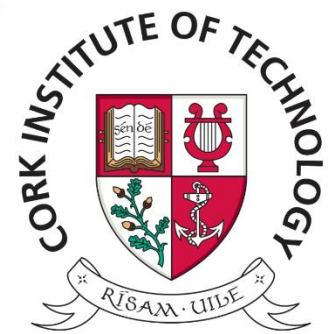
MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

Programming Language Design

Basics of Programming Language Design

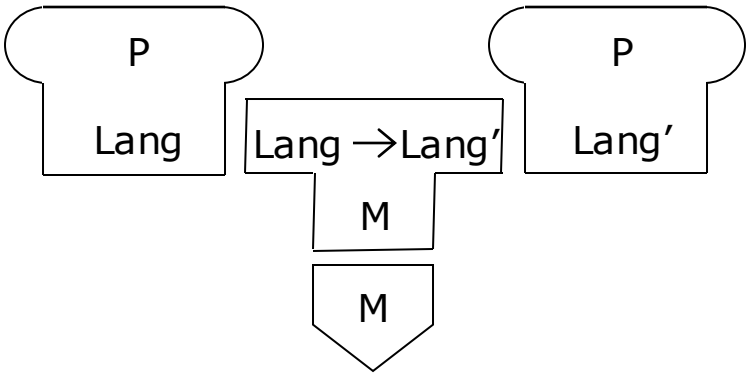
Francisco Ortin



Department of
Computer Science

Language, Processor and Translator

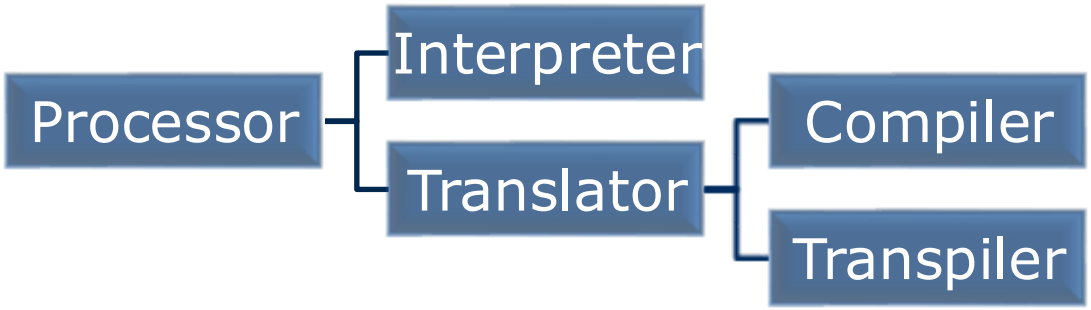
- A **programming language** is an artificial language for writing instructions, algorithms or functions to be executed by a computer
- A programming **language processor** is any system that manipulates programs, expressed in a particular programming language
- A language **translator** is a language processor that translates source programs in a programming language into equivalent target programs in another language



Example: EiffelStudio
Lang: Eiffel
Lang': C
M: 80x86

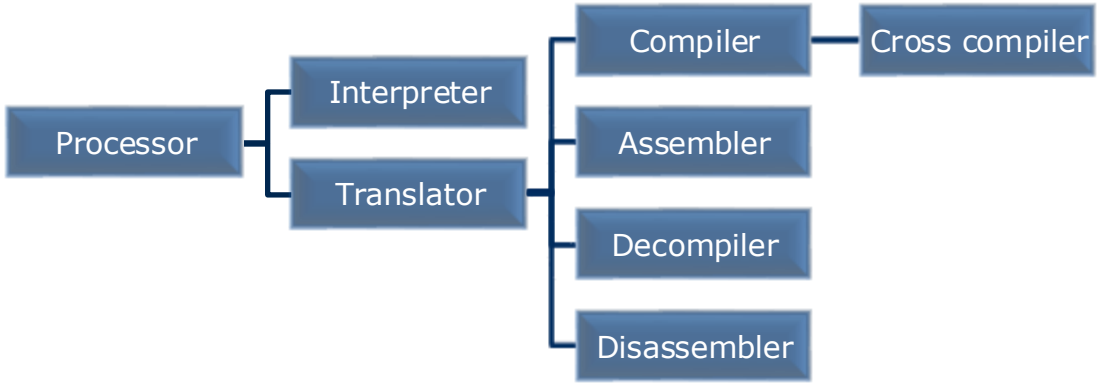
Compiler and Interpreter

- A **compiler** is a language translator that translates high-level programs into low-level machine code
  - A compiler is a specific type of language translator
- A **transpiler** is a language translator that translates source to source code at the same level of abstraction (e.g., the EiffelStudio example or TypeScript)
  - Source and target language may be the same (e.g., Prepack JavaScript optimizer and Python's 2to3)
- An **interpreter** is a program that executes (runs) programs written in a specific programming language



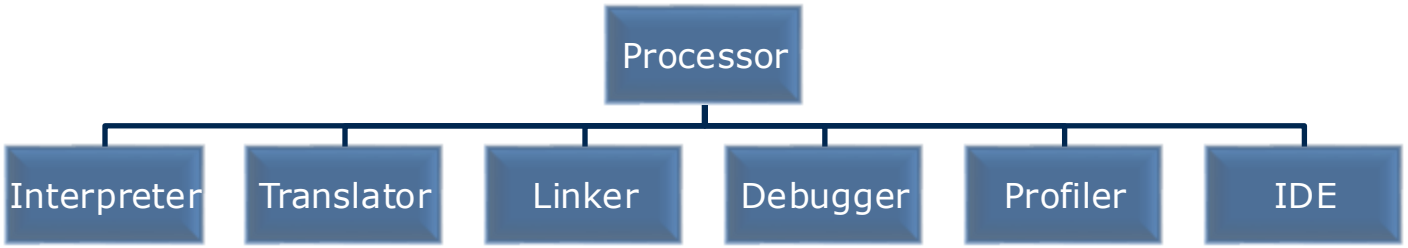
Other Translators

- **Assembler**: Translates assembly language into binary code
- **Disassembler**: Translates binary/machine code into assembly code
- **Decompiler**: Translates binary code into high-level programs
- **Cross compiler**: A compiler that generates binary code for a platform other than the one on which the compiler is running



Other Language Processors

- **Linker**: Collects code compiled separately in different binary files into an executable file
- **Debugger**: Analyzes the execution of a program to help determine execution errors
- **Profiler**: Collects statistics on the behavior of a program at runtime (typically to analyze execution time)
- **IDE**: Integrated Development Environments provides different services to programmers (editing, refactoring, debugging, profiling…)



Uses of PL Design

1. Model-Driven Development (**MDD**) and Engineering (**MDE**). Models (programs) are defined with meta-models (grammars) and transformed with graph-based transformations (code generation), using PL processing techniques
2. **Domain-Specific Languages** (DSL) are many times designed and implemented for different software applications
   - View templates (Razor and Velocity), hardware description (VHDL and Verilog) and sound and music processing (CSound)
3. Design and processing of **data formats**. Complex data formats require the use of PL processing:
   - GraphViz DOT, AutoLISP for AutoCAD and GrGen (graph-structured data)
4. Formal **software verification** to ensure that software meets all the expected requirements
   - Amazon Web Services (using TLA+), the PikeOS real-time operating system and the C CompCert compiler

Uses of PL Design

5. **Security**. It is common to (de)compile code to understand its behavior, analyzing code generation templates

   Language-based security strengthens the security of applications by using the properties of programming languages

6. Design and implementation of **visual languages** for final users

   Node-RED, NETLab and miniBloq languages for IoT

7. **Framework design and implementation** require lots of meta-programming techniques (programming programs)

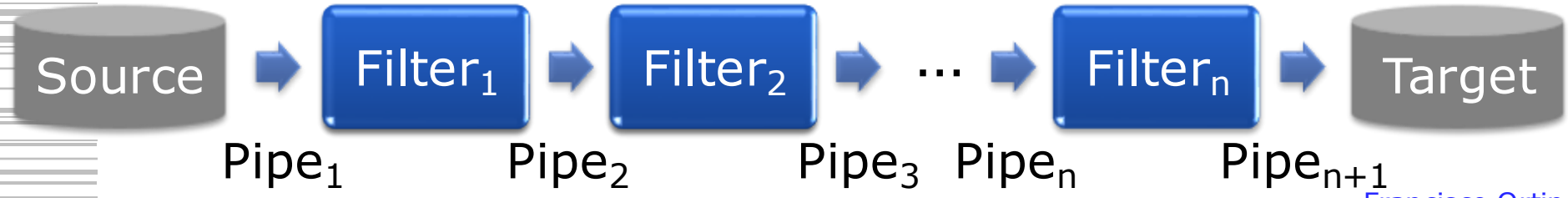
   Ruby on Rails, Django and Hibernate

8. **Natural Language Processing** (NLP) uses lexing and parsing techniques to process the ambiguous natural language

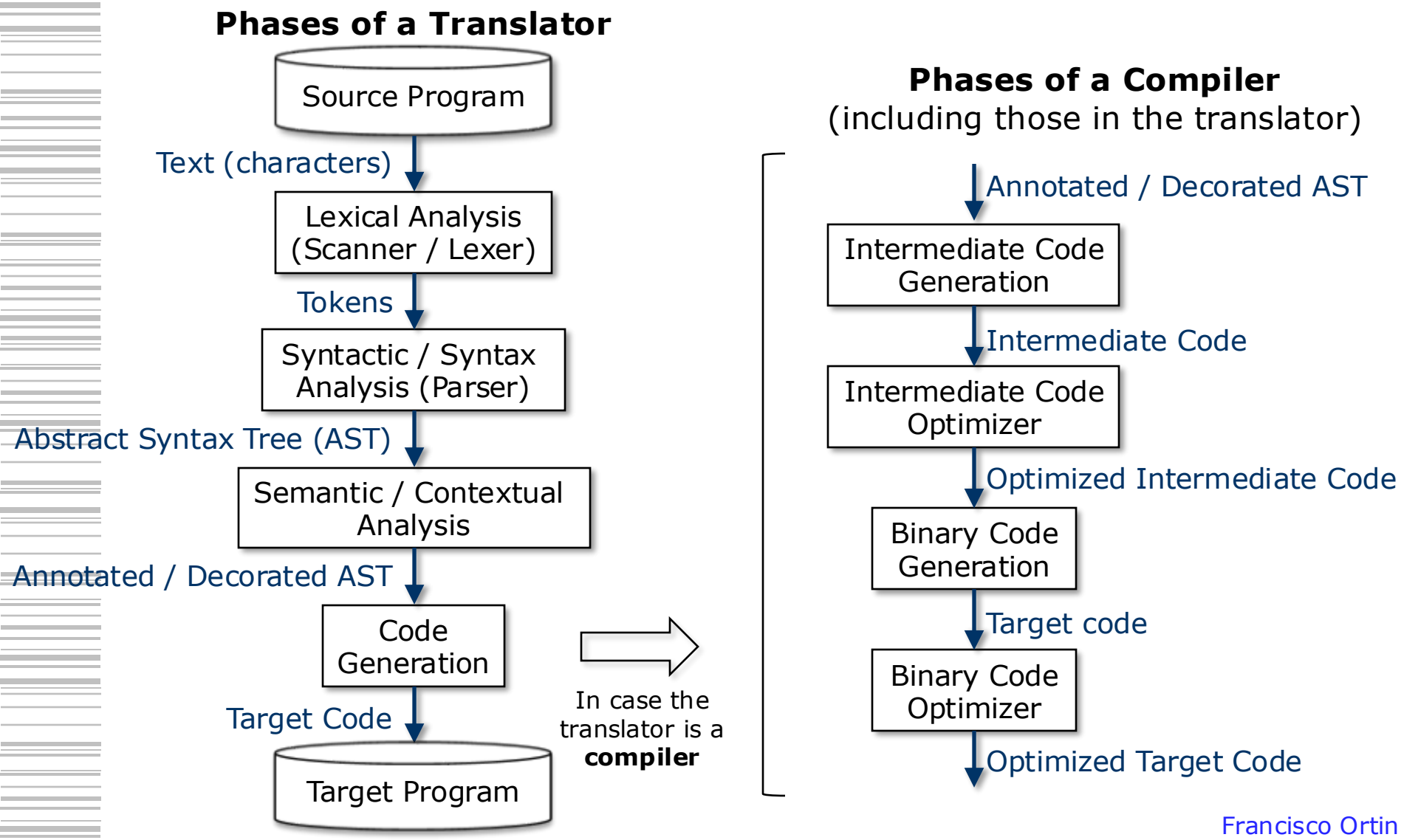
   Probabilistic grammars, grammar induction, tokenization, text segmentation, GNNs, Graph Transformers

Architecture of Translation Process

- The architecture of a language translator follows the **Pipes and Filters** architectural pattern:
  - This pattern provides a structure for systems that **process a stream of data**
  - Each processing step is encapsulated in a **filter** component
    - Consumes data from its input and produces (potentially different) data on its output
  - Data are transformed and passed through **pipes** between adjacent filters

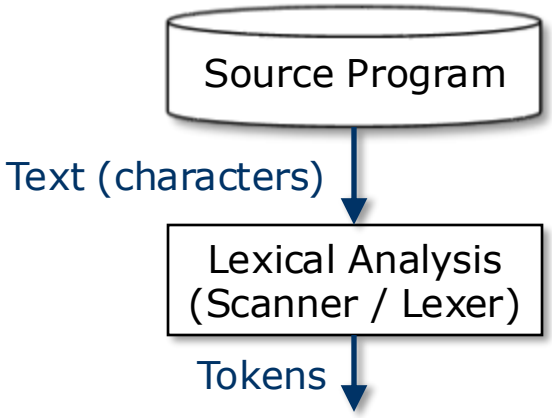


Phases of a Translator / Compiler



Lexical Analysis

- Phase that collects sequences of characters into meaningful units, called **tokens**
  - Tokens are commonly represented with integers for performance reasons
- The lexical analysis is performed by the **scanner** or **lexer**
- The scanner also discards some unnecessary characters (e.g., new line, tabs, comments…)



Source:

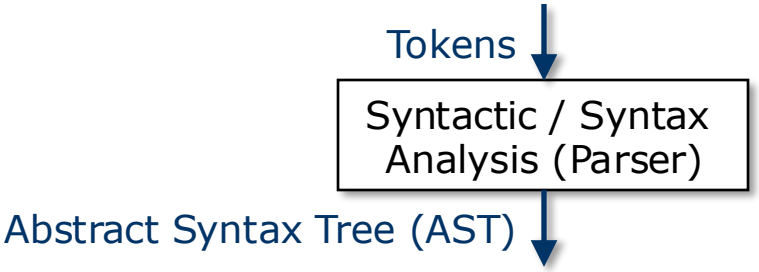
```
int a,b;
read b;
// addition
a = b+3;
```

Tokens:

```
INT ID ',' ID ';'
READ ID ';' ID '='
ID '+' INT_CONSTANT
';'
```

Syntactic / Syntax Analysis

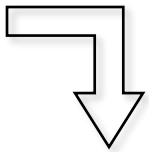
- Phase that determines the **structure of the source program**
  - It is performed by the **parser**
- Receives a one-dimensional sequence of **tokens**…
- …and creates a multidimensional **parse tree** representing the structural elements of the source program and their relationships
  - Abstract Syntax Trees (**AST**s) are simplified parse trees
- If a parse tree cannot be found for the input token sequence, a **syntax error** is produced



Syntactic / Syntax Analysis

Source:

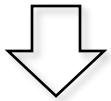
```
int a,b;
read b;
// addition
a = b+3;
```



Lexical analysis

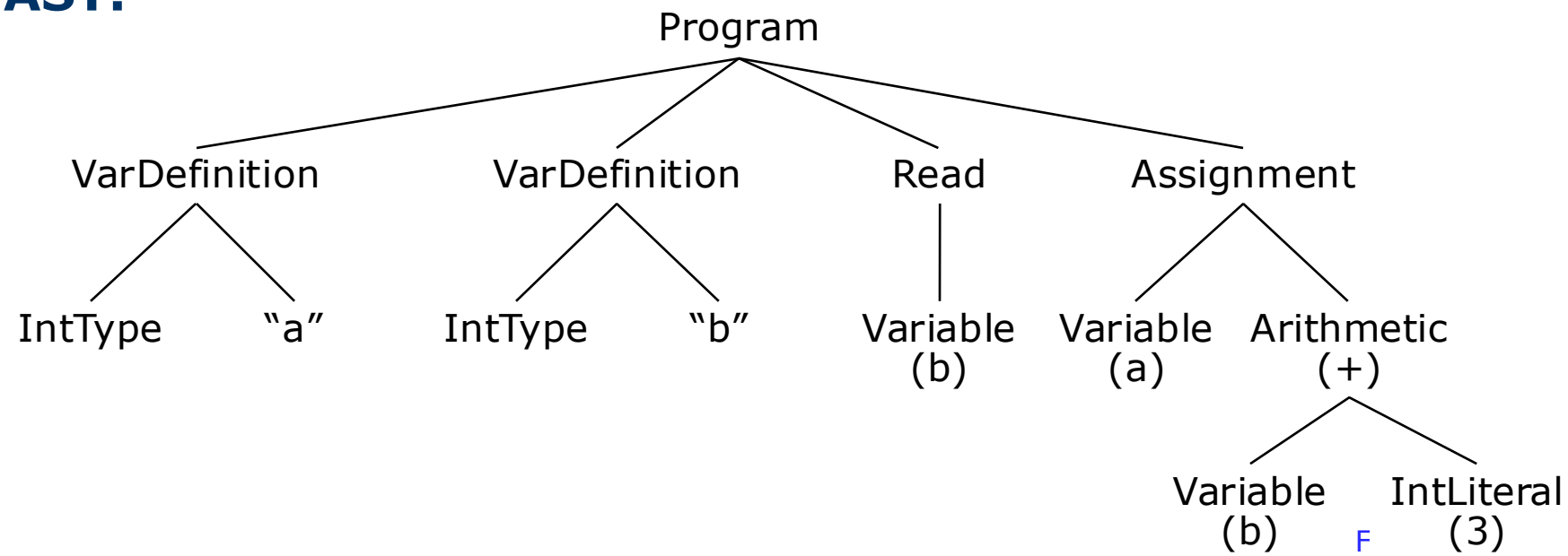
Tokens:

```
INT ID ',' ID ';' READ ID ';' ID '=' ID '+' INT_CONSTANT ';'
```



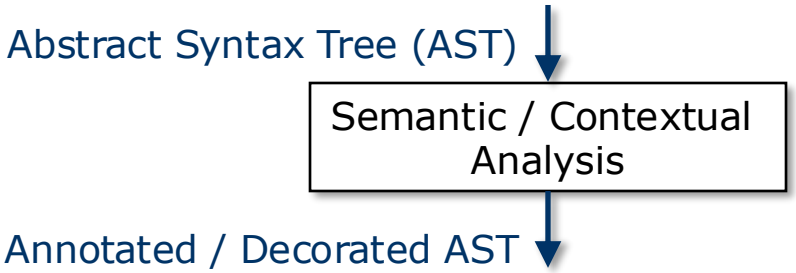
Syntax analysis

AST:



Semantic / Contextual Analysis

- Phase that
  1. enforces the **semantic** (**constraint**) **rules**, not checked by the parser
  2. adds **semantic information** to the AST (e.g., types) ⇒ the AST is annotated/decorated with that information
- Performed by the **semantic analyzer**
- If any semantic rule is not fulfilled by the input program, a (semantic) error is shown



Semantic / Contextual Analysis

- Are the following programs correct?

```
int b;
read b;
a = b+3;
```

```
int a;
double b;
read b;
a = b+3;
```

```
int a,b;
read b;
a = b+3;
```

Semantic / Contextual Analysis

- Are the following programs correct?

```
int b;
read b;
a = b+3;
```

Variable "a" has not been defined.

```
int a;
double b;
read b;
a = b+3;
```

```
int a,b;
read b;
a = b+3;
```

Semantic / Contextual Analysis

- Are the following programs correct?

```
int b;
read b;
a = b+3;
```

Variable "a" has not been defined.

```
int a;
double b;
read b;
a = b+3;
```

A double cannot be assigned to an integer.

```
int a,b;
read b;
a = b+3;
```

Semantic / Contextual Analysis

- Are the following programs correct?

```
int b;
read b;
a = b+3;
```

Variable "a" has not been defined.

```
int a;
double b;
read b;
a = b+3;
```

A double cannot be assigned to an integer.

```
int a,b;
read b;
a = b+3;
```

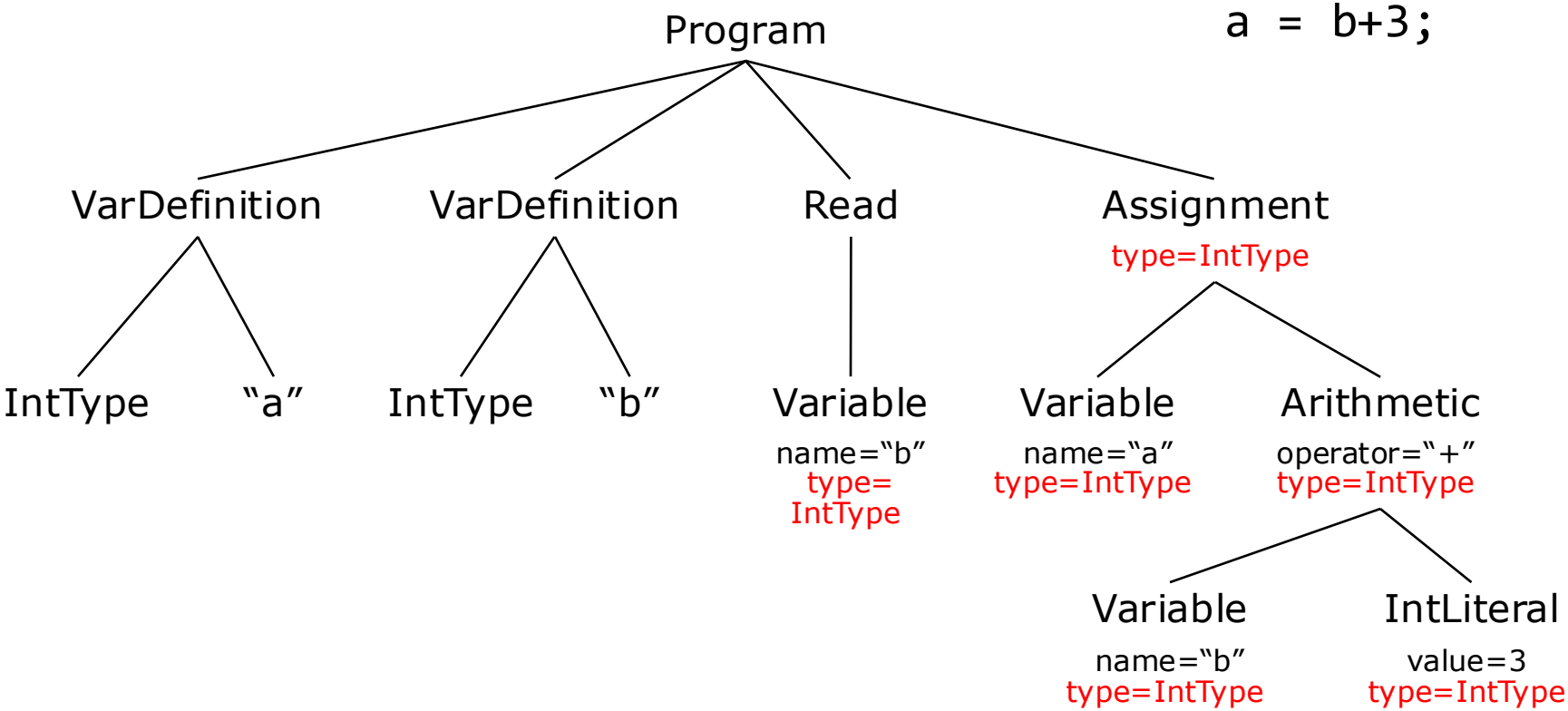
Semantic / Contextual Analysis

- If no semantic error exists, the AST is decorated

Annotated/decorated AST:

Source:

```
int a,b;
read b;
a = b+3;
```



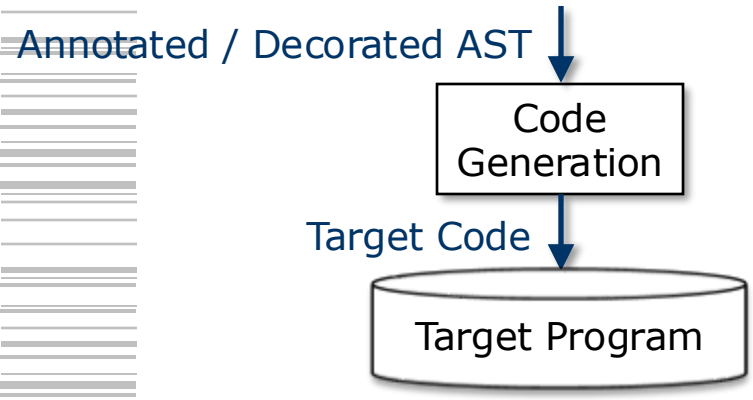
Code Generation

- Phase that **generates** the **target code**
  - Performed by the code generator
- The target program must have the **same semantics** (behavior) as the source program
- If the translator is a compiler,
  - an **intermediate** (platform agnostic) **translation** is commonly performed (e.g., JVM or .Net CLI)
  - there may be intermediate phases **optimizing** the (intermediate and target) generated code

Code Generation

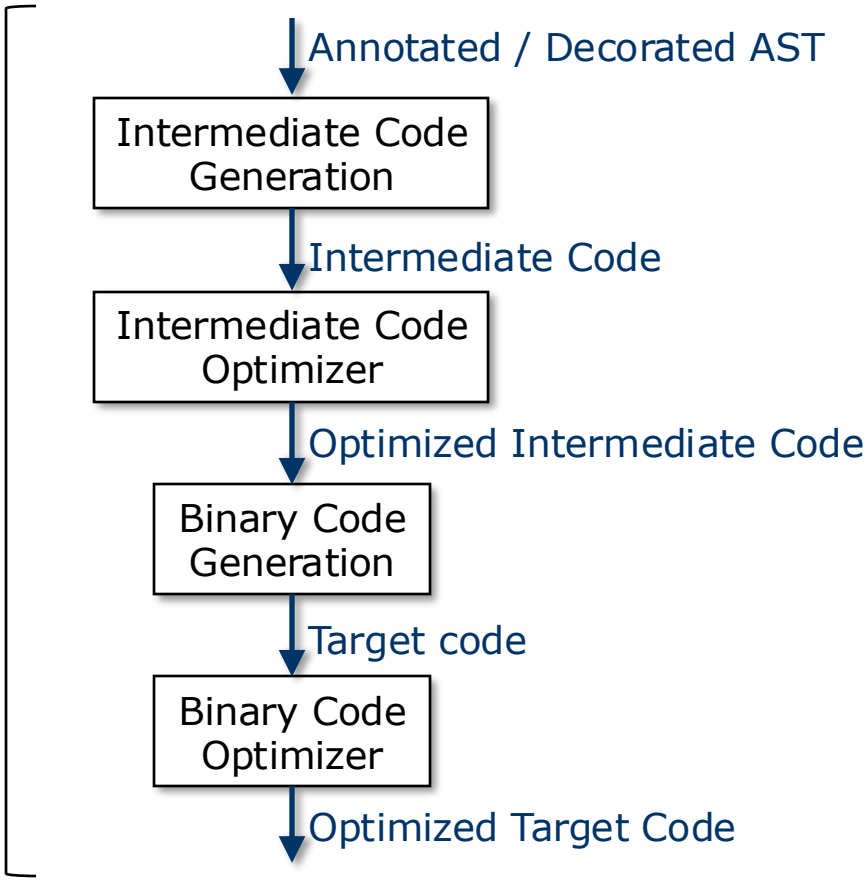
- The code generator requires the **semantic information** (e.g., types) added to the AST by the semantic analyzer

Translator



In case the translator is a **compiler**

Most professional compilers

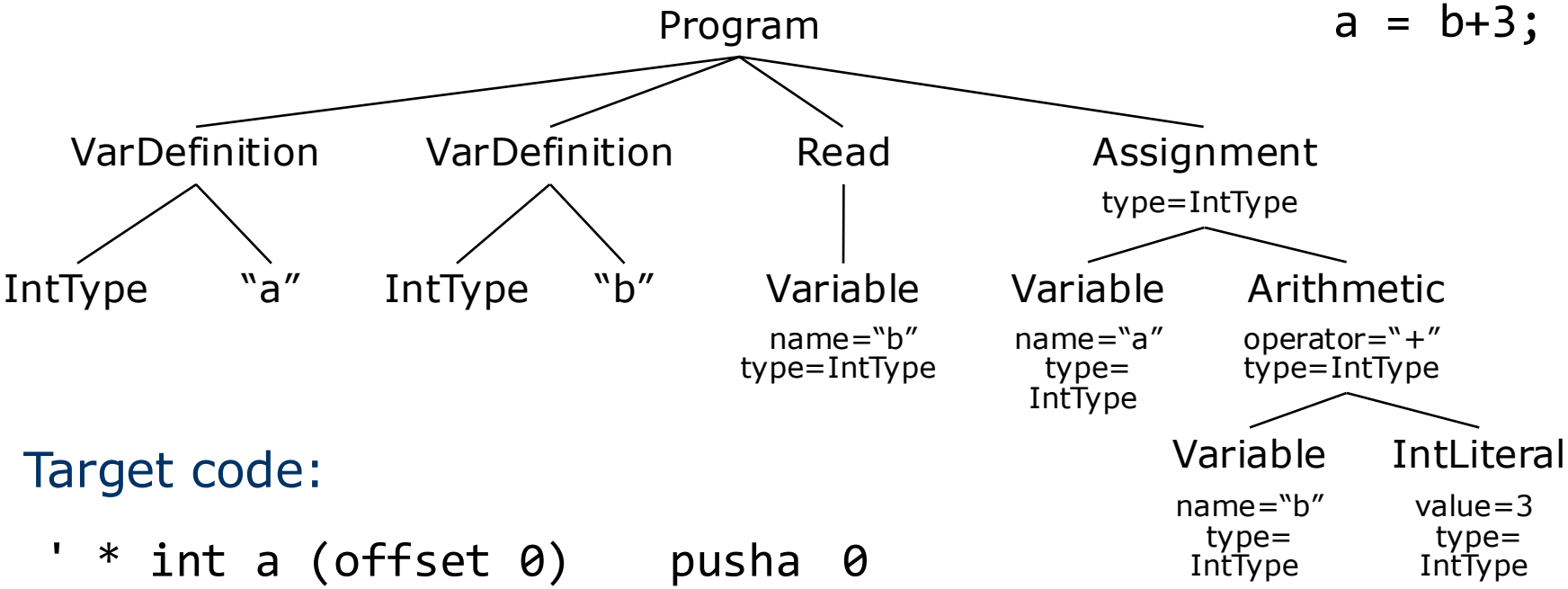


Code Generation

Annotated/decorated AST:

Source:

```
int a,b;
read b;
a = b+3;
```



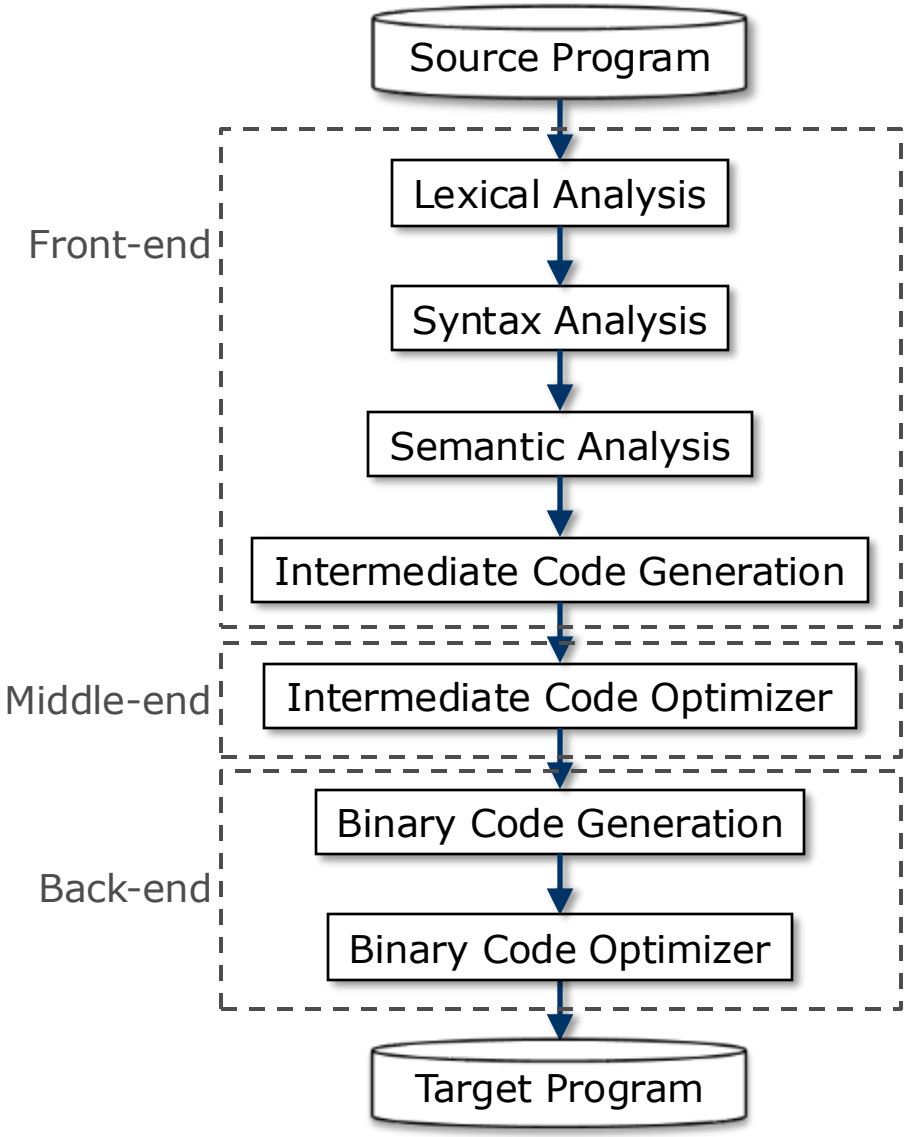
Target code:

```
' * int a (offset 0)
' * int b (offset 2)

pusha  2
ini
storei
```

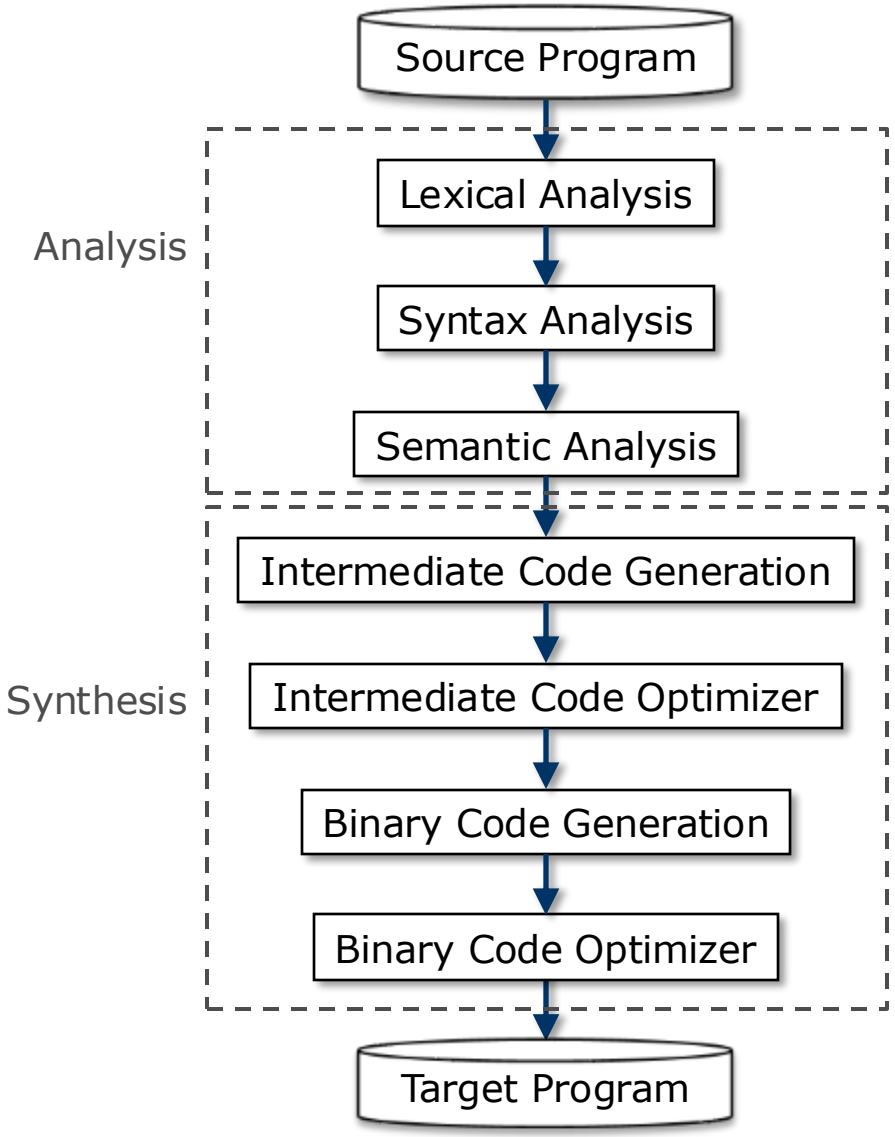
```
pusha  0
pusha  2
loadi
pushi  3
addi
storei
```

Front- and Back-end



- **Front-end**: phases that deal with the source language ⇒ independent of the target language
- **Middle-end**: phases that deal with the intermediate representation ⇒ independent of the source and target languages
- **Back-end**: generates target (binary) code, performing different target-specific optimizations ⇒ independent of the source language

Analysis and Synthesis



- **Analysis**: phases that analyze the source program
- **Synthesis**: phases that produce (generate) a representation of the source program in another language (not necessarily in a persistent store)

Mandatory Activity

- Identify the phases with the following responsibilities
  1. Perform static type checking
  2. Ignore comments in the source code
  3. Show syntax errors
  4. Count the line and column numbers in the source program
  5. Perform implicit conversions of expressions
  6. Check the correct context of expressions (e.g., an identifier could appear on the left-hand or right-hand side of assignments, or `/` operator is not unary)
  7. Check that a variable has been previously defined
  8. Compute the offsets of record (struct) and object fields
  9. Traverse the AST

Bibliography

- Andrew W. Appel. Modern Compiler Implementation in Java, 2 Edition. Cambridge University Press, 2002.
- Alfred V. Aho, Monica S. Lam. Compilers: Principles, Techniques, and Tools, 2 Edition. Addison Wesley, 2006.
- David A. Watt. Programming Language Processors in Java: Compilers and Interpreters. Prentice Hall, 2000.