quick & dirty*" (BAD) for a subset of S used to write GOOD

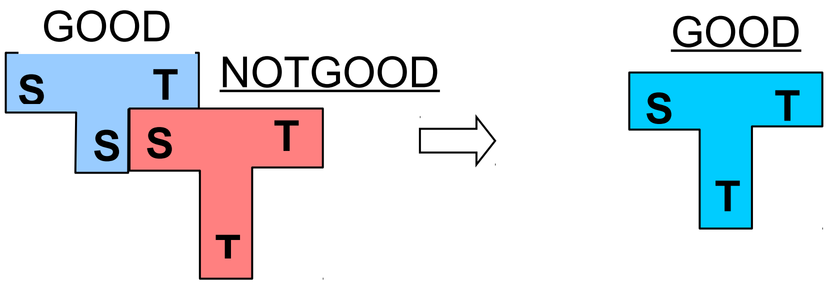
2. Compiling of GOOD using BAD → NOTGOOD

NOTGOOD:
- inefficient (runtime)
- compiles every program written in S
- generates efficient target code!



Requirement of functional correctness for the code generated by BAD

3. Compiling of GOOD using NOTGOOD → GOOD



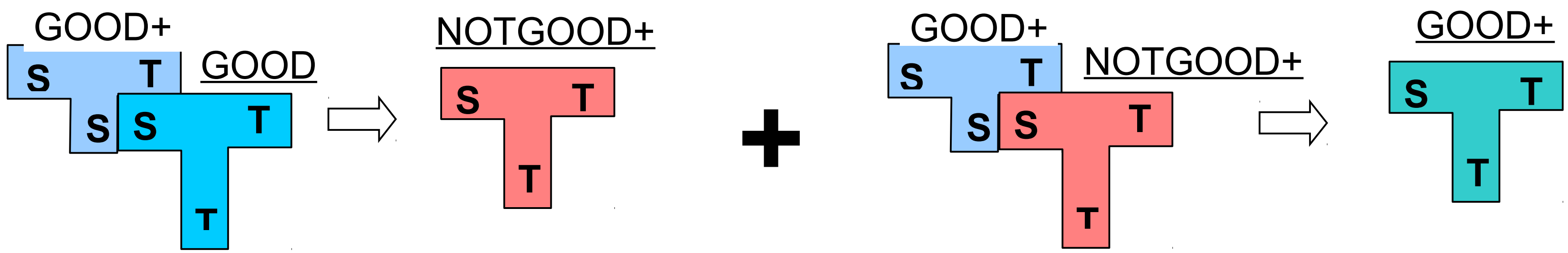
GOOD:
- efficient (runtime)
- compiles every program written in S
- generates efficient target code

Bootstrapping & Porting (vi)

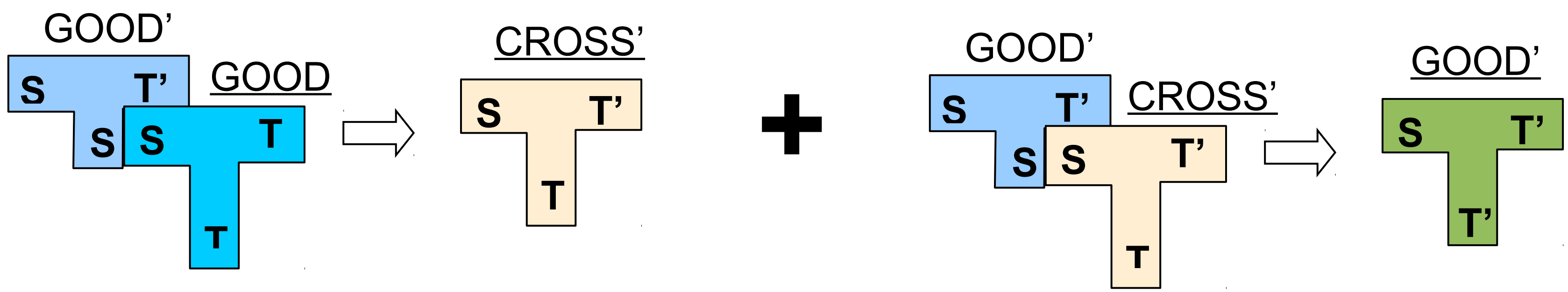
- After bootstrapping → compiler specified in 2 forms
  - source S = GOOD
  - target T = <u>GOOD</u>

<u>Advantages</u>:

a) Improvement of GOOD → GOOD+ → immediately bootstrapped by steps 2 and 3 in <u>GOOD+</u>



b) Porting of GOOD to a new host computer → only rewritten the Back-End of GOOD → GOOD’



Bootstrapping & Porting (vii)

- Similar considerations when “*quick & dirty*” written in a high-level language ≠ S