LANGUAGE PROCESSORS

UNIT 1: INTRODUCTION

uc3m

David Griol Barres
dgriol@inf.uc3m.es
Computer Science Department
Carlos III University of Madrid
Leganés (Spain)

OUTLINE

- Aims of the course
- Why to take this course?
- Related topics
- What are compilers?
- Compilers vs. Translators
- Compilers vs. Interpreters
- Related programs
- Schema of a compiler
- Evolution
- Compilers classification

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

OUTLINE

- Phases of a compiler
  - Introduction: Front-end and Back-end
  - Lexical Analysis
  - Syntax Analysis
  - Semantic analysis
  - Symbol table
  - Intermediate Code Generation
  - Code Optimization
  - Code Generation
  - Error Handling

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

OUTLINE

- Programming language design
- T-notation and Bootstrapping
- Programming paradigms
- Bibliography

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Aims

- This module/course is designed to introduce the student to the principles and practices of programming language implementation.

- We cover **lexical analysis, parsing theory, semantic analysis**, runtime environments, code generation, and optimization.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Aims

- At the end of the course you should know how to:
  - Write a grammar for a given language.
  - Verify that a grammar fulfills some properties.
  - Modify a grammar so it fulfills some properties.
  - Write a parser for a given language.
  - Verify the semantics of a given language.
  - Generate intermediate/final code.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Why to take this course?

- Understand compilers/languages:
  - Understand the code structure.
  - Understand the language semantics.
  - Understand the relation between source code and generated machine code.
  - Become a better programmer.

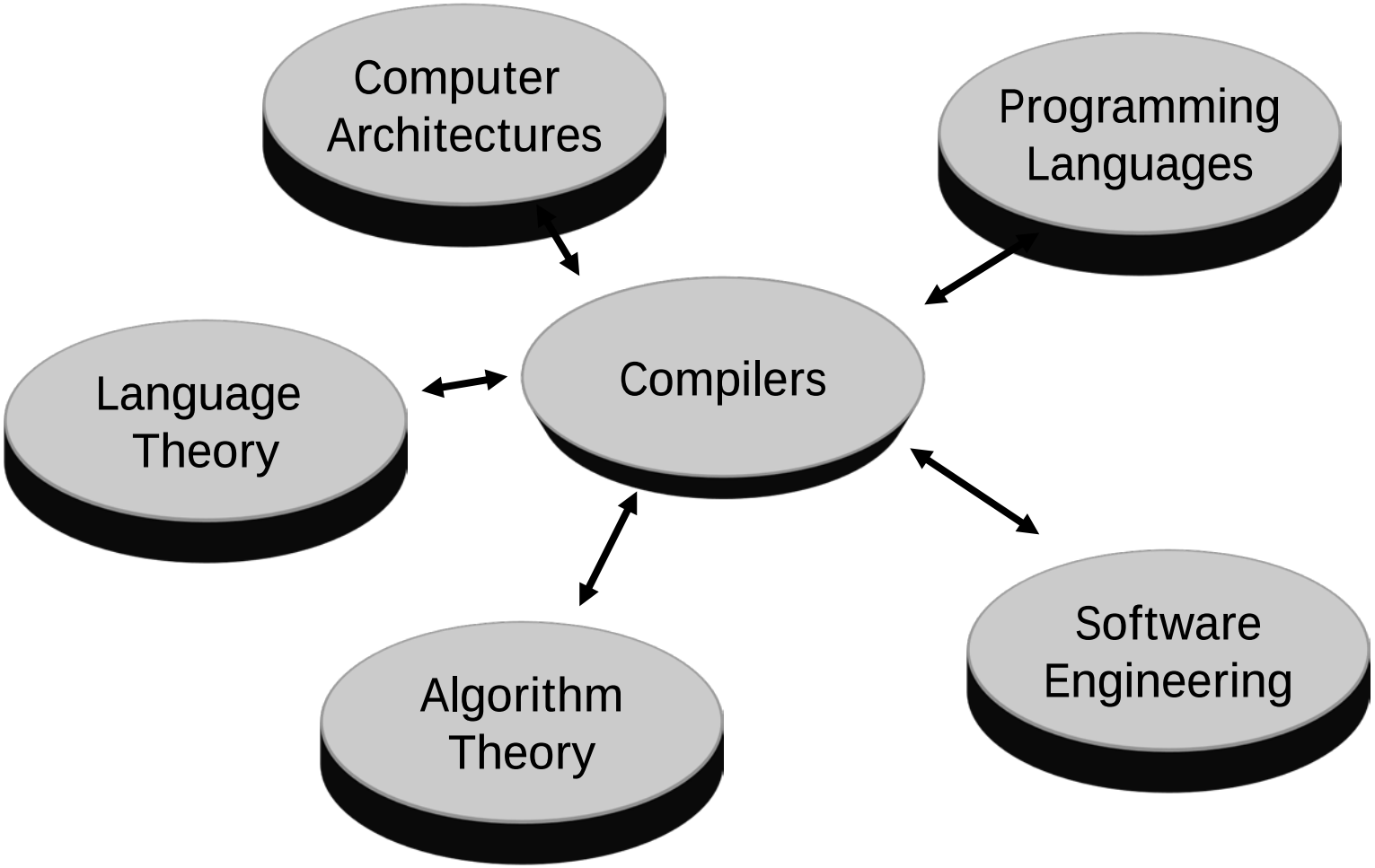
David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Why to take this course?

- Theory is essential:
  - Before the application of automata theory and formal languages, programming, etc. compilers were pretty bad.
- Compiler theory and tools are applicable to other fields:
  - Command and query interpreters;
  - Text formatters (TeX, LaTeX, HTML);
  - Graphic interpreters (PS, GIF, JPEG);
  - Translating javadoc comments to HTML;
  - Generating a table from the results of a SQL query;
  - Spam filter;
  - Server that responds to a network protocol;
  - …

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

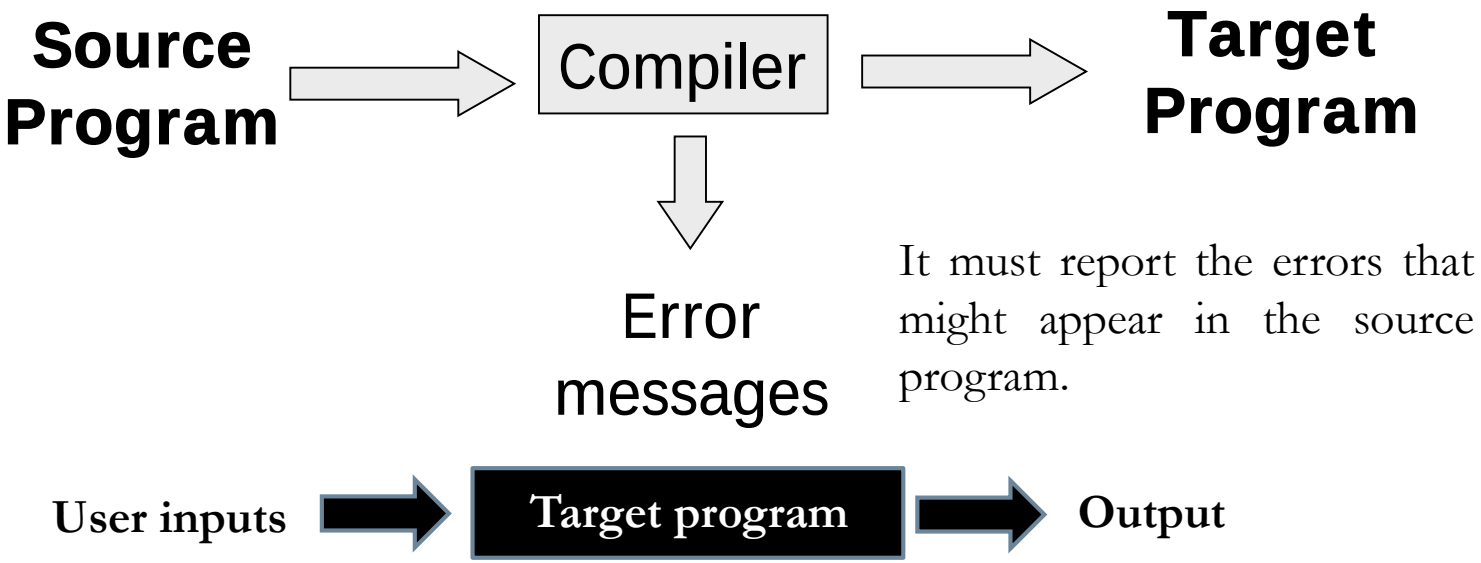
Related topics



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

What are compilers?

- A program that reads a program written in one high-level language and translates it into an equivalent program in another (object) language, which is ready to be executed on a computer.



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Compilers vs. Translators

- Compilers typically refer to the translation from high-level source code to low-level code.

- Translators refer to the transformation at the same level of abstraction.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

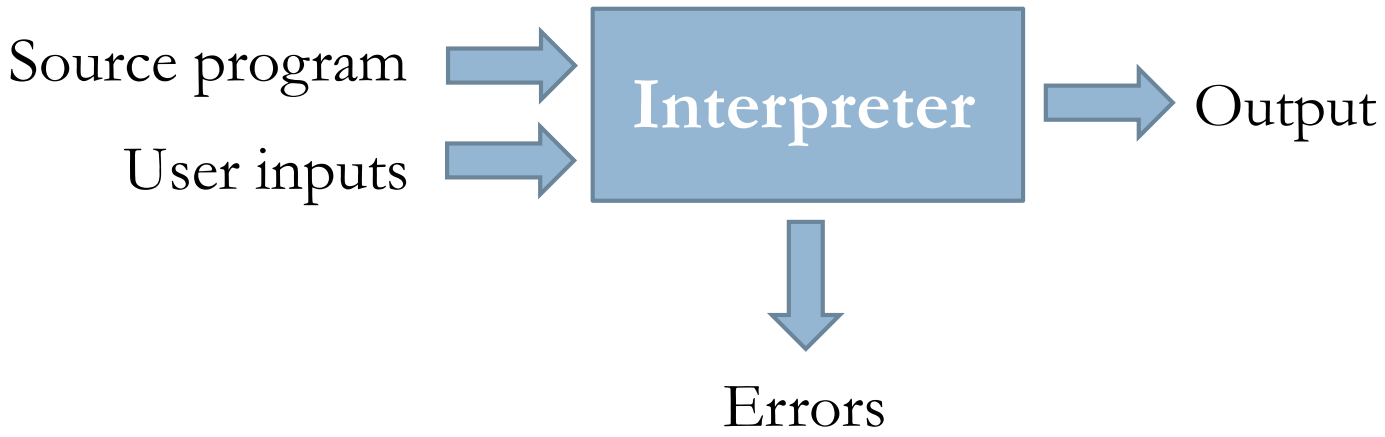
Examples

- Typical compilers: gcc, javac…
- Non-typical compilers:
  - Latex (document compiler).
  - C-to-silicon compiler.
- Translators
  - F2c: Fortran-to-C translator (both high-level).
  - Latex2html (both documents).
  - Dvips2ps (both low-level).

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Compilers vs. Interpreters

- Interpreter:
  - It directly executes the source program on inputs supplied by the user.



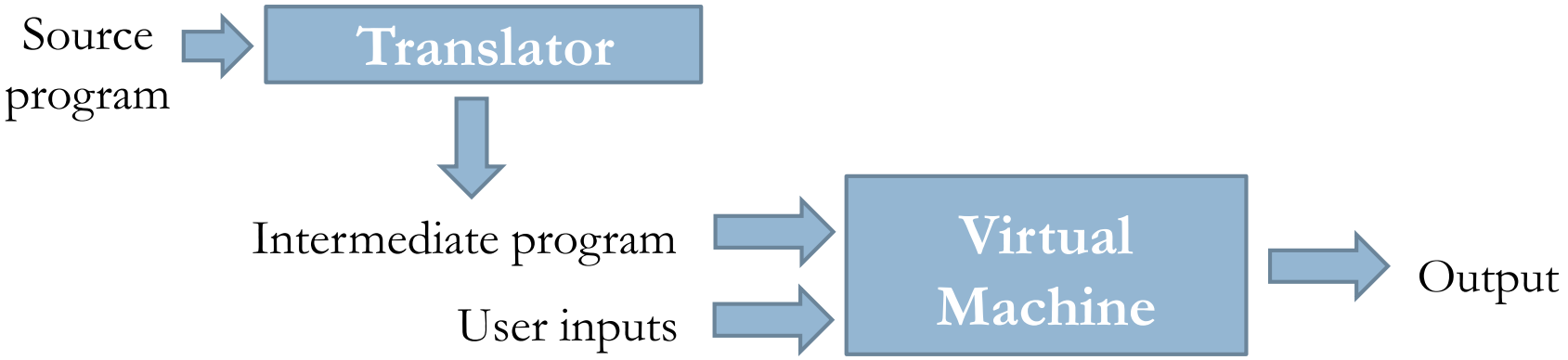
David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Compilers vs. Interpreters

- Comparison Compiler – Interpreter:

Interpreter	Compiler
It can usually give better error diagnostics. It usually can use more sophisticated functions and operators.	The target program generated maps faster input to outputs.

- **Hybrid compiler** (combine compilation and interpretation):



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Related programs

- **Assembler**: Program that translates an assembly-language program into a relocatable machine code.
- Large programs are often compiled in pieces:
  - **Preprocessor**: Previous program used to eliminate comments, include other files…
  - **Linker**: Collects the source program when it is divided into separate files and resolves external memory addresses.
  - **Loader**: Puts together all of the executable object files into memory for execution.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

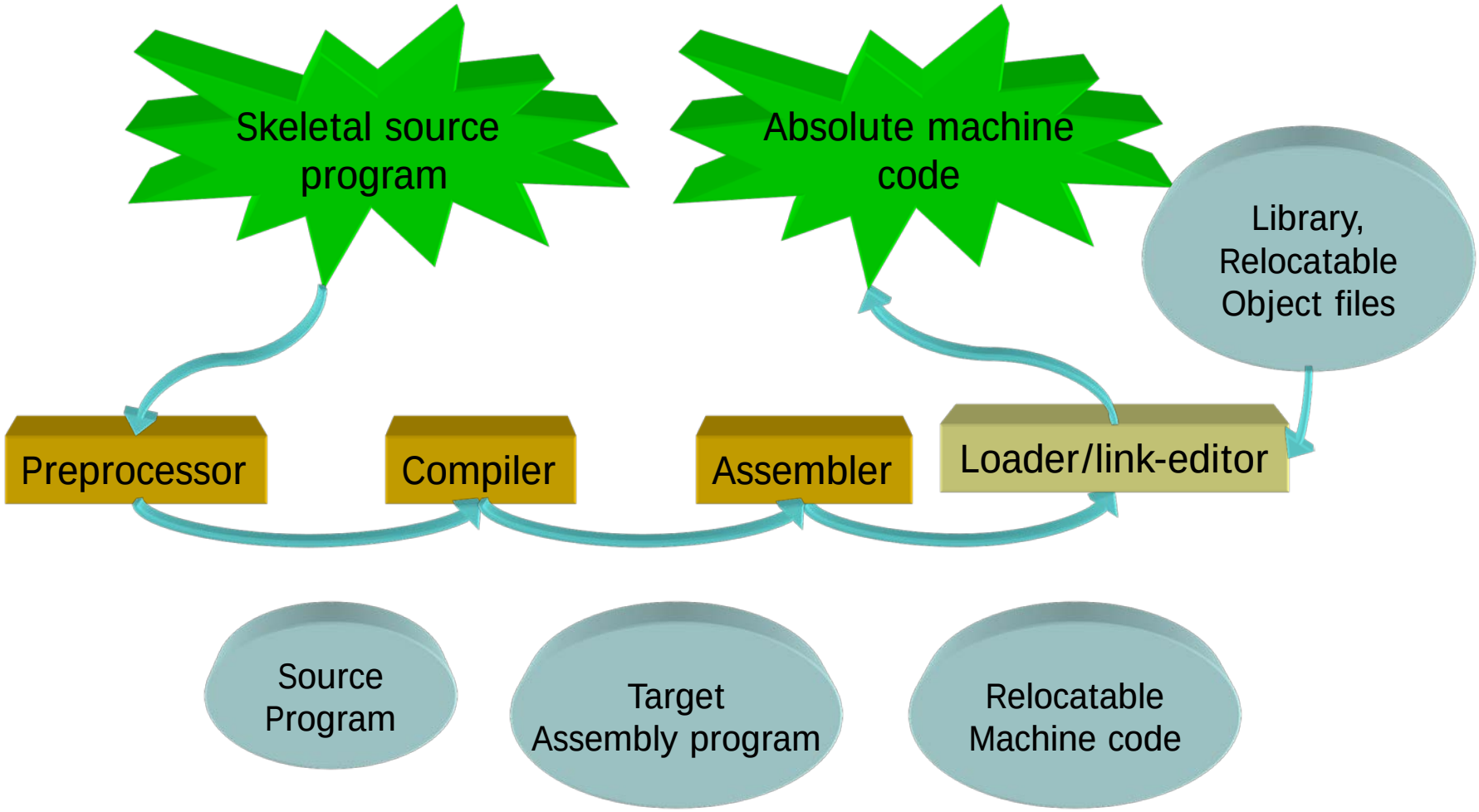
Related programs

- **Outliner**: Collects statistics about the execution of the program (calls and time in each procedure…).
- **Debugger**: Determines the errors during the execution of a compiled program.
- **Project administrator**: Coordination of the files that are modified by several programmers (sccs, rcs).
- **Editor**: Includes the call to the compiler.

- **Decompiler**: Program that translates from a low level language to a higher level one.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Compiler, schema



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Evolution

- In the 50's compilers were considered difficult programs to write.
- The first FORTRAN compiler took 18 staff-years to implement.
- Since then, systematic techniques, programming environments and software tools have been developed to facilitate the task of building a compiler.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Evolution

- First compilers translated arithmetic formulas into machine code.
  - Eg. FORTRAN stands for FORmula TRANslator.

- Nowadays:
  - A wide variety of source languages.
  - A wide variety of object languages, either another high-level language or machine language.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Evolution

- **40's**: First computers (John von Neumann) → Programs written in machine code.

C7 06 0000 0002 (Intel 8x86)

- **Second step**: Assembly language.

MOV X, 2

  - It is not easy to read, write and understand.
  - It depends on the machine for wich it has been generated.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Evolution

- **Write programs in a natural language:**

$$X = 2$$

  - FORTRAM (John Backus).
  - Natural language structure → Chomsky hierarchy:
    - Type 0 grammars: Unrestricted grammars.
    - Type 1 grammars: Context-sensitive grammars.
    - Type 2 grammars: **Context-free grammars**.
    - Type 3 grammars: Regular grammars.
- 60's and 70's: Syntax analysis (type 2 grammars).
- Regular expressions and Finite Automata (Type 3) → Lexical analysis (structure of the words in a language).

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Evolution

- Techniques for code optimization.

- Programs to automatically perform:
  - Syntax analysis → Yacc.
  - Lexical analysis → Lex.

- Interactive development environment (IDE): programs window-based that include the different parts of a compiler.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Compilers classification

- There are several classifications and types of compilers:
  - assemblers, cross compilers, single-pass compilers, multi-pass compilers, incremental compilers, self-hosting compilers, metacompilers, decompilers…

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

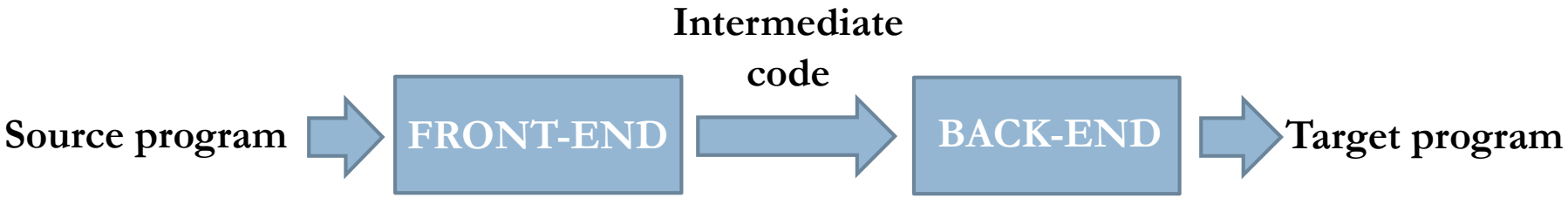
Compilers classification

- Compiling evolves in two phases, namely the analysis and synthesis phases.
  - During the analysis phase the source program is read, fragmented into pieces and transformed into a representation that is suitable for the phase of synthesis.
  - The synthesis phase constructs the target program from the representation provided by the analysis phase.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

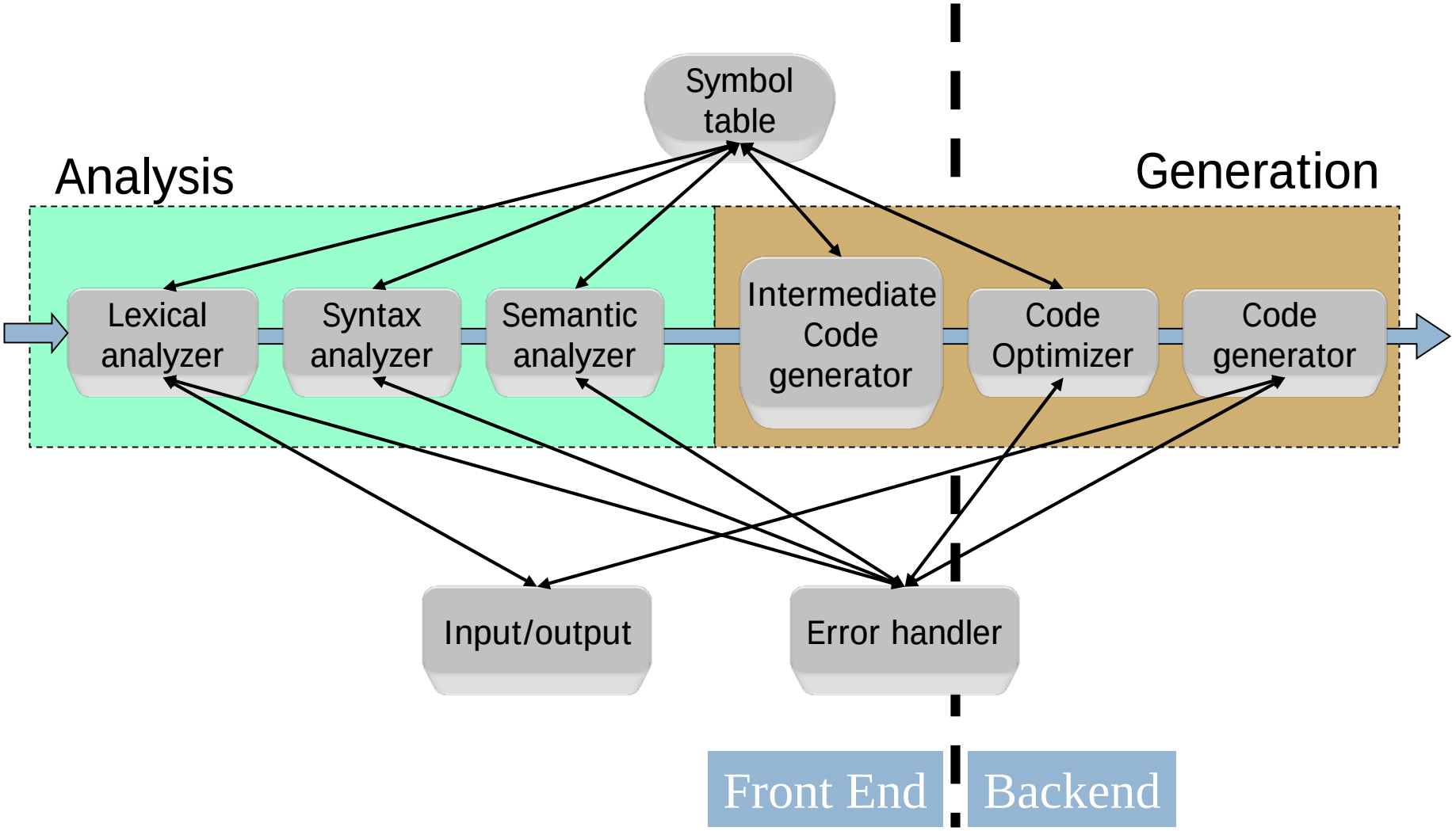
Phases of a Compiler

- **Analysis (Front-end):** All the operations related to the source program (Lexical, syntax and semantic analysis, intermediate code generation and handling of errors of every part).
- **Synthesis (Back-end):** Phases that depend on the intermediate code and/or the intermediate language (Code optimization, code generation and symbol-table management).



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

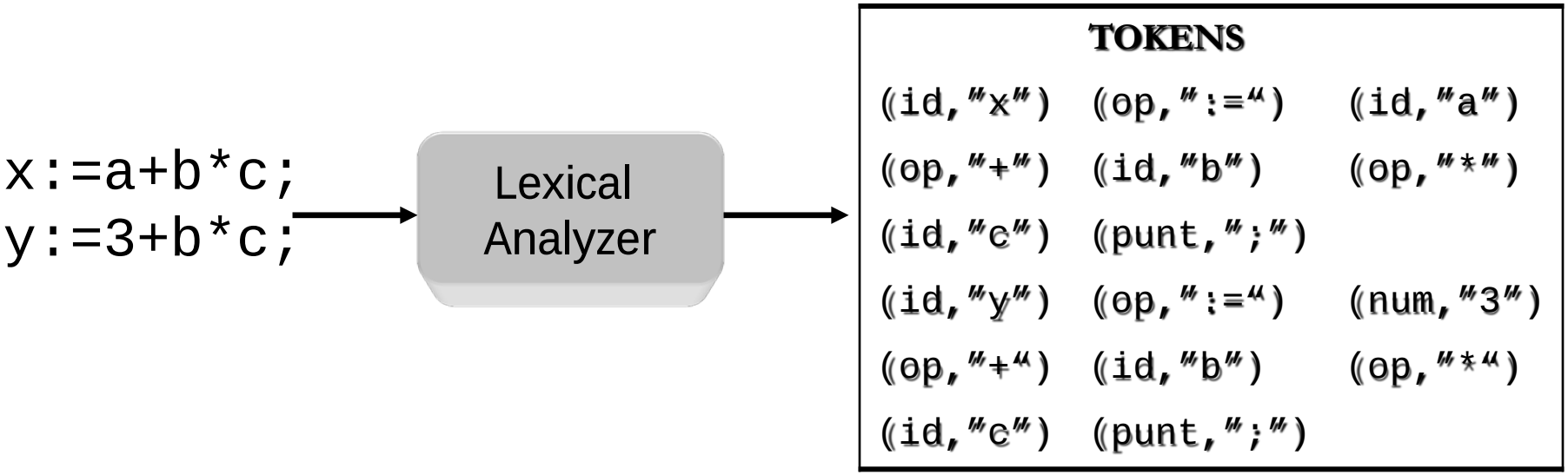
Phases of a Compiler



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Lexical Analysis (scanning)

- Natural language: "She wrote a program" words: "She" "wrote" "a" "program"
- The stream of characters of the source code is read from left to right and grouped into ***tokens*** (sequences of characters having a collective meaning)



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Lexical Analysis (scanning)

- TOKEN: < *token-name, attribute-value* >

Syntax analysis

Semantic analysis, Code generation
Symbol table

- *Token: Sequence of characters with a collective syntax meaning.*
- *Lexeme: Sequences of characters that makes up a token.*
- *Pattern: Rules that describe the lexemes of a specific token.*

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Syntax Analysis (parsing)

- Syntax analysis: To generate a tree-like intermediate representation (**syntax tree**) from the tokens generated in the lexical analysis.

- During hierarchical analysis the characters or tokens are grouped into grammatical phrases that are used by the compiler to synthesize output.

- Usually, the grammatical phrases are represented by a parse tree.

- The hierarchical structure of a program is usually expressed by recursive rules.

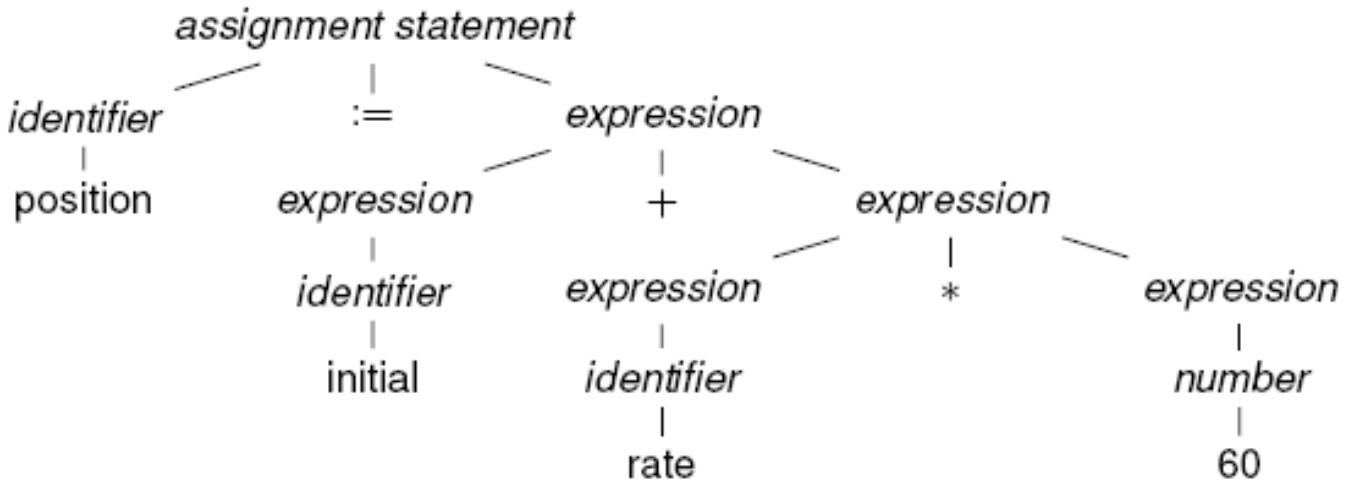
David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Syntax Analysis (parsing)

- Having the rules presented, we can represent the statement

  position := initial + rate * 60

  with the following parse tree:



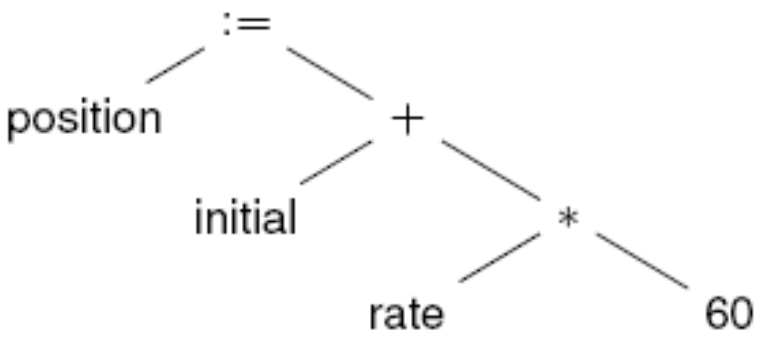
David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Syntax Analysis (parsing)

- The syntax analyzer uses a syntax tree instead.
- Having the rules presented, the syntax analyzer captures the statement

  position := initial + rate * 60

  with the following syntax tree:



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Compiler; Grammar

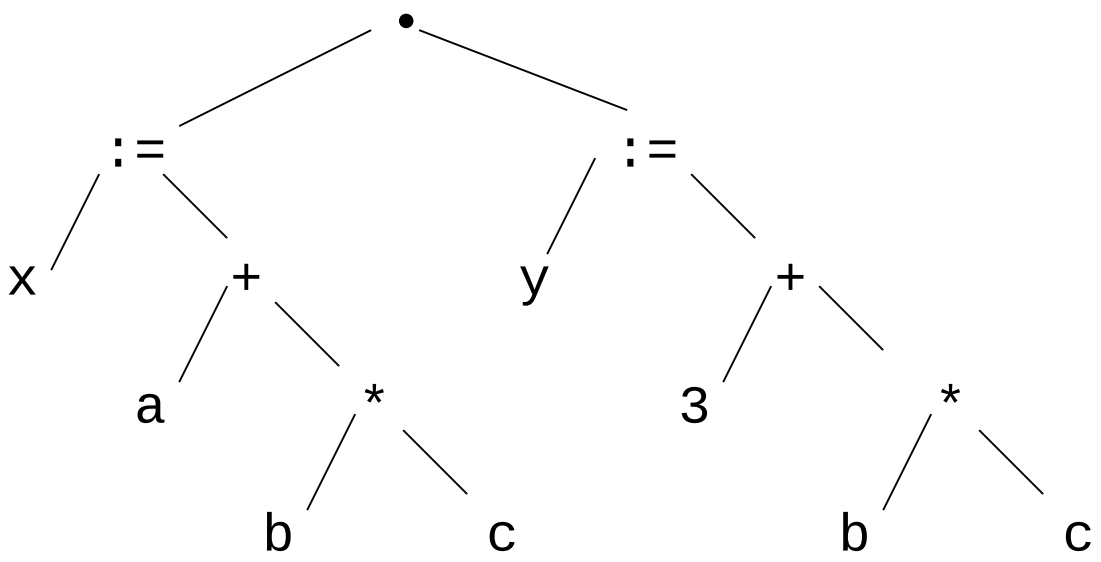
```
statements ::= statement “;” statements | statement
statement ::= assignment | conditional | iteration
assignment ::= variable “:=“ expresion
condicional ::= “if” condition “then” statements “else”
   statements
iteration ::= “while” condition “do” statements
expresion ::= variable-number “+” expresion |
              variable-number “*” expresion |
              variable-number “-” expresion |
              variable-number “/” expresion |
              variable-number
variable ::= [A-Za-z] [A-Za-z0-9]*
variable-number ::= variable | number
number ::= [0-9]+
```


David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Semantic analysis

- Checks for semantic errors and gathers type information; identifies the operators and operands of expressions and statements, and performs type checking.

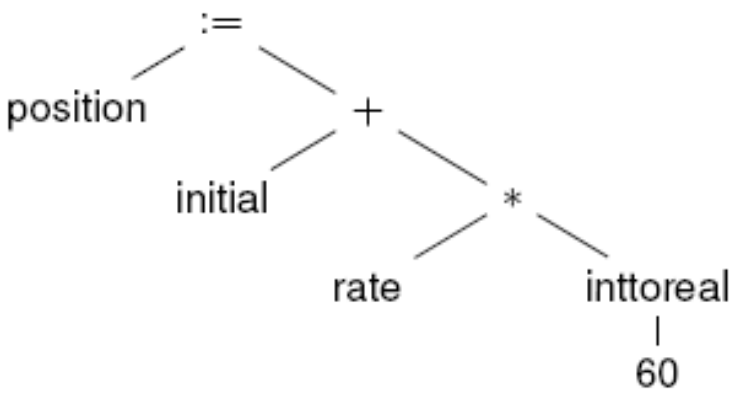
```
x:=a+b*c;
y:=3+b*c;
```



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Semantic analysis

- The syntax tree is used for the identification of the operators and operands of expressions and statements.
- Examples:
  - A semantic error is an undeclared variable.
  - During type checking incompatible operands and operators are identified, and conversions are made. If position, initial, and rate are declared to be real then:



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Symbol Table

- The symbol-table captures the identifiers found in the source program along with information about their attributes.
- When the identifier is a variable, the attributes may provide information about the type and scope of the identifier.
- When the identifier is a function, the attributes may provide information about the type returned, the number and type of its arguments, etc.
- The look-up table is populated through the three analysis phases.
- Identifiers are captured during lexical analysis, and their properties during syntactic (scope inferred) and semantic analysis (types inferred).

Intermediate Code Generation

- After the analysis phase some compilers generate an explicit intermediate representation of the source program.
- This representation should be easy to produce and translate into the target program.
- An example of representation is the "three-address code":
  - Each instruction has at most three operands.
  - Each instruction has at most one operator in addition to the assignment. Hence, the compiler has to decide the order in which the statements should be executed (for instance a multiplication should happen before an addition).
  - The compiler must generate a temporary name to hold the value computed by each instruction.
  - Some "three-address" instructions have fewer than three operands.

Intermediate Code Generation

- For example the statement

position := initial + rate * 60

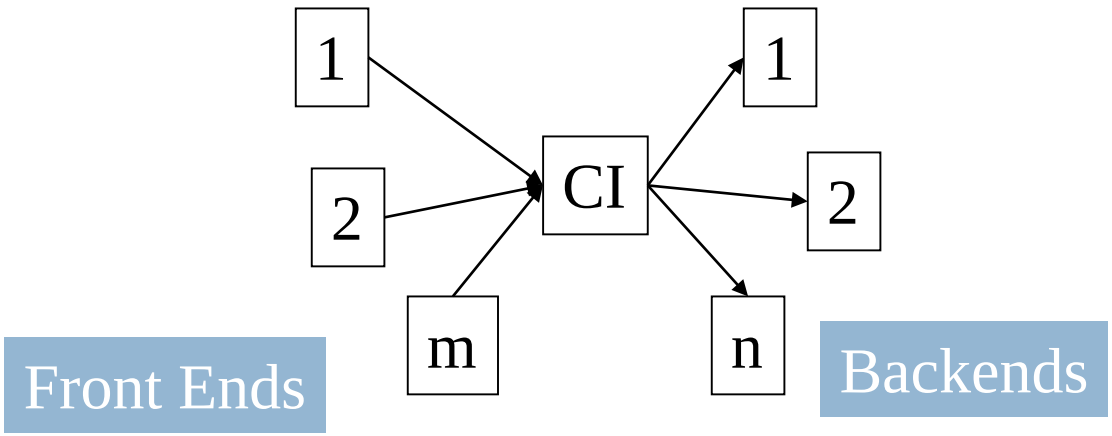
would result in the following representation:

tmp1 := inttoreal(60)
tmp2 := id3 * tmp1
tmp3 := id2 + tmp2
id1 := tmp3

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Intermediate Code Generation

- By using intermediate code, the complexity of developing compilers is reduced.
  - m front ends and n backends share a common intermediate code:



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Code Optimization

- The code optimizer attempts to refine the intermediate code into an equivalent faster-running intermediate code.

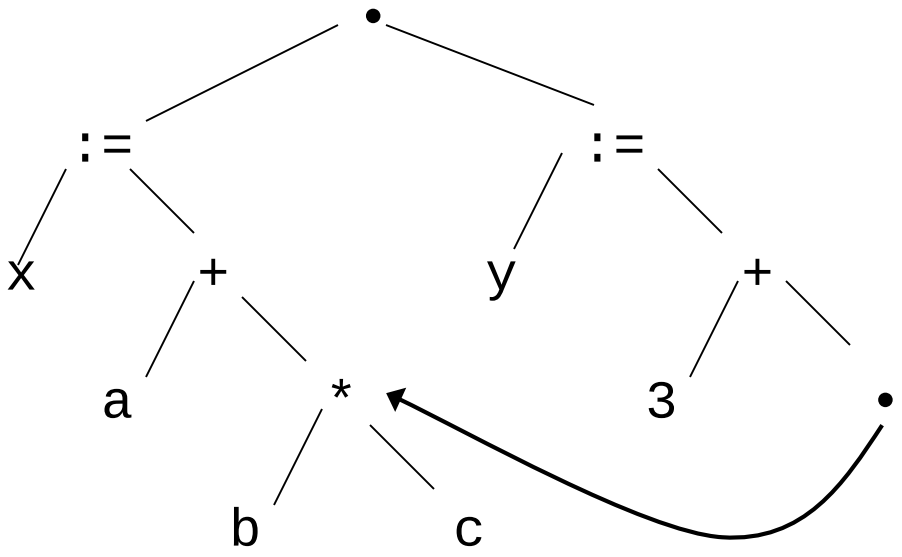
- Programming language-independent vs. language-dependent: Most high-level languages share common programming constructs and abstractions. Thus similar optimization techniques can be used across languages.

- Machine independent vs. machine dependent: Many optimizations that operate on abstract programming concepts (loops, objects, structures) are independent of the machine targeted by the compiler.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Code Optimization

```
x:=a+b*c;
y:=3+b*c;
```



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Code Generation

- The code generator generates the code in the target language, usually machine code or assembly.
- Register allocation is determined (i.e. the allocation of operands to processor registers). The goal is to keep as many operands as possible in registers to maximize the execution speed of software programs.
- The intermediate representation is translated into the target code through the selection of instructions.
- The instructions are then placed in the right order so that the semantics of the program are maintained while performance is maximized.

Code Generation

- Through code generation the optimal intermediate representation:

```
tmp1 := id3 * 60.0
id1 := id2 + tmp1
```

may be translated into:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

- R1 and R2 are registers, the F in each instruction tells us that instructions deal with floating-point numbers, and the # signifies that 60.0 is to be treated as a constant.

Error Handling

- A compiler must respond to possible errors in the source code.
- They can be detected in each one of the related phases:
  - Scanner : detection of stray tokens,
  - Syntax analyzer: invalid combinations of tokens
  - Semantic analyzer: type errors…

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

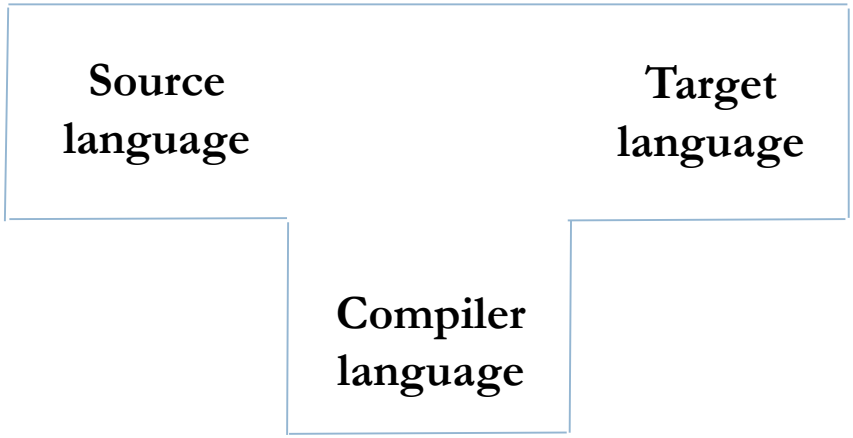
Error Handling

- Some of the errors may cause the compiler to end prematurely.
- Some of the errors are of lesser importance, and may allow the compiler to continue.
- For instance the GCC rarely halts prematurely. Instead it tries to recover from each error and present as many warnings and errors as possible.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

T-notation and Bootstrapping

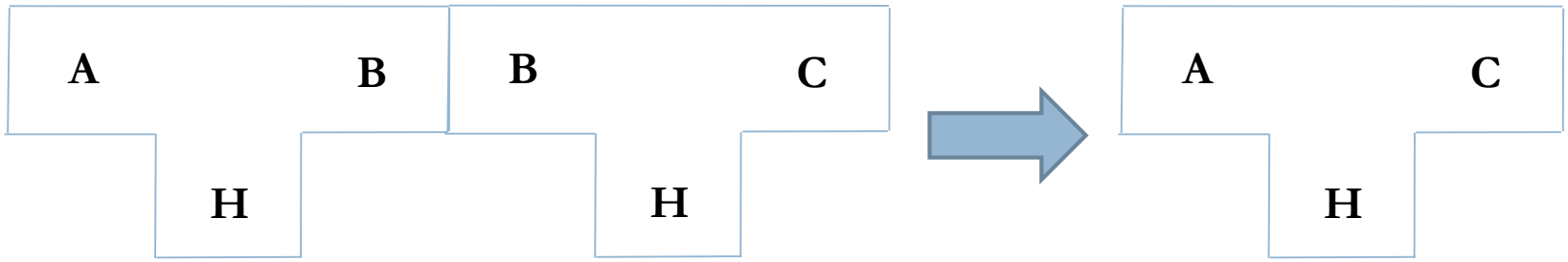
- T-notation represents the three languages that are involved for building a compiler:



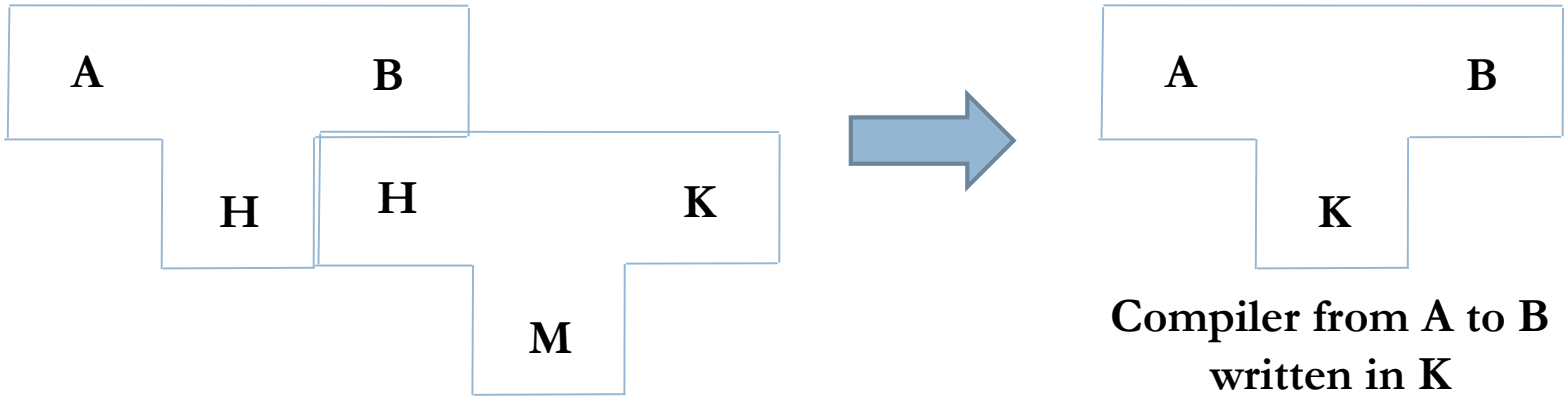
David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

T-notation and Bootstrapping

- Case 1:



- Case 2:



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Programming language design

- Building a compiler → design of the source language:
  - Express algorithms.
  - Well-defined syntax: programs easy to design, write, verify, understand and modify.
  - As simple as possible.
  - Provide data structures, data types and operations.
  - It should be possible to visualize the behavior of a program from its written form.
  - Reduced compiling and executing costs.
  - No program that violates the definition of the language should escape detection.
  - Not features or facilities that tie the language to a particular machine.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Programming paradigms

- *Imperative (40's):*
  - languages statement-oriented.
  - we move from state to state characterized by the current values of the registers, memory and external storage.
  - control structures.
  - E.g. FORTRAN,COBOL,ALGOL, PL/I, C, Pascal, and Ada.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Programming paradigms

- *Functional*:
  - function that must be applied to the initial state to get the desired result.
  - to build more complex functions until a final function is reached which computes the desired result.
  - control structures.
  - E.g. LISP and ML.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Programming paradigms

- *Rule-Based or Declarative*:
  - Conditions → actions.
  - to build more complex functions until a final function is reached which computes the desired result.
  - control structures.
  - E.g. Prolog.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Programming paradigms

- *Object-Oriented:*
  - extension of the imperative paradigm.
  - we build *objects* (data structures and operations that manipulate those data)
  - E.g. Smalltalk, Eiffel, C++, and Java.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Bibliography

- *Dragon book: Compilers: Principles, Techniques and Tools*. Aho, Sethi and Ullman. Pearson Addison Wesley, 2007.
- *The Theory of Parsing, Translation, and Computing, I: Parsing and volume II. Compiling*. Aho and Ullman. Prentice Hall, 1973.
- *Compiler construction : principles and practice*. K. Louden. Course Technology. 1997