



1. Introduction

Motivations and General Structure of a Compiler

Andrea Polini, Luca Tesei

Compilers
Master of Science in Computer Science
University of Camerino

WARNING

Slides are distributed to help students in their preparation to the exam. In **no way** they intend to substitute text books. Instead a **thorough study of text books** constitutes the **most wise strategy** to maximise the chances to pass the final exam.

- 1 General Information
- 2 Introduction to Compilers

ToC

- 1 General Information
- 2 Introduction to Compilers

Teacher and Course

- Luca Tesei
 - e-mail: luca.tesei@unicam.it
 - web:
 - General: <http://www.lucatesei.com>
 - Unicom & Office Hours: <http://docenti.unicam.it/...>
- For objectives, description, material and exam see the **wiki**
<http://didattica.cs.unicam.it/...>
- For recorded lectures, notices and other material see the **Google Classroom** page




Course Objective

At the end of the course:

- you will know the most common ways of specifying programming languages and other structured languages
- you will know the basic theory and methodology behind the construction of a compiler
- you will be able to understand the basic issues related to compilers construction
- you will have acquired basic skills to develop a compiler/transformer for a simple language

Study material

- **Reference book:**

-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
Compilatori. Principi, tecniche e strumenti. Ediz. MyLab. Con aggiornamento online, Pearson, 2019.
-  Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
Compilers – Principles, Techniques and Tools, 2nd Ed.
Addison-Wesley, 2007.
-  Terence Parr
The Definitive ANTLR4 Reference
The Pragmatic Programmers, 2012.

- **Further references possibly provided by the teacher**

Final Exam

1. **Project** – dates of delivery scheduled in the ESSE3 system
 - You will be asked to develop a compiler/translator for a simplified language using the ANTLR4 parser generator
2. **Written Test** – dates scheduled in the ESSE3 system
 - The paper will contain exercises that ask to the student to solve problems not solved during classes, or questions on more theoretical aspects. During the lectures we will discuss the solutions of similar exercises
3. **Registration of the Mark** – once the student's project is accepted (mark ≥ 18) **and** the student passed the written test (mark ≥ 18), the final mark is the average
4. Project and Written Test are independent, they do not have to be passed in the same exam session. Each grade lasts for one solar year and is automatically cancelled if the same partial exam is attempted again.

ToC

- 1 General Information
- 2 Introduction to Compilers**

Compilers vs. Interpreters

Two approaches to permit the execution of a program, written using an high level language, on a physical machine:

- Compilers: use of a program that can read a program in one language (**source**) and translate it into an **equivalent** program in another language (**target**)
- Interpreters: use of a program that takes in input the program and data and run the program on the data without the need to make an explicit translation into the machine code

Java?

Compilers vs. Interpreters

Two approaches to permit the execution of a program, written using an high level language, on a physical machine:

- Compilers: use of a program that can read a program in one language (**source**) and translate it into an **equivalent** program in another language (**target**)
- Interpreters: use of a program that takes in input the program and data and run the program on the data without the need to make an explicit translation into the machine code

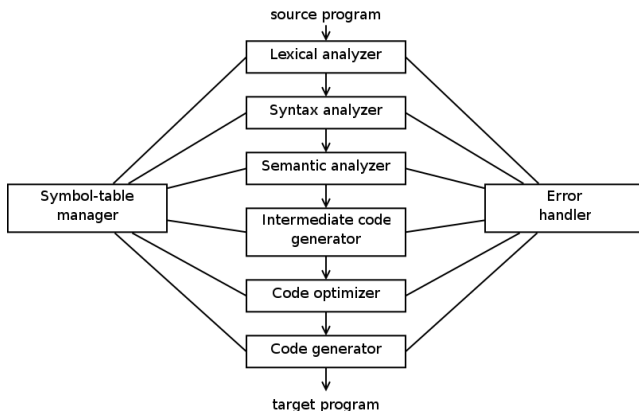
Java?

Birth

- 1954 – IBM develops the 704 (software cost > hardware cost)
- 1954 - 1957 – FORTRAN I (FORMula TRANslating system) is developed (In 1958 50% of code is written in FORTRAN)
 - The definition of the first compiler led to an enormous body of theoretical work

Compiler construction is a complex engineering activity (practice) which need to be based on well defined theoretical background (theory)

Structure of a Compiler



Two main parts:

Analysis(front end) and Synthesis (back end)

Lexical analysis

After having defined the alphabet to be used, the first things to do is to recognize words

This is a sentence

The lexical analysis divides the program text into words and produce a sequence of tokens ($\langle token-name, attribute-value \rangle$)

```
position = initial + rate * 60
```

Lexical analysis

After having defined the alphabet to be used, the first things to do is to **recognize words**

This is a sentence

The lexical analysis divides the program text into words and produce a sequence of **tokens** ($\langle token-name, attribute-value \rangle$)

```
position = initial + rate * 60
```

Syntax Analysis

After having understood the words we need to **understand the sentence structure**. Not so much different from the syntax of what we do for understanding natural languages

This line includes a long sentence

Semantic Analysis

Once the structure of the sentence is clear we need to understand the meaning:

- Humans can manage quite well this activity, **the same is not so true for machines**

Examples:

- Jack said Jerry left his assignment at home
- Jack said Jack left his assignment at home?
- Jack left her assignment at home
- Compilers perform many semantic checks besides variable bindings and type checking

Intermediate Code Generation

An intermediate representation that is easy to produce and easy to translate is useful.

Typically based on a *three-address code* form i.e. assembly like instructions with three operands per instruction:

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimization

Not so important for natural language! It is the most complex and effort prone activity in the construction of modern compilers

- The compiler modifies the program so that it
 - runs faster
 - uses less memory
 - uses less power
 - makes less database accesses
 - uses less bandwidth
 - ...

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Optimisations can be **machine dependent** or **machine independent**

Code generation

Takes as input the intermediate representation of the source program and produces assembly code to be run on the target machine

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

Judicious **assignment of registers** to hold variables is a crucial aspect, as well as the **memory management**, which is a relevant aspect also for the intermediate code generation phase

Proportions of the various phases
changed from the pioneering era

Code generation

Takes as input the intermediate representation of the source program and produces assembly code to be run on the target machine

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

Judicious **assignment of registers** to hold variables is a crucial aspect, as well as the **memory management**, which is a relevant aspect also for the intermediate code generation phase

Proportions of the various phases
changed from the pioneering era

General remarks

- New computer architectures need new compilers
 - **parallelism** (*instruction-level, processor-level*)
 - **memory hierarchies**
- New linguistic constructions ask for the development of new algorithms and new data structure to translate the code
- Code optimisation faces many undecidable problems, thus theory alone is not enough and we need heuristics, good engineers, and good programmers

General remarks

- New computer architectures need new compilers
 - **parallelism** (*instruction-level, processor-level*)
 - **memory hierarchies**
- New linguistic constructions ask for the development of new algorithms and new data structure to translate the code
- Code optimisation faces many undecidable problems, thus theory alone is not enough and we need heuristics, good engineers, and good programmers

General remarks

- New computer architectures need new compilers
 - **parallelism** (*instruction-level, processor-level*)
 - **memory hierarchies**
- New linguistic constructions ask for the development of new algorithms and new data structure to translate the code
- Code optimisation faces many **undecidable** problems, thus theory alone is not enough and we need heuristics, good engineers, and good programmers

Interesting Questions?

- Why are there so many programming languages?
 - Application domains have distinctive needs (scientific computing, business applications, system programming, etc.)
- Why are there new programming languages?
 - Introducing changes in a widely used language is complex
 - Cost of training programmers is the dominant cost for a programming language
 - productivity > training cost?
- What is a good programming language?
 - No definitive metric exists
 - Is it the one programmers use?

Interesting Questions?

- Why are there so many programming languages?
 - Application domains have distinctive needs (scientific computing, business applications, system programming, etc.)
- Why are there new programming languages?
 - Introducing changes in a widely used language is complex
 - Cost of training programmers is the dominant cost for a programming language
 - productivity > training cost?
- What is a good programming language?
 - No definitive metric exists
 - Is it the one programmers use?

Interesting Questions?

- Why are there so many programming languages?
 - Application domains have distinctive needs (scientific computing, business applications, system programming, etc.)
- Why are there new programming languages?
 - Introducing changes in a widely used language is complex
 - Cost of training programmers is the dominant cost for a programming language
 - productivity > training cost?
- What is a good programming language?
 - No definitive metric exists
 - Is it the one programmers use?