ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

CORSO 72671
COMPLEMENTI DI LINGUAGGI DI PROGRAMMAZIONE

COSIMO LANEVE
cosimo.laneve@unibo.it

welcome

COMPLEMENTI DI LINGUAGGI DI PROGRAMMAZIONE

code transformation and analysis

CODE TRANSFORMATION AND ANALYSIS

* course informations
* why study CODE TRANSFORMATION AND ANALYSIS?
* a quick history of CODE TRANSFORMATION AND ANALYSIS
* the structure of a translator/analyzer
* the arguments in this course
* `ANTLR`
* some background

COURSE INFORMATIONS

instructor: me. If you want to discuss

- send email to `cosimo.laneve@unibo.it`
- come in my studio during office hours
- fix ad-hoc meetings

this year there will be a TUTOR:

Marco Bertoni

- exercises
- assistance in the team project

`marco.bertoni5@unibo.it`

e-learning page: download material

- go to the course page at

`https://www.unibo.it/sitoweb/cosimo.laneve/teachings`

grading policies

- **written examination** max score: **26** (min **15**)
- **team project** max score: **6**

both are necessary

EXAMINATION

* there are 6 **written examination** sessions (2 in Jan/Feb; 3 in May/Jun/Jul; 1 in Sept.)
* the **team** of the team project must be composed of 2 students (exceptionally 3)
* you may release the project **within a deadline** (usually in June: we will find an agreement — miss the deadline = -1 point)
* the score of the project is individual: **one may pass** and **another of the group may fail** (in this case a new project must be delivered)
* the project must be uploaded on `virtuale` — **don't send the project by email!**

COURSE INFORMATIONS

course prerequisites

* Programming Languages (code `04138`)
* **Fondamenti Logici dell'Informatica** (Laurea Magistrale, 1^ sem), <u>not necessary, but highly recommended</u>

references: downloadable material

* Torben Morgensen: **Basics of Compiler Design**. 2010. Downloadable from http://www.diku.dk/~torbenm/Basics/
  * you may also download it from the e-learning website
* Terence Parr: **Language Implementation Patterns**. 2010 (this is for `ANTLR`) [look for it by yourself…]
* material on the e-learning website

MOTIVATIONS

modern software development requires fast and sophisticated code transformation and analysis tools

* `Java` code is verified by the Virtual Machine before execution
* `Facebook`, before releasing its mobile apps, always submits them to a tool that finds bugs without running the code
* `Google Chrome` and `Mozilla Firefox` analyze and optimize `JavaScript` code to make browsers more responsive
* performance-critical applications require compilers that derive correct and optimized machine code from high-level source code

CODE TRANSFORMATION AND ANALYSIS

ADDITIONAL MOTIVATIONS

- it is hard because code transformations and analyses must be **semantically correct**
- see the **application of the theory**
- learn **how programming languages work**
- learn **how a development tool works** and how to use it
- focus on **concepts** that we use all the time in a **translation**: data structures, model-driven code generators, source-to-source translators, source analysers, interpreters
  - most of us will never build compilers for general-purpose programming languages (which requires a strong computer science background)

OBJECTIVE OF THE COURSE

discuss modern code transformation and analysis techniques and illustrate their implementation

* we will refer to the `ANTLR` framework that is widely used in academia and industry to build all sorts of languages, tools, and frameworks
* `Twitter` search uses `ANTLR` for query parsing, with over 2 billion queries a day
* you will apply the theory by extending a small, yet expressive and powerful language, by means of the `ANTLR` framework

CLP IS HARD

the average score of the last year is **25.2**

the course combines theory and practice

* therefore you need to know in detail the theory and the development tool to write the sw
* e.g. the grammars and the syntax trees and the **visitor process**; the semantic rules and the proof trees and **their implementation**, the code generation and the **assembly language**

CLP IS HARD

CLP requires a strong background (and motivation)

comments from students:

* Tutto il corso è un problema. In particolar modo non si può alla magistrale di informatica essere obbligati a fare questo esame in quanto dovrei poter scegliere cosa vorrei fare. Mi pento di non essermene andato a Milano a studiare dove forse sarei stato valorizzato di più e avrei fatto qualcosa di più interessante. Il docente almeno qualche materiale di supporto in più potrebbe fornirlo dimentica sempre tutto e non ci dà sicurezza sulla struttura dell'esame poco chiara
* Tutto è piuttosto difficile da capire
* Le prime lezioni sono state difficili da capire se non si avevano già conoscenze di questa materia. Le slide non sono molto chiare. Fornire più conoscenze di base e dare meno cose per scontate. Migliorare le slide. Fare più esercizi completi.
* Gli esercizi svolti in classe sono subito esercizi d'esame o simili, sarebbe meglio una difficoltà più graduale nella presentazione degli esercizi in modo da consolidare meglio i concetti.
* Troppo difficili gli argomenti del corso per studenti che alla triennale non hanno avuto corsi, i quali li permettono di avere una base già solida su questi argomenti

CLP IS HARD

if you are not strongly motivated, consider to ask for

variazione piano di studi

you may replace **CLP** with one of these

* Sistemi Context-Aware
* Didattica dell'Informatica
* Internet of Things
* Artificial Intelligence, Blockchain e Criptovalute nello sviluppo software
* Human Data Science

as a motivation you may write one of these

* La scelta del corso X deriva dal desiderio di approfondire le competenze dell'area Y
* La scelta del corso X deriva dalla volonta` di aumentare i possibili sbocchi professionali del mio percorso formativo anche con esso.

A SHORT HISTORY OF CODE TRANSFORMATION & ANALYSIS

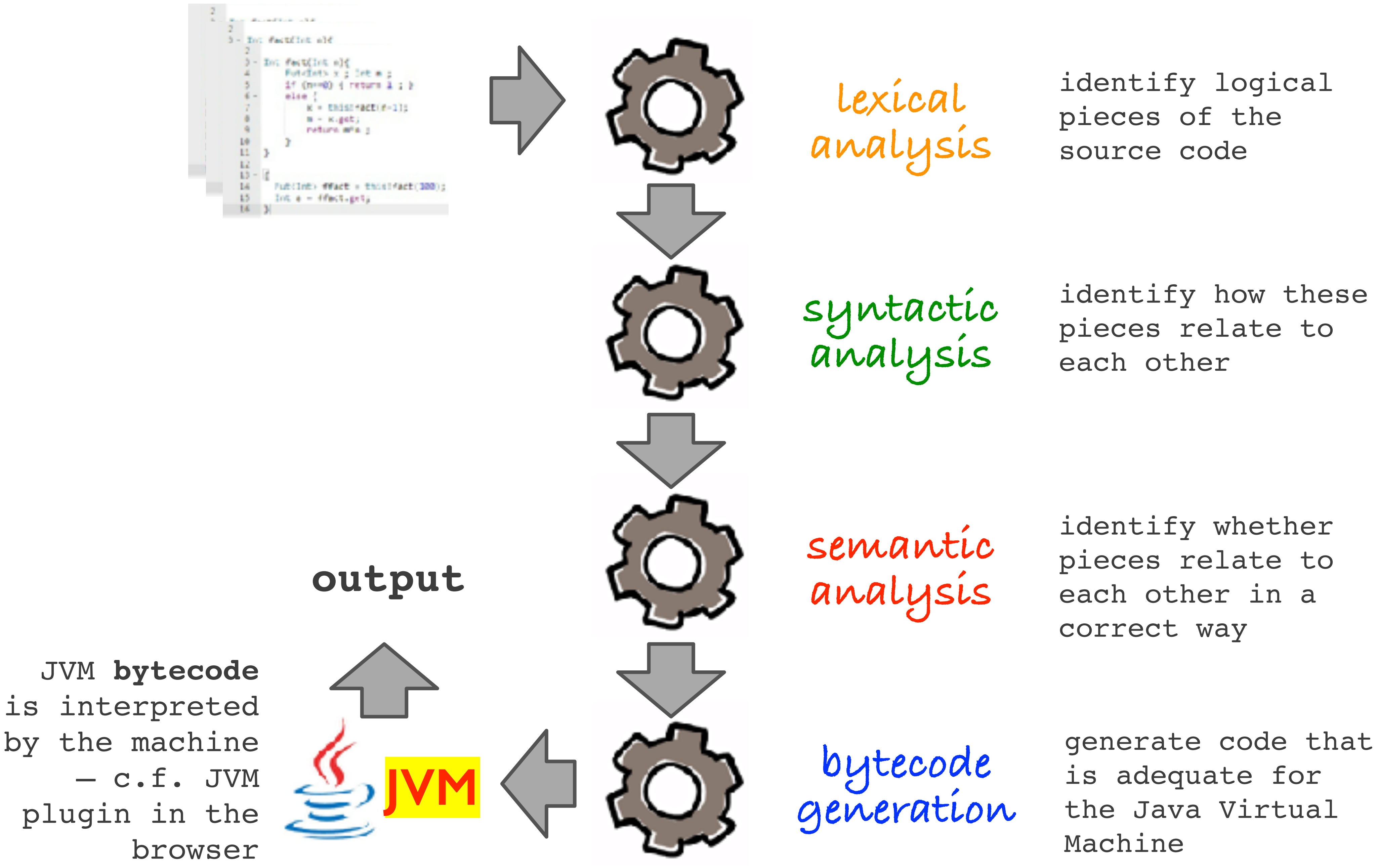
* **first**, there was nothing
* **then**, there was machine code
* **afterwards**, there were assembly languages (see Computer Architecture, code 11925)
* **programming was expensive**: 50% of costs for machines went into programming
* **high-level languages were/are conceived**: people needs translators to low-level codes (compilers and interpreters)
* **commercial and critical codes** require powerful translators removing bugs and delivering optimized code

CODE TRANSFORMATIONS: COMPILATION AND INTERPRETATION

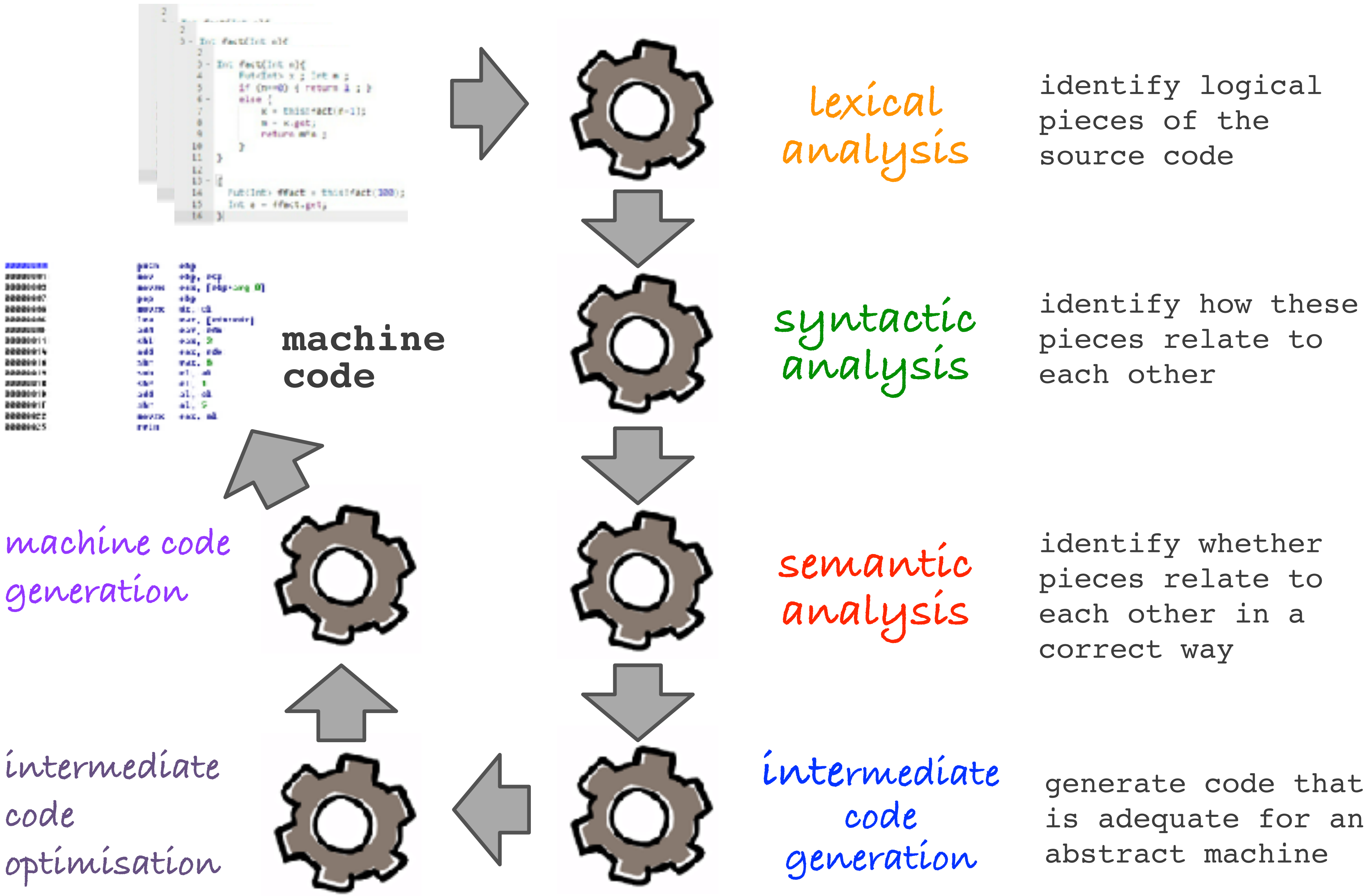
COMPILERS	INTERPRETERS
* transform code written in a high-level programming language into the machine code, at once, before program runs, * compiled code runs faster * display all errors after compilation * take an entire program	* covert each high-level program statement, one by one, into the machine code, during program run * interpreted code runs slower * display errors of each line one by one * take a single line of code

* **compiled languages**: `C`, `C++`, `C#`, `Scala`, `Java`
  * `Java` is compiled into bytecode which is then interpreted by the Java Virtual Machine
* **interpreted languages**: `PHP`, `Perl`, `Ruby`, `JavaScript`, `Python`
  * `Python` is compiled into bytecode which is then saved and executed afterwards instead of the source code

COMPILERS AND INTERPRETERS: JAVA

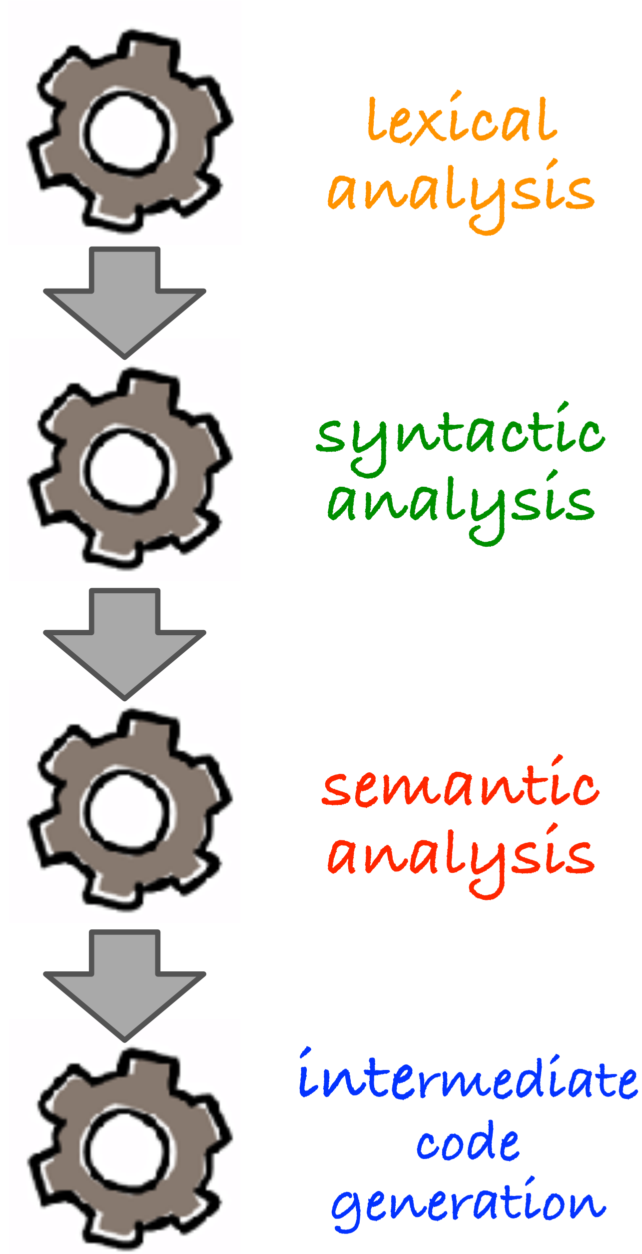


COMPILERS: C (NO INTERPRETERS)



A COMPILER AT WORK

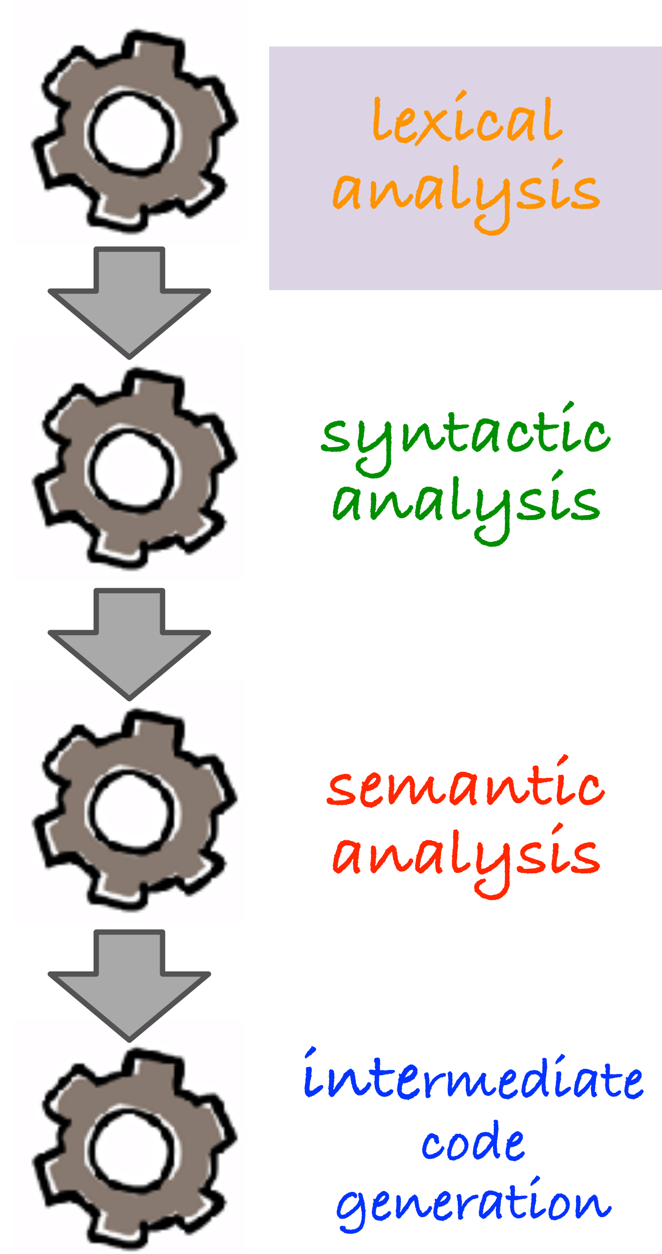
```
while (y < z) {
    int x = a + b;
    y = y + x;
}
```



A COMPILER AT WORK

```
while (y < z) {
    int x = a + b;
    y = y + x;
}
```

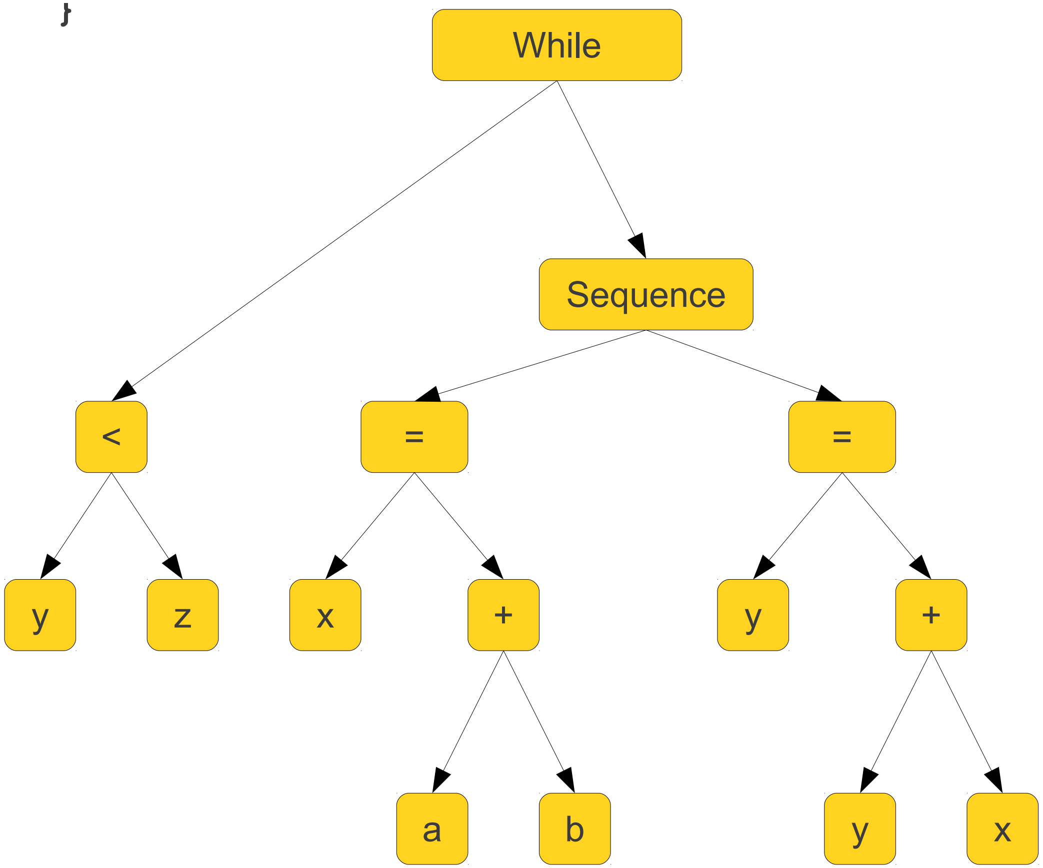
```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_Assign
T_Identifier y
T_Plus
T_Identifier x
T_Semicolon
T_CloseBrace
```

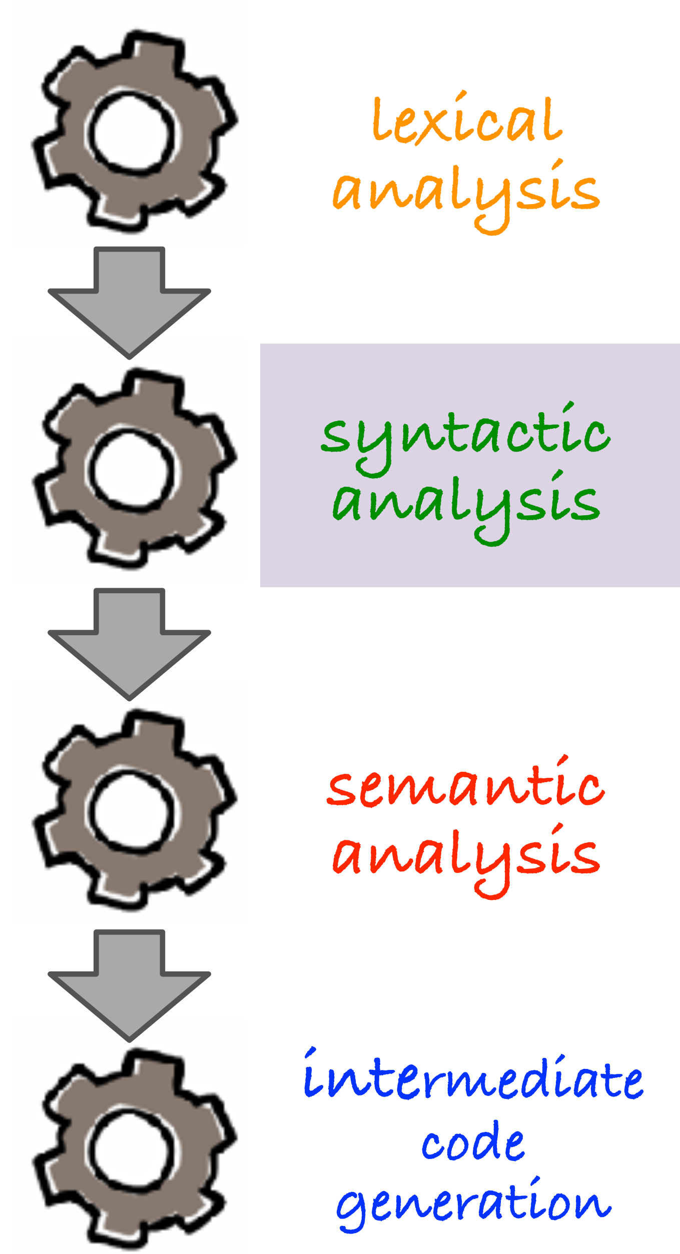


A COMPILER AT WORK

```
while (y < z) {
    int x = a + b;
    y = y + x;
}
```

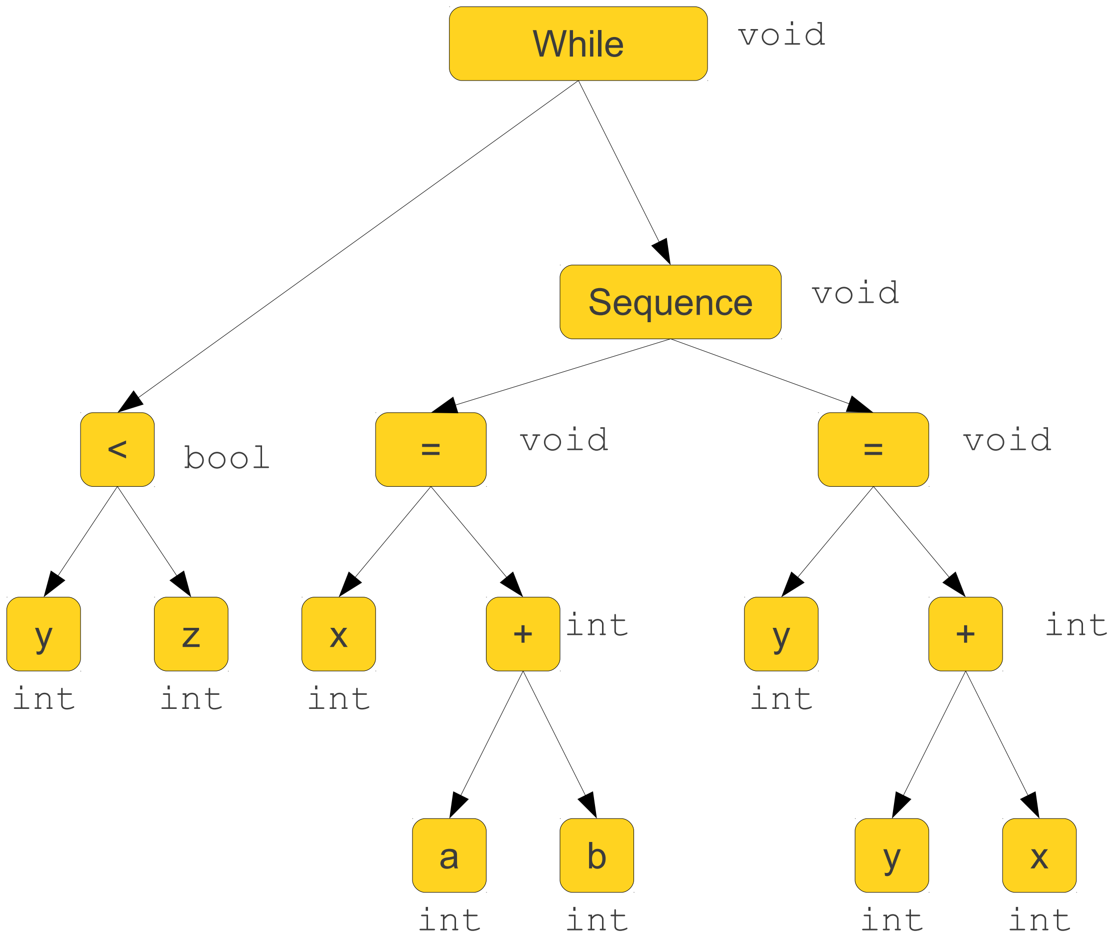
```
T_While
T_LeftParen
T_Identifier y
T_Less
```

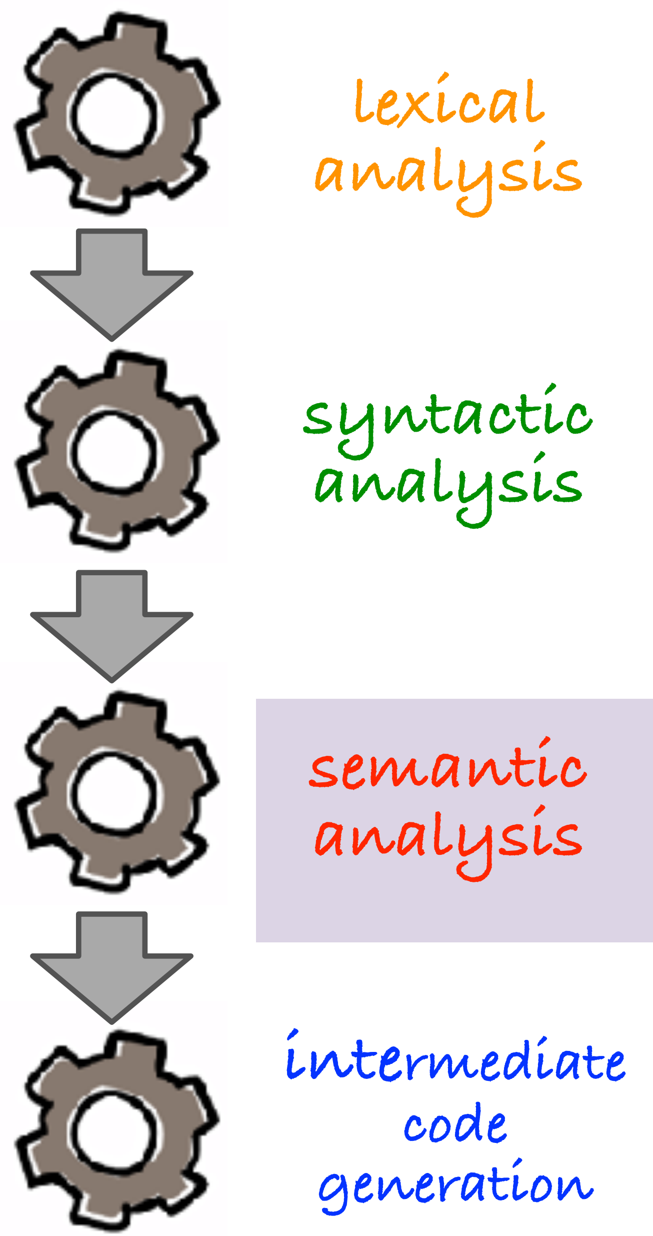




A COMPILER AT WORK

```
while (y < z) {
    int x = a + b;
    y = y + x;
}
```





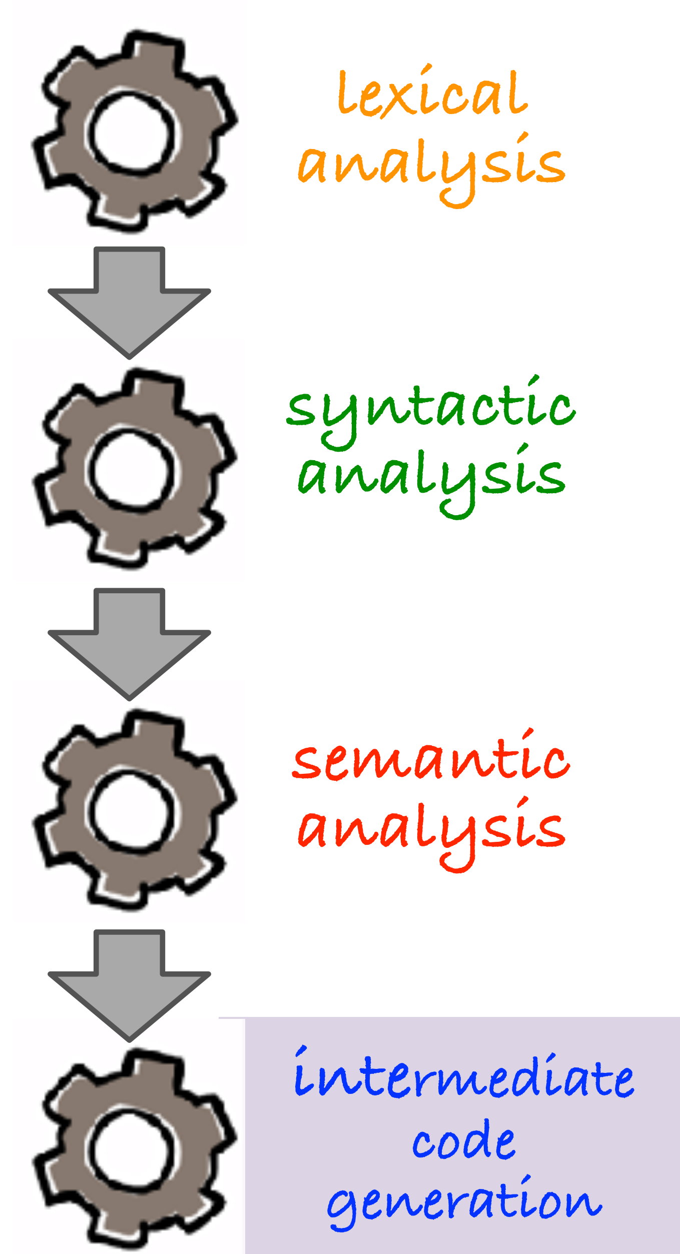
A COMPILER AT WORK & THE JVM INTERPRETER

```
while (y < z) {
    int x = a + b;
    y = y + x;
}
```

```
Loop:  _t1 = y < z
       iffalse _t1 goto Exit
       x = a + b
       y = x+y
       goto Loop
Exit:
```

this bytecode is given in input to the JVM which

* interprets every instruction
* executes it on the machine

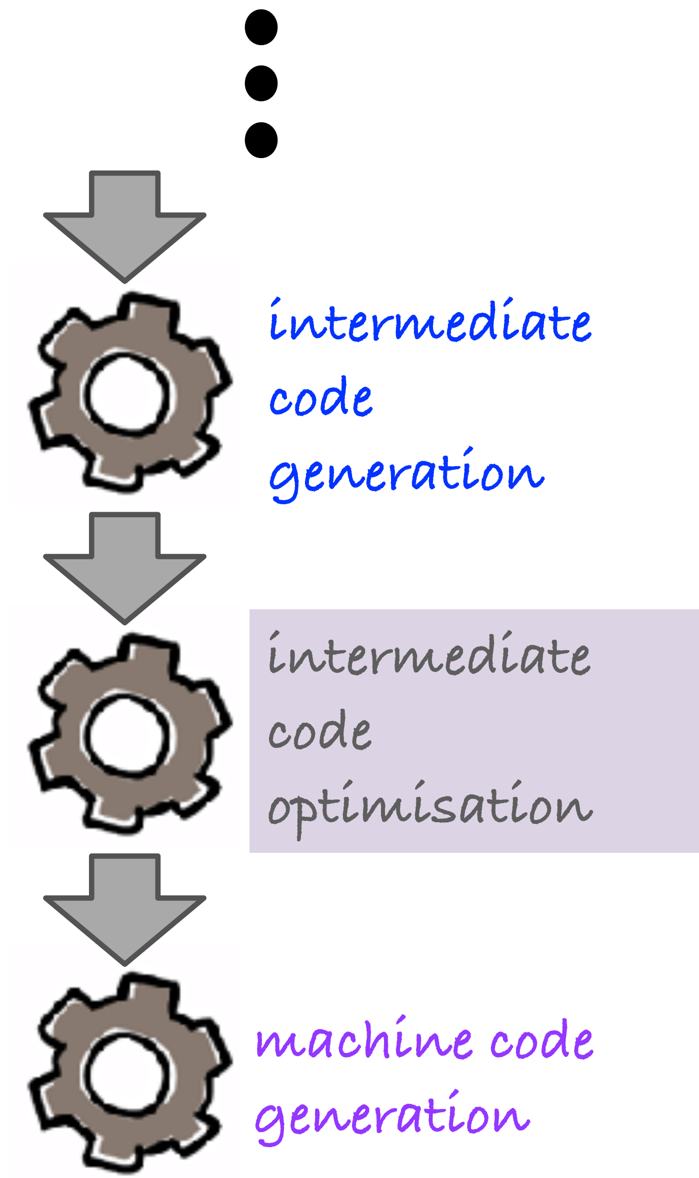


A COMPILER AT WORK (IN c LANGUAGES)

```
while (y < z) {
     int x = a + b;
     y = y + x;
}
```

```
        x = a + b
Loop:   _t1 = y < z
        iffalse _t1 goto Exit
        y = x+y
        goto Loop
Exit:
```

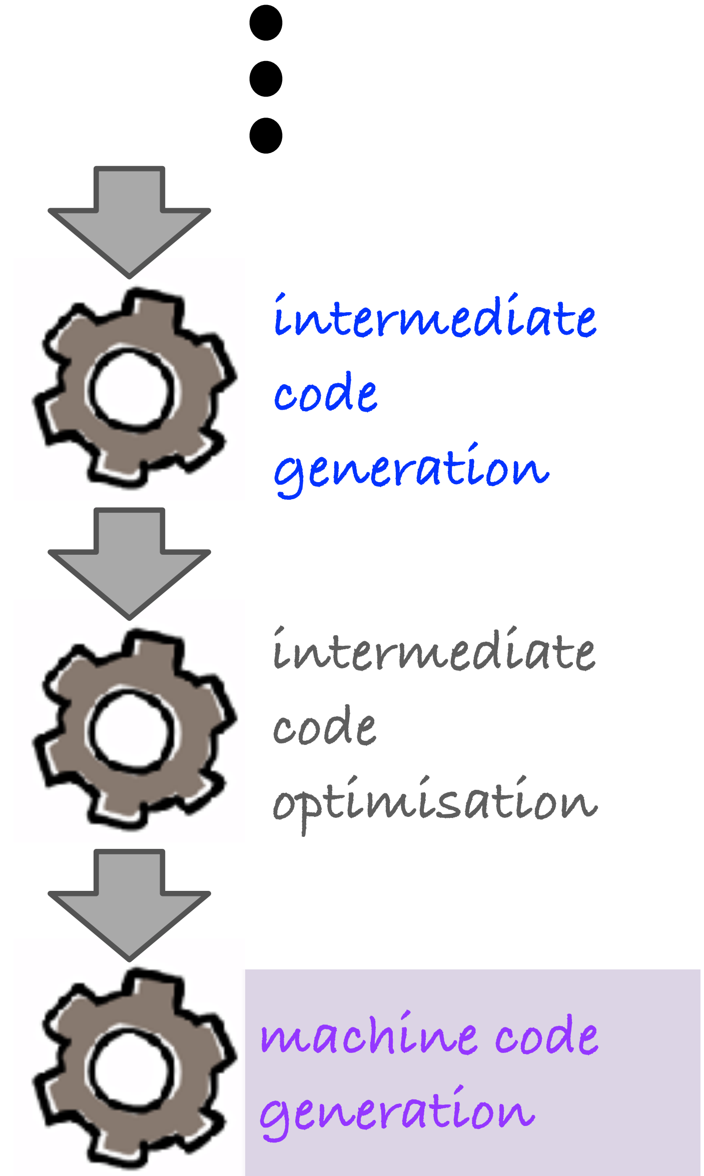
optimizations produce better code



A COMPILER AT WORK (IN c LANGUAGES)

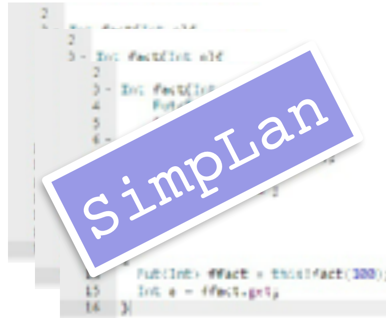
```
while (y < z) {
    int x = a + b;
    y = y + x;
}
```

```
        add $1, $2, $3
Loop:   slt $6, $4, $5
        beq $6, Exit
        add $4, $1, $4
        b Loop
Exit:
```



build the code for your machine

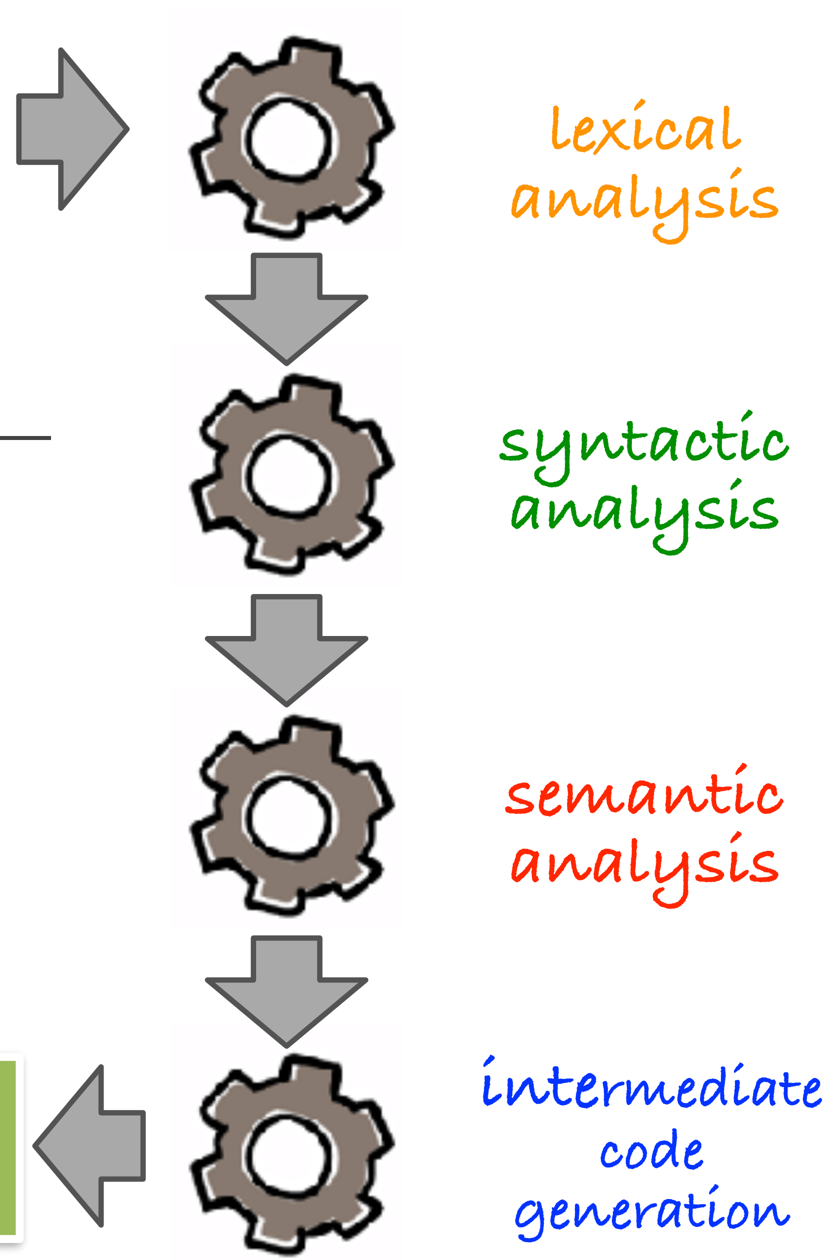
THIS COURSE



we will **build a compiler** for a simple programming language — **`SimpLan`** — and we will build an interpreter for executing **`SimpLan`**

as **team project** you must

- develop a compiler for an extension of **`SimpLan`**
- use the **`ANTLR`** development tool



RECAPS ABOUT GRAMMARS

* see Torben Morgensen: **Basics of Compiler Design**, chapter 3, section 1, 2 and 3

RECAPS ABOUT GRAMMARS

Definition: context-free grammar

A **context-free grammar** is a tuple (**N**, **T**, →, S) where

- **N** is a finite set of **non-terminal symbols**
- **T** is a finite set of **terminal symbols**
- → is a finite set of **productions** of type

  $A \to \alpha_1 \ldots \alpha_n$ with $A \in \mathbf{N}$ and $\alpha_1 \ldots \alpha_n \in \mathbf{N} \cup \mathbf{T}$
- $S \in \mathbf{N}$ is called **initial symbol**

example:

BExp → (BExp)
BExp → Digit
Digit → 0 | 1 | . . . | 9

compact syntax that represents 10 productions

we will always keep the tuple IMPLICIT!

FORMAL NOTIONS

Definition: derivations

Let $\mathcal{G}$ = (**N**, **T**, →, S) be a context-free grammar and γ and δ be sequences of symbols in **N** ∪ **T**. A **one-step derivation** of $\mathcal{G}$ is

$$\gamma A \delta \rightarrow \gamma \alpha_1 \ldots \alpha_n \delta$$

where $A \rightarrow \alpha_1 \ldots \alpha_n \in \rightarrow$

notation: $\rightarrow^*$ (0 or more steps) $\rightarrow^+$ (1 or more steps)

$\gamma \rightarrow^* \delta$ is called **derivation**

Definition: language generated by a context-free grammar

The **language (generated)** by $\mathcal{G}$ = (**N**, **T**, →, S) is the set

$$\mathcal{L}(\mathcal{G}) = \{ \gamma \mid \gamma \in \mathbf{T}^* \text{ e } S \rightarrow^+ \gamma \}$$

T* is the **Kleene closure**: every sequence of symbols in **T**

example: if **T** = {a, b}, **T*** = {ε, a, b, aa, ab, ba, bb, aaa, ...}

DERIVATIONS — EXAMPLES

take the grammar

$$\text{BExp} \rightarrow (\ \text{BExp}\ )$$
$$\text{BExp} \rightarrow \text{Digit}$$
$$\text{Digit} \rightarrow \mathtt{0} \mid \mathtt{1} \mid \ldots \mid \mathtt{9}$$

$$\text{BExp} \rightarrow (\ \text{BExp}\ ) \rightarrow (\ (\ \text{BExp}\ )\ ) \rightarrow (\ (\ \text{Digit}\ )\ ) \rightarrow (\ (\ \mathtt{1}\ )\ )$$

- is a **derivation**
- the **sequence of terminals** ((1)) belongs to the language generated by the grammar
- the sequences ((9)) e ((((1)))) and 3 **belong** to the language, as well — what are the derivations?
- the sequences ((BExp)) and ((((10)))) and ((3) **do not belong to** the language — why?

DERIVATIONS — LEFTMOST AND RIGHTMOST

a **leftmost derivation** is a derivation where the non-terminal symbol that is replaced every time is the **leftmost** one

rightmost/the rightmost one

example Exp → Exp - Exp

Exp → Digit

Digit → 0 | 1 | . . . | 9

I am replacing this Exp

Exp → Exp - Exp → Exp - Exp - Exp

→ Digit - Exp - Exp → 3 - Exp - Exp → 3 - Digit - Exp

→ 3 - 2 - Exp → 3 - 2 - Digit → 3 - 2 - 1

is a **leftmost derivation**

note: Exp → Exp - Exp → Digit - Exp → 3 - Exp

→ 3 - Exp - Exp → 3 - Digit - Exp

→ 3 - 2 - Exp → 3 - 2 - Digit → 3 - 2 - 1

this is also a leftmost derivation!

ANTLR

ANTLR = ANother Tool for Language Recognition

- is a powerful parser generator for reading, processing, executing, or translating structured text or binary files
- it's widely used to build languages, tools, and frameworks
- from a grammar, ANTLR generates a parser that can build and walk parse trees

ANTLR

you need

* `Eclipse/IntelliJ`
* `ANTLR` plugin

in the e-learning website, there is a folder (see Argomento 2) where

* you can find installation informations about **`ANTLR`**
* a simple example of what you can do with **`ANTLR`**

to play, you can use the online tool:

`http://lab.antlr.org/`

ANTLR — AN INITIAL EXAMPLE

grammar file has suffix g4

Arrayofint.g4

the syntax

```
grammar ArrayofInt ;

init : '{' value (',' value)* '}' ;
value : init | INT ;


INT : [0-9]+  ;
```

this is the first line; the name is the same of the file without suffix

these are the parser productions

- nonterminal begins with lowercase letter
- the first production identifies the initial symbol

this is a lexer production

- nonterminal are in capital letter
- rhs are always regular expressions

`ANTLR` — AN INITIAL EXAMPLE

programs:

* `{ 12, 245, 3327}`
* `{ 1, { 1, 2, 3 }, 3 }`

ANTLR — THE ANALYSIS

ANTLR returns the syntax tree

- you may compute the sum of integers
- the number of integers
- the maximal nesting
- …

example: `{ 1, { 1, 2, 3 }, 3 }`

- the sum is 10
- the number of integers is 5
- the maximal nesting is 2

NEXT LECTURE