System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block C Compiler Construction
1-2. Introduction to the area

Eugene Zouev

Why the Course?

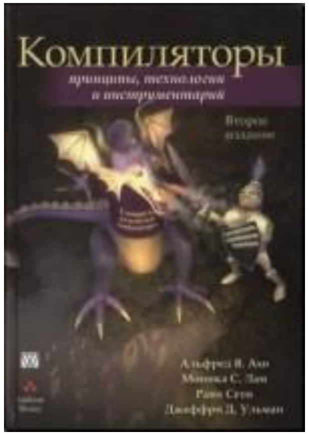
- The area of compiler construction is one of three cornerstones of Computer Science (two others are databases & operating systems).
- The area includes many fundamental results: formal grammar theory, type theory, graph theory, algorithm theory and many others.
- The similar courses are given in universities all over the world.
- Key topics of the course is in the Outline & Syllabus.
- It's the extremely interesting area ☺.

Who Is This Guy? ☺

- **Eugene Zouev**
- Have been working at Moscow Univ., Swiss Fed Inst of Technology (ETH Zürich), EPFL (Lausanne); PhD (1999, Moscow Univ).
- Prof. interests: compiler construction, language design, programming languages' semantics.
- The author of the 1st Russian **C++** compiler (Interstron Ltd., 1999-2000).
- **Zonnon** language implementation for .NET & Visual Studio (ETHZ, 2005).
- **Swift** language prototype compiler for Tizen & Android (Samsung Electronics, 2015).
- Books: «Редкая профессия», ДМК Пресс, Москва 2014.

References (1)

- Alfred V.Aho, Monica S.Lam, Ravi Sethi, Jeffrey D. Ullman, **Compilers. Principles, Techniques, & Tools**, Second Edition, Addison-Wesley, 2007, ISBN 0-321-48681-1. (“Dragon book”)



  Russian translation:
  Ахо, Альфред В., **Лам**, Моника С., **Сети**, Рави, **Ульман**, Джеффри Д. **Компиляторы: принципы, технологии и инструментарий**, 2-е изд.: Пер. с англ.- М.: ООО «И.Д.Вильямс», 2008.- 1184 с.: илл. ISBN 978-5-8459-1350-4.

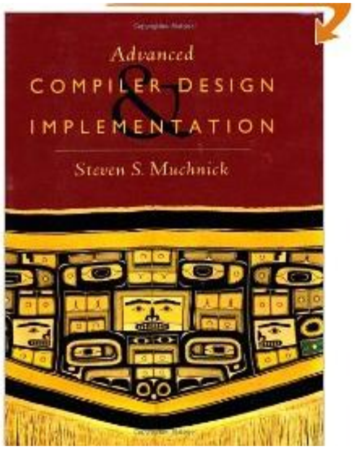
References (2)

- N.Wirth, **Compiler Construction**, Addison-Wesley, 1996, ISBN 0-201-40353-6; http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf, 2005.

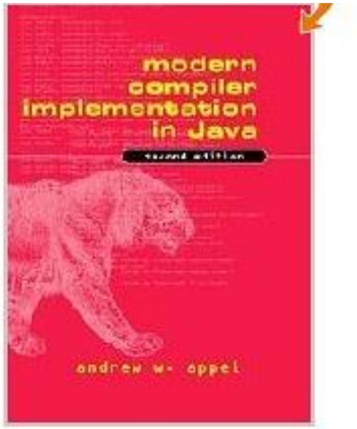
  Russian translation:
  Никлаус Вирт, **Построение компиляторов**,
  ДМК-Пресс, 2010,
  ISBN 978-5-94074-585-3.

References (3)

- S.Muchnik.
  Advanced Compiler Design and Implementation
  1997, ISBN 1-55860-320-4.

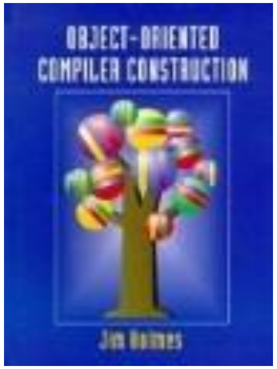


- Andrew Appel
  Modern Compiler Implementation in {Java, C, ML}
  1998



References (4)

- Jim Holmes
  Object-Oriented Compiler Construction
  1994



Textbooks: Problems (1)

- A lot of theory but a few actual examples (even if there are many "exercises").
- Many books in compilers (except some ☺) are not *practically-targeted*.
- The books consider **artificial**, "model" languages: small, simple, regular, "academic". Real languages are much more complicated and typically have (very) bad design – hard to implement.

  Typically, books do not discuss how **real language features** get implemented in compilers.

Textbooks: Problems (2)

- Language semantics is the most important part of any language (syntax is just “decoration”), and the major part of any compiler deals with semantics, not with syntax. However, textbooks consider syntactic aspects of languages and compilers very carefully, in contrast with semantic implementation issues…
- Some aspects are not presented in those books **at all**: e.g., development tools, error recovering, garbage collection, language-specific optimizations etc.

Extra Information Sources

- Magazine papers, PhD theses, conference presentations.
- Source codes of real compilers.
- International standards for PLs.
- Some extra books/papers.

Prerequisits

- Some knowledge of one or two programming languages is highly recommended (the more the better ☺)
  - C, **C++**, **C#**, **Java**, Pascal, Python, …
- The preference is Object-Oriented paradigm.
- Some experience in working with modern IDEs is also needed.
  - Visual Studio, Eclipse, IDEA, NetBeans, …

The Structure of the Course

- **The schedule**: The lecture + the seminar each week (Thursdays, 14.00-15.30, 15.40-17.10).
- **Assignments**: for individuals and/or for a team of two (for two weeks typically).
- **Projects**: for teams of ~four (second part of the course).
- **Presentations** (treated as assignments).
- The final **grade**: assignments 30%, project 70%.

Compilation: The Goal & The Task

- **The Goal**:
  To overcome the «semantic gap» between human’s way of thinking and computer solving task defined by a human.
- **The Task**:
  To transform a human-written and human-friendly ☺ program to a form which can be executed by a computer.

Semantic Gaps

Application domain

Train traffic management system: Trains, velocity, distance, railways, switches, train stations, conflict resolution, time schedule etc.

Simulation of the Universe evolution: Planet movement rules, planet systems, stars classification, star activity etc.

…

Computer program/ programming language

- Variable, array, …
- Function, operator, procedure
- Control structures
- Data types
- Module, class, interface
- …

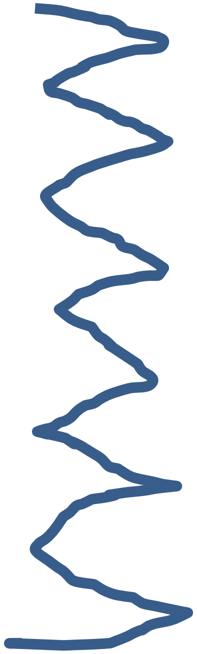
Computer hardware

- Memory cell
- Memory address
- Register
- Instruction, instruction set
- …

Semantic Gap

Computer program/ programming language

- Variable, array, …
- Function, operator, procedure
- Control structures
- Data types
- Module, class, interface
- …



Computer hardware

- Memory cell
- Memory address
- Register
- Instruction, instruction set
- …

So, the main compiler task is:

- To **map** human-written & human-friendly program to computer hardware – to solve a problem

OR:

- **To overcome the semantic gap**.

Machine Code & Assembly Language

Off-topic

Mnemonic notation:

ADD 5 7

Binary code of the instruction:

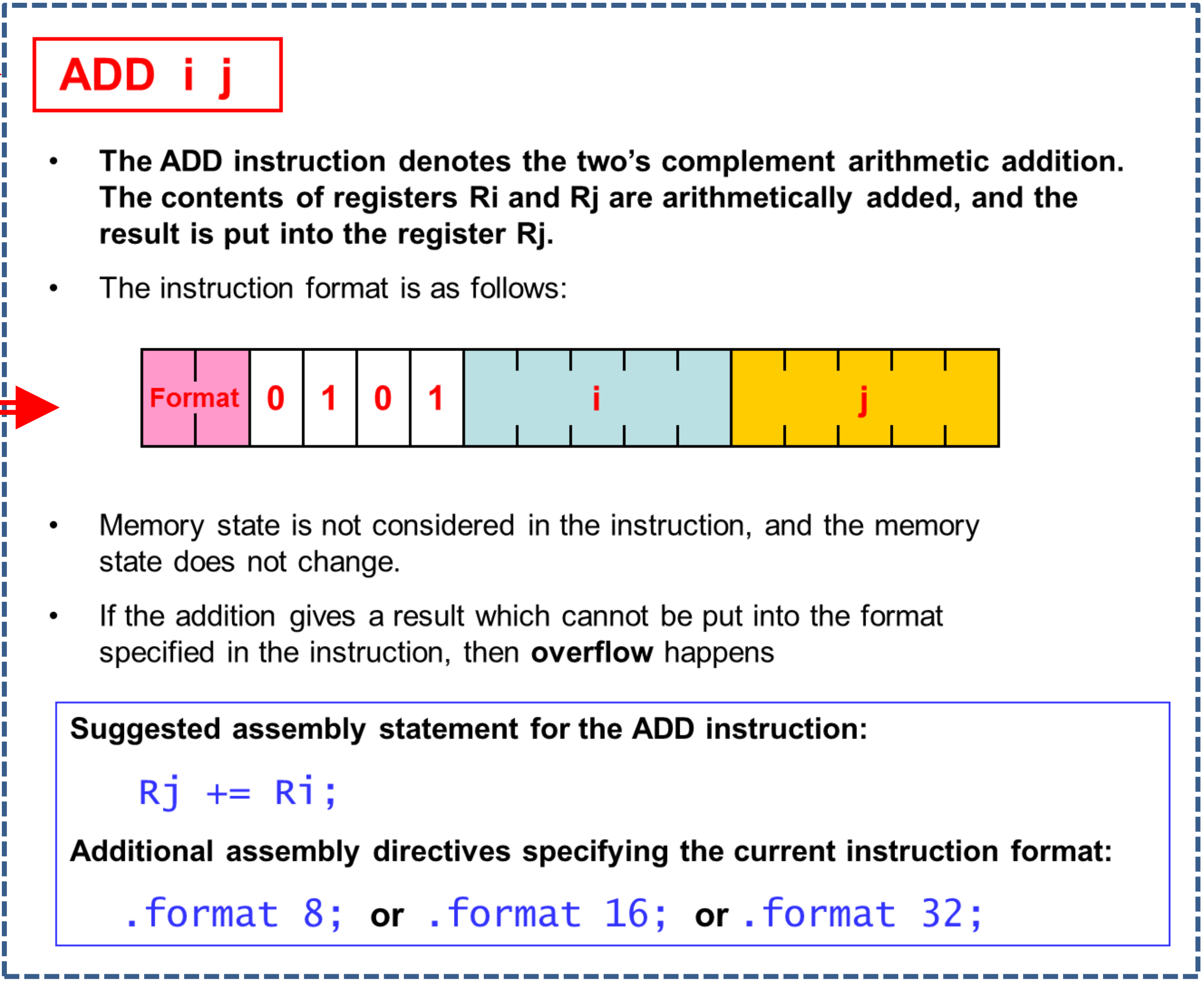
1101010010100111

Hexadecimal code of the instruction:

D4A7

Assembler code:

```
.format 32
R5 += R7
```



ADD i j

- **The ADD instruction denotes the two's complement arithmetic addition. The contents of registers Ri and Rj are arithmetically added, and the result is put into the register Rj.**
- The instruction format is as follows:

Format	0	1	0	1	i	j

- Memory state is not considered in the instruction, and the memory state does not change.
- If the addition gives a result which cannot be put into the format specified in the instruction, then **overflow** happens

Suggested assembly statement for the ADD instruction:

```
Rj += Ri;
```

Additional assembly directives specifying the current instruction format:

`.format 8;` **or** `.format 16;` **or** `.format 32;`

Compilers: Some History (1)

Off-topic

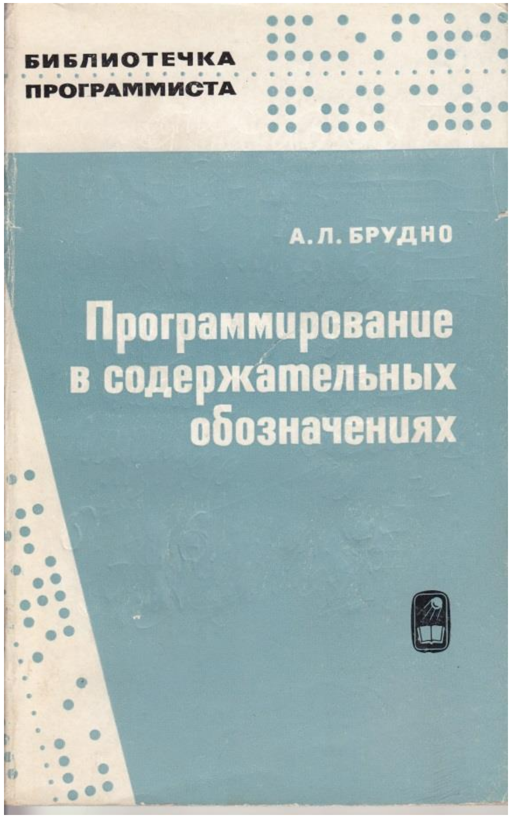
~~Автокод~~ -> Ассемблер (Язык ассемблера)
~~Автокод 1:1~~ Assembler (Assembly language)

~~Программирующая программа~~

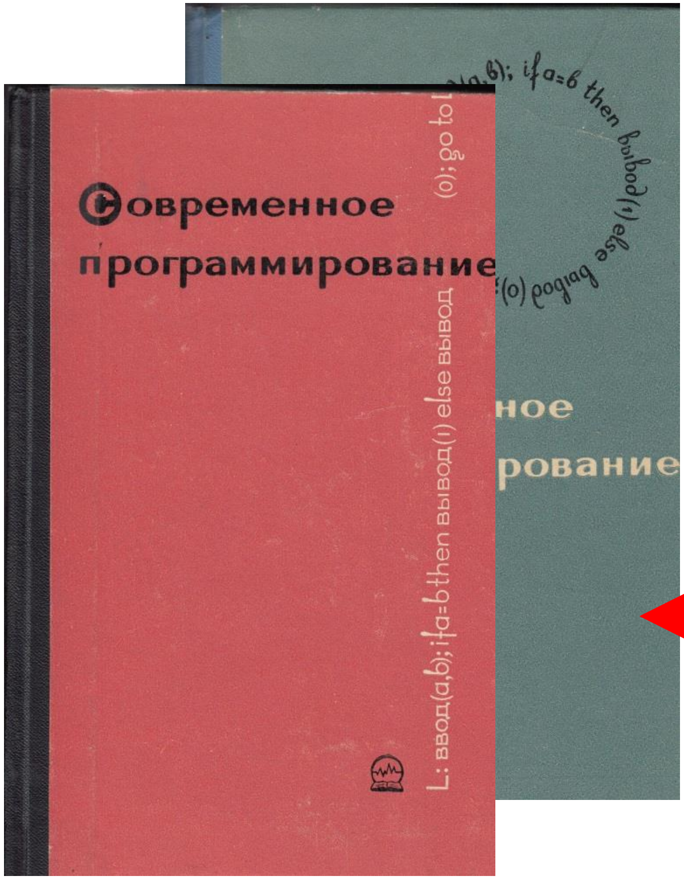
-> Транслятор -> Компилятор
Translator Compiler

Compilers: Some History (2)

Off-topic



1968


Современное программирование
L: ввод(a,b); if a=b then вывод(1) else вывод(0); go to L
(a,b); if a=b then вывод(1) else вывод(0)
ное
рование

1966-1967

Appendix:
More than **300** translators for Fortran, Cobol, Jovial etc. for various computers

Compilers: Now and Then

Off-topic

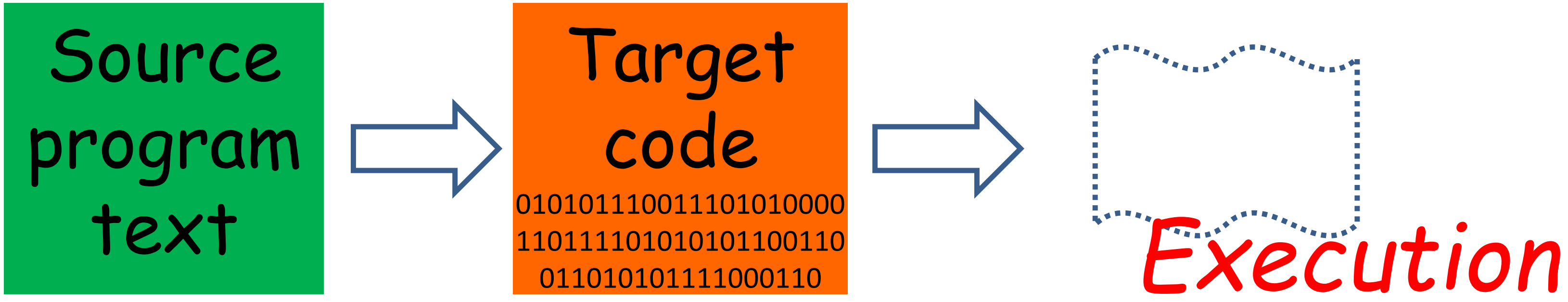
The Past:

- Compiler construction **was** the high level of the programming art, the sign of the top command and/or individual mastery.
- Compilers were among the most complicated software components: one of three cornerstones of computer science (two others are Operating Systems and Databases).

The Present:

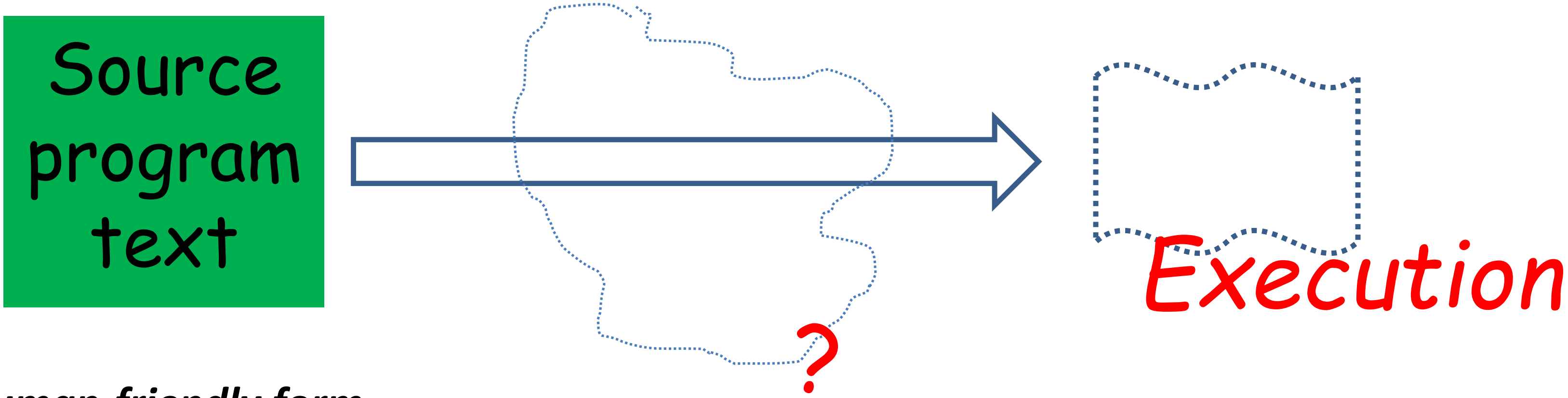
- Compiler construction **is** a sort of usual programming job. The average programming qualification became much higher; to make a compiler is under the force of many.
- Software in general becomes much more complicated; for example, implementation of an Internet browser or an XSLT processor is not an easier job than to write a compiler.
- Compiler construction field is now very monopolized. It's hard for a new compiler to compete with existing ones – even if it's better.

Compilation vs Interpretation



Human-friendly form

Machine form

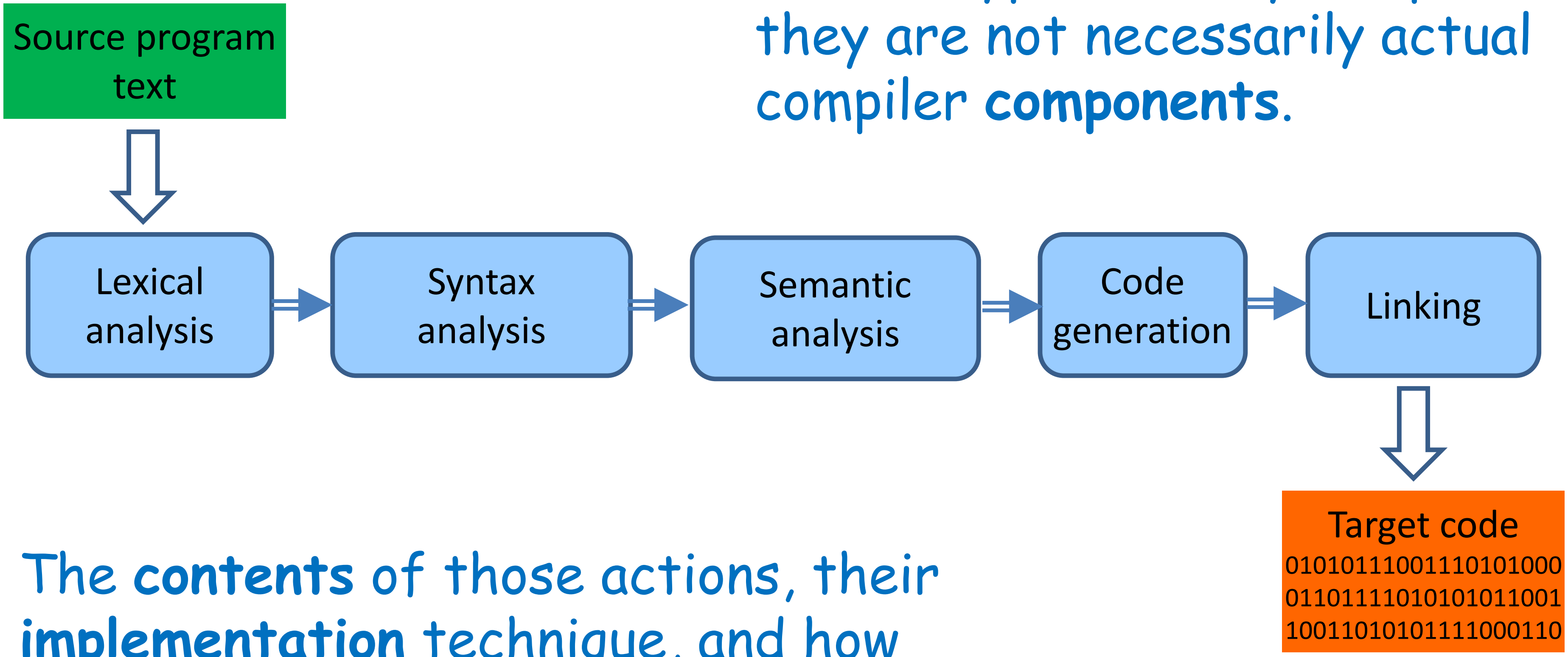


Human-friendly form

Compilation: An Ideal Picture

A program written by a human (or by another program)

Blue squares just denote some **actions** typical to any compiler; they are not necessarily actual compiler **components**.



Source program text

Lexical analysis

Syntax analysis

Semantic analysis

Code generation

Linking

Target code

01010111001110101000
01101111010101011001
10011010101111000110

The **contents** of those actions, their **implementation** technique, and how they **interact** with other actions - is just the subject of the course.

A program binary image suitable for immediate execution by a machine

Compilation Phases: C++

ISO/IEC 14882:2011(E) Programming Languages -- C++, Third Edition (Excerpt from the section 2.2).

Initial source text analysis

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set…
2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines…
3. The source file is decomposed into **preprocessing tokens**…

Preprocessing

4. **Preprocessing directives are executed**, macro invocations **are expanded**, and _Pragma unary operator expressions are executed...

Literal processing

5. Each source character set member in a character literal or a string literal, as well as each escape sequence and universal-character-name in a character literal or a non-raw string literal**, is converted** to the corresponding member of the execution character set…
6. Adjacent string literal tokens **are concatenated**.

Translation

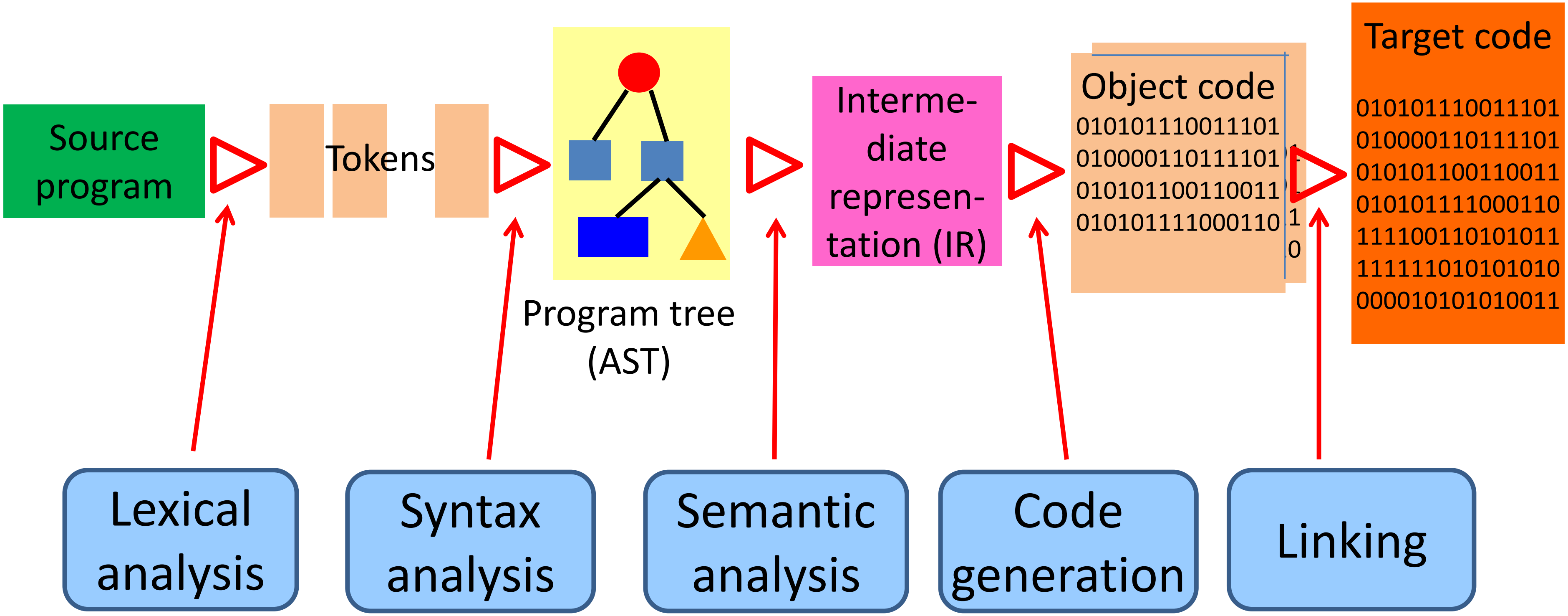
7. Each preprocessing token is converted into a **token**.
8. The resulting tokens are **syntactically and semantically analyzed** and translated as a **translation unit**. Each translated translation unit is examined to produce a list of required instantiations… The definitions of the required templates are located. All the required instantiations are performed to produce **instantiation units**.

Linking

9. All external entity references are **resolved**. Library components are **linked** to satisfy external references to entities not defined in the current translation. All such translator output is collected into **a program image** which contains information needed for execution in its execution environment.

Compilation Data Structures

This is a different, **data-centric** view at the compilation process. Compilation is **a sequence of transformations** of a source program.



Source program

Tokens

Program tree (AST)

Interme-diate represen-tation (IR)

Object code
010101110011101
010000110111101
010101100110011
010101111000110

Target code
010101110011101
010000110111101
010101100110011
010101111000110
111100110101011
111111010101010
000010101010011

Lexical analysis

Syntax analysis

Semantic analysis

Code generation

Linking

(c) Copyright 2018, E.Zouev

The Ideal Picture: Modifications (1)

- There may be **several** source units. They can be represented not only as disk files, but may have arbitrary nature (generally, a stream of bytes).
- Lexical analysis may be implemented as a **separate "pass"**, or as a component invoking **"by demand"**, or…
- Lexical analysis can comprise **several "passes"**. Example: *C/C++*: first preprocessing (macro expansion), and then "true" lexical analysis.
- Lexical & syntax analyses can work either **separately** or **simultaneously** (interacting with each other). Will be examples for *C++* later.

The Ideal Picture: Modifications (2)

- Syntax analysis can be implemented as a single "pass", or as a **sequence of "passes"** (Java, Scala are examples), and/or can run together with semantic analysis.
- Syntax & semantic analyses can be implemented **as the single stage** (only for simple languages like Pascal or Oberon).
- Typically, semantic analysis can include **several "passes"**, or tree traverses (examples for C# and Scala follow); the AST gets modified while each "pass".

The Ideal Picture: Modifications (3)

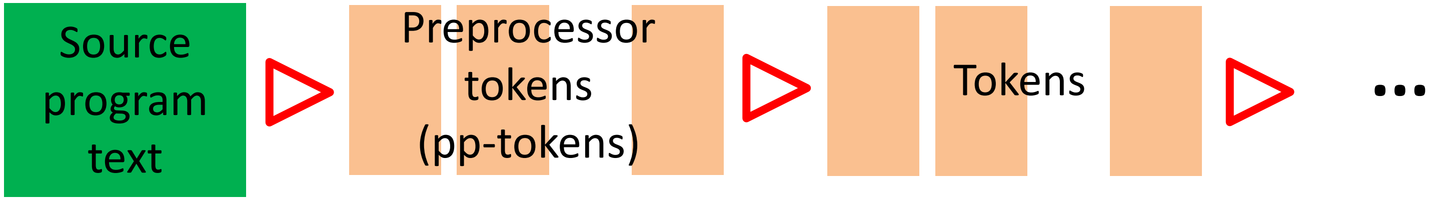
- Program tree is build for the complete program or sequentially, for its parts (functions, classes).
- Intermediate representation: either specially designed *compile-time structures*, (e.g. Control Flow Graph – CFG) or *the program tree itself*, or some (other) language.
  Examples: lower-level language C--; standard C language as intermediate program representation.
- Can be several intermediate languages/structures: see Muchnik; gcc: GENERIC, GIMPLE, RTL.

The Ideal Picture: Modifications (4)

- On each compilation phase (from lexical analysis to code generation) external sources can be added to the program: either in the textual form (include files in C/C++), or as precompiled components (libraries).
- Some languages assume/require **linking** stage which is often considered as a standalone phase.
- Code generation phase often (typically) includes some **optimizing sub-phases**.

Compilation Stages: Lexical Analysis

Components: preprocessor, scanner



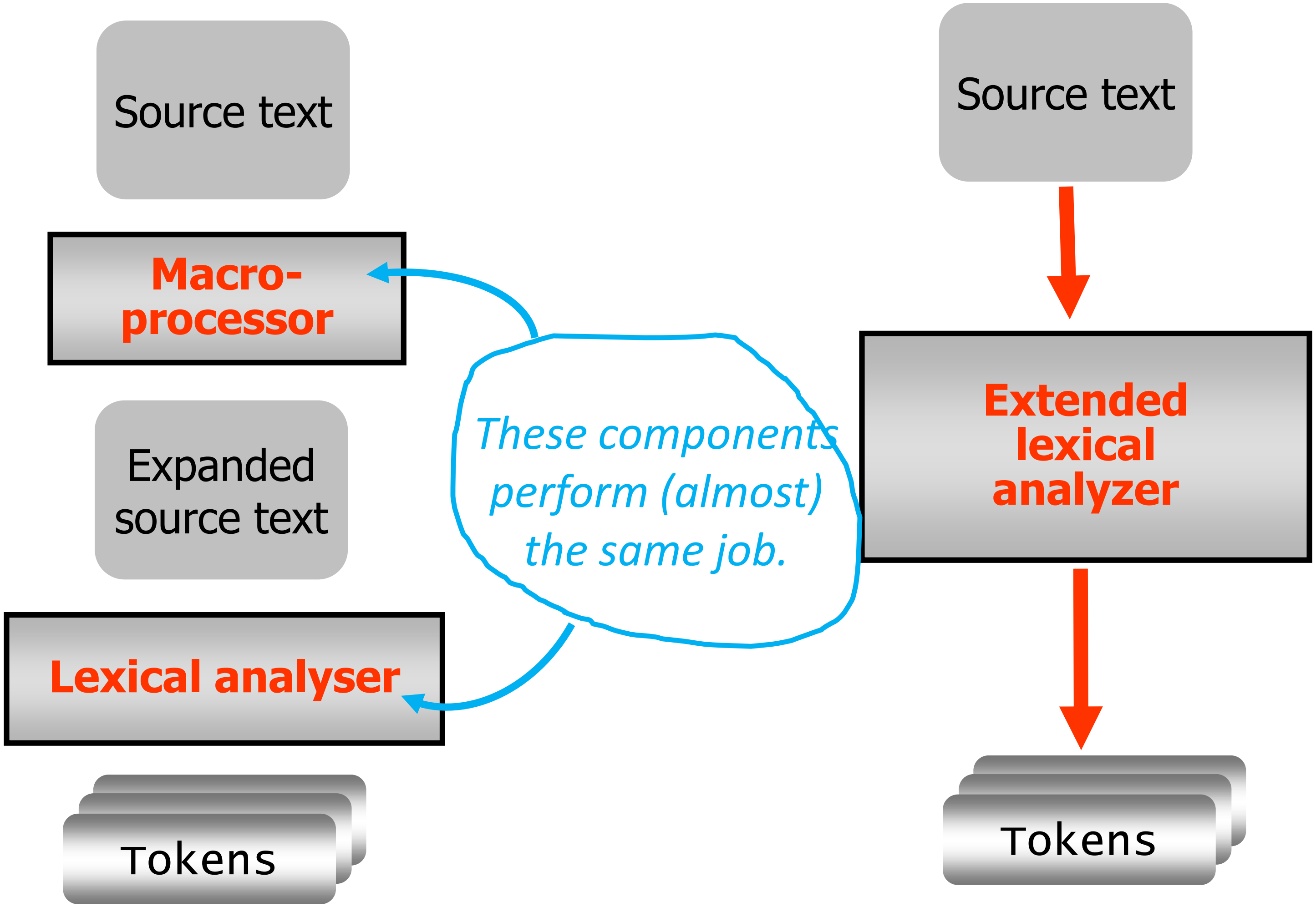
Source program text ▷ Preprocessor tokens (pp-tokens) ▷ Tokens ▷ …

- **Token** is the minimal element of the language alphabet. Examples are operator signs, delimiters, identifiers, keywords, literals.
- **Token representations** in the compiler can be either very simple (e.g., coded by integer values) or be a structure with a set of attributes (see lecture 3 for details).
- How lexical analysis **interacts** with other compiler components: either passing the current token "on demand" or buffering tokens and traversing those buffers (with returns) – see lectures 3-5.

Compilation Stages: Lexical Analysis

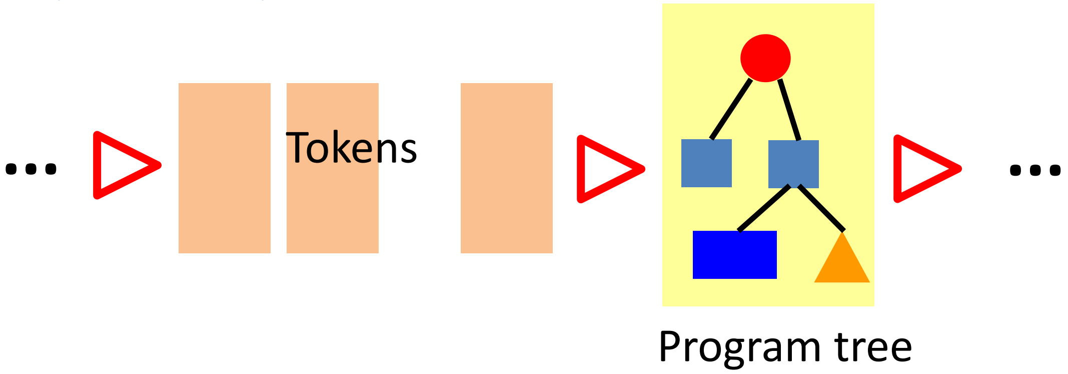
Implementation alternatives:

- Separately OR together?



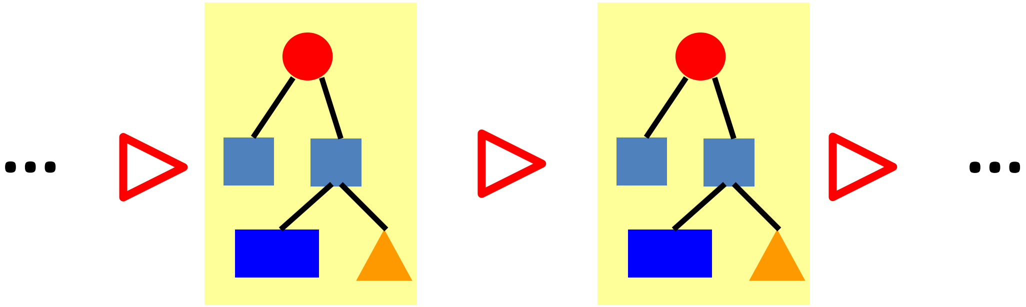
Compilation Stages: Syntax Analysis

Component: parser



- **Syntax analysis**: Checks the correctness of the syntactical structure of the program in accordance with the source language grammar.
- **The result of the analysis**: An internal **tree-like** representation of the source program where nodes and sub-trees represent structural elements of the source.
- **Parser implementation**: Either hand-written (recursive descent parsing), or automatic parser generation based on a formal language grammar.
- Syntax analysis can be combined with semantic analysis.

Compilation Stages: Semantic Analysis

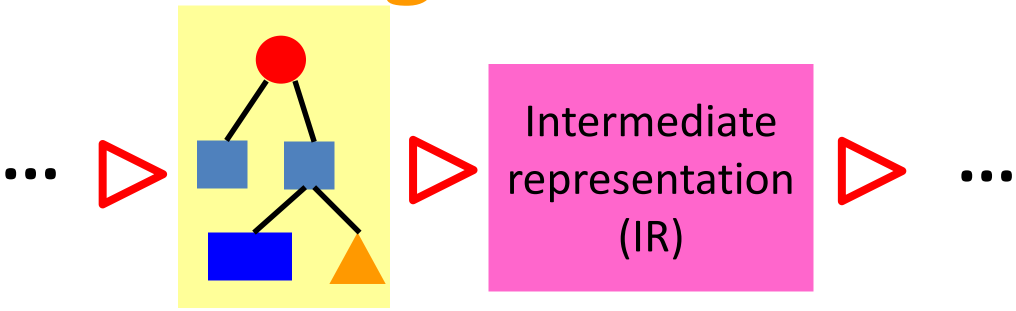


Program tree

Modified program tree

- **Semantic analysis**: Checks semantic correctness of the program against source language definition.
- This is very crucial and the most complicated part of every compiler; the stage typically cannot be automated.
- The result of the analysis: **modified program tree**, perhaps with some extra node attributes («Annotated AST», AAST).
- The semantic analysis is implemented either as a separate component ("analyzer", "validator") or as a "distributed" stage: it can start from the early compilation stages (even from the lexical analysis).

Compilation Stages: IR Generation

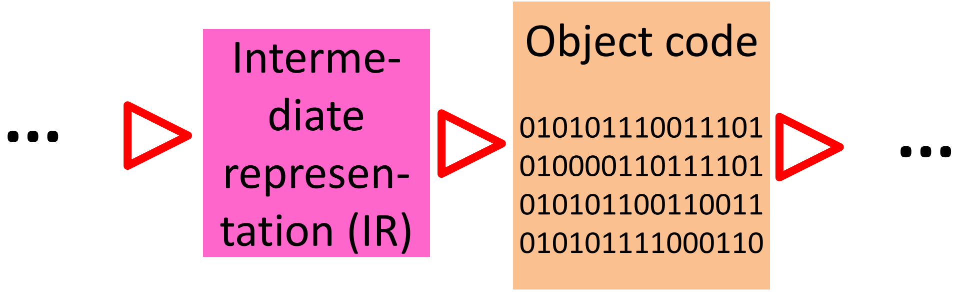


… ▷ ▷ Intermediate representation (IR) ▷ …

Program tree

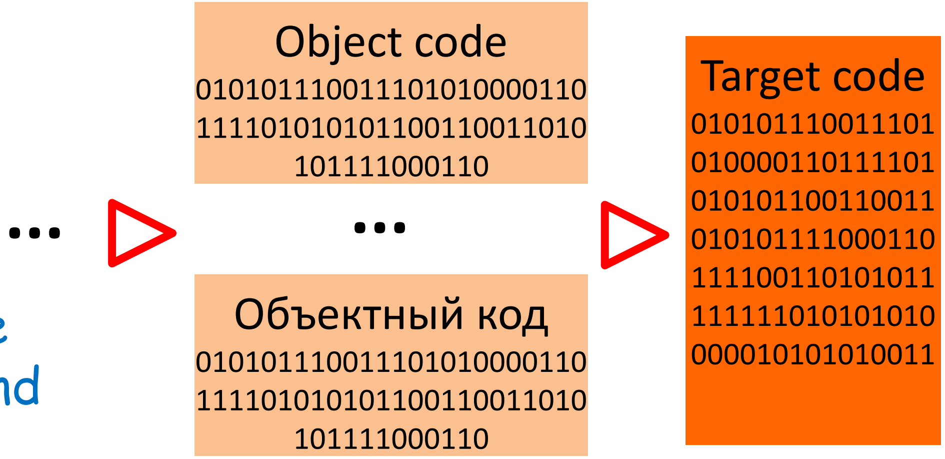
- **IR generation** produces a low-level program representation – either for optimization purposes or as a common base for multi-target code generation.
- Often **the last tree traversing** performs IR generation.
- IR examples:
  - C language ☺; a specially designed low-level language (C--);
  - Assembler language or “generic” assembler language (LLVM bitcode);
  - A proprietary format (e.g., ICode for Scala);
  - Control Flow Graph, CFG.
- Can be dropped for simple cases; (final) code generation can be done directly using AST.

Compilation Stages: Object Code Generation



- **Code generation** performs construction of lowest-level machine instructions.
- "Object code" notion is typical to languages like C/C++ where the concept of **separate compilation** is supported. There is no "object code" & linking stage in languages like Java, C#, Oberon.
- Usually, object code is true machine code where addresses are relative (and therefore need to be replaced for real ones), or the code with unresolved external references (which are the subject for resolving).
- Resolving is preformed by **linkage editor** (linker).

Compilation Stages: Object Code Linking



...

Object code
0101011100111010100001101111010101011001100110101011110001 10

Объектный код
01010111001110101000011011110101010110011001101010111100 0110

...

Target code
010101110011101
010000110111101
010101100110011
010101111000110
111100110101011
111111010101010
000010101010011

- Component: linker, **linkage editor**.
- The linker combines object code parts in the single code image and resolves external references.
- Conceptually this stage is a part of compilation process (see Section 2.2 of the ISO C++ Standard); however, in most cases it is implemented as a standalone component.
- The result of the stage: the sequence of machine instructions with resolved references, ready for execution (or with partially resolved references, ready for further linking).
- For languages that do not support separate compilation, the stage is omitted: «object code» from the previous stage is ready for execution.

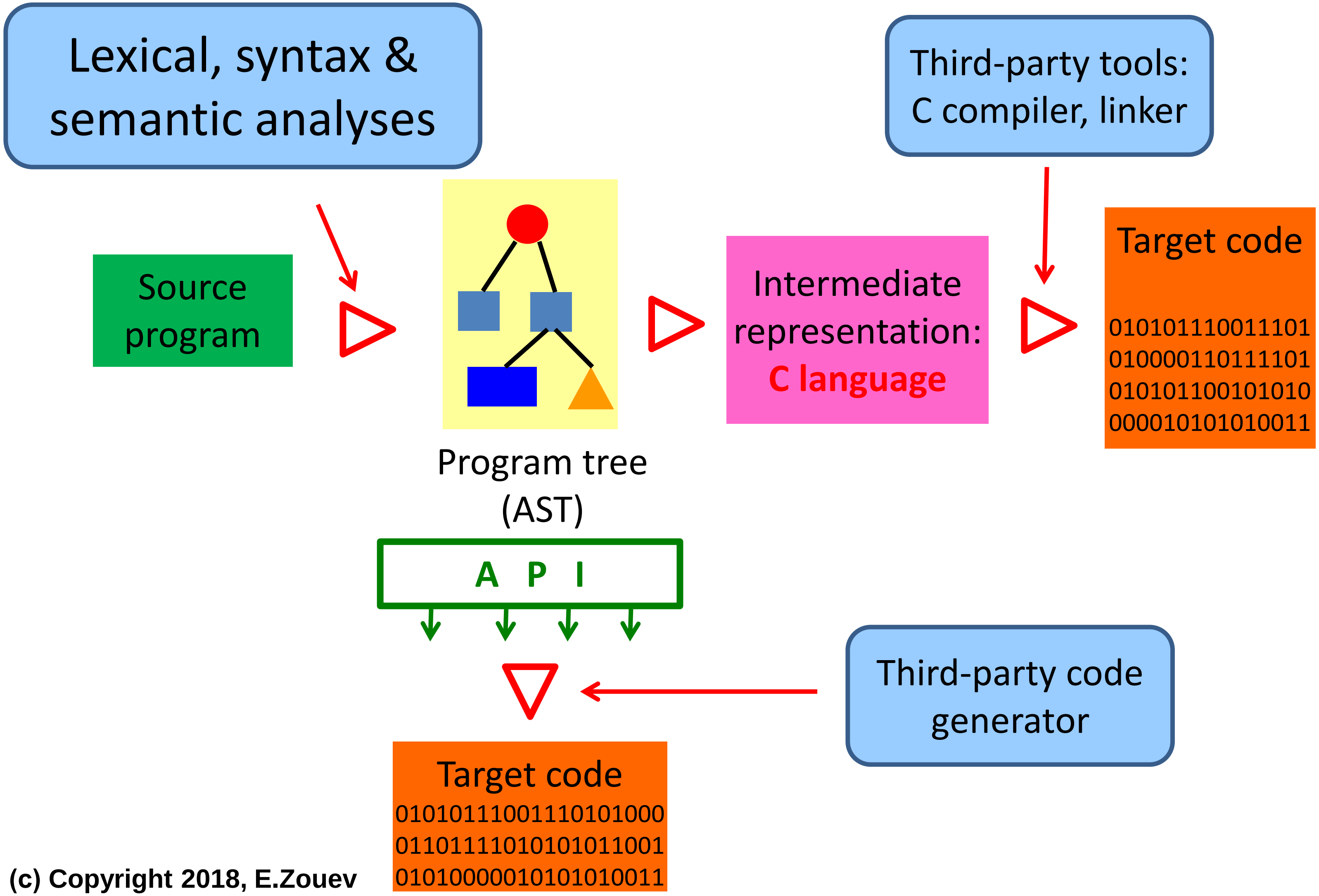
Compilation stages: C# compiler

1. Primary lexical analysis (the result is a stream of tokens).
2. High-level syntax analysis (method bodies are not processed but get stored).
3. Declaration processing.
4. Two passes checking cyclic type & generic dependencies.
5. Two passes for some semantic checks and constant fields processing.
6. Lexical analysis (!) of method bodies, constructors, properties and indexers.
7. Type analysis in method bodies.

Subsequent stages are actually AST tree “walkers” modifying the AST.

- Loop conversion: replacing loops for goto’s & labels.
- **Eight** passes looking for various types of errors.
- **Fourteen** (!) passes for optimization and simplifying AST.
- **Four** passes generating MSIL-код.

Compilation Phases & Structures: Edison Design Group C++ Front End



Lexical, syntax & semantic analyses

Third-party tools: C compiler, linker

Source program

Program tree (AST)

Intermediate representation: **C language**

Target code

010101110011101
010000110111101
010101100101010
000010101010011

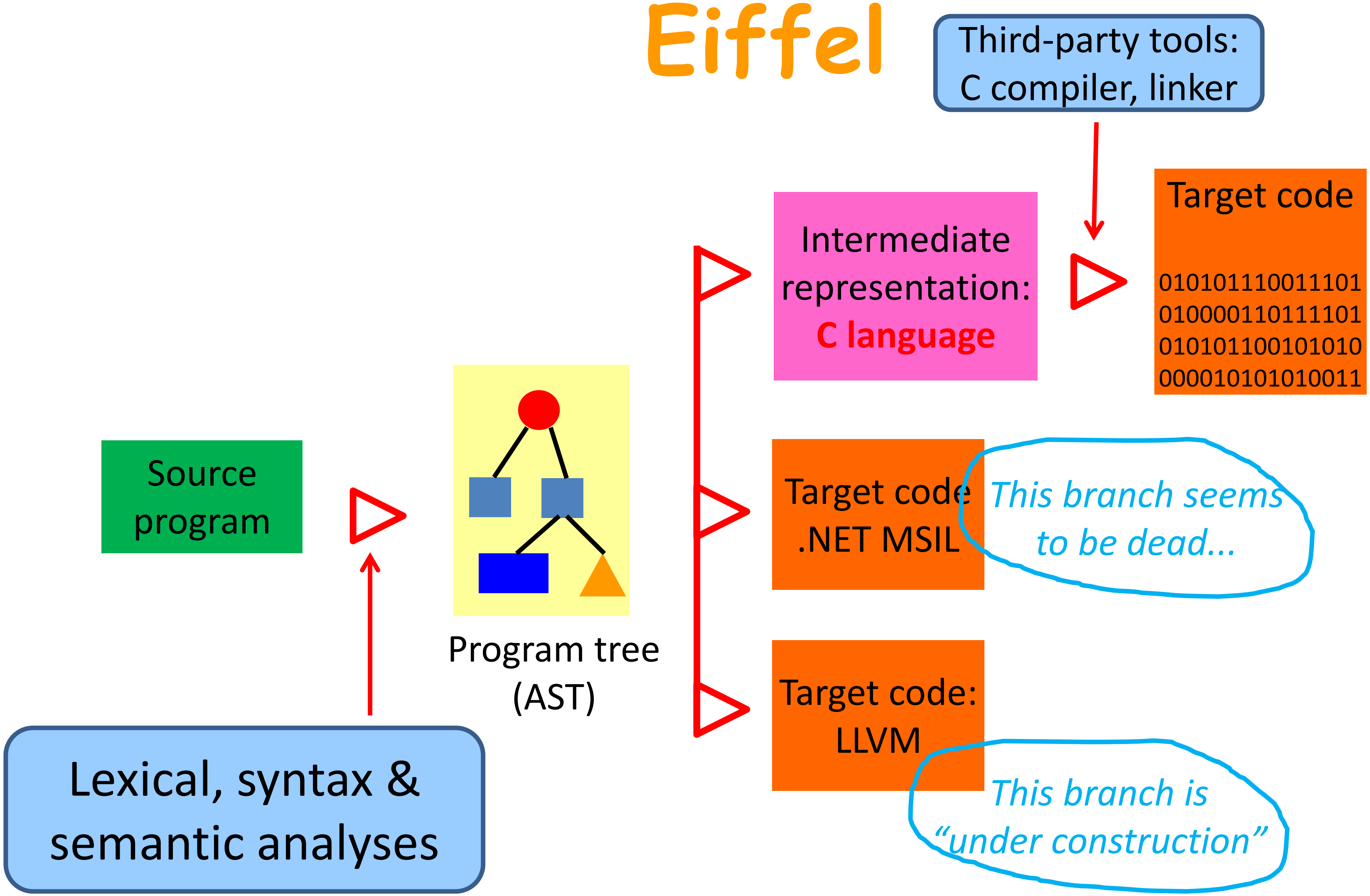
A P I

Third-party code generator

Target code

01010111001110101000
01101111010101011001
01010000010101010011

Compilation Phases & Structures: Eiffel



Third-party tools:
C compiler, linker

Source program

Program tree (AST)

Intermediate representation:
C language

Target code

010101110011101
010000110111101
010101100101010
000010101010011

Target code
.NET MSIL

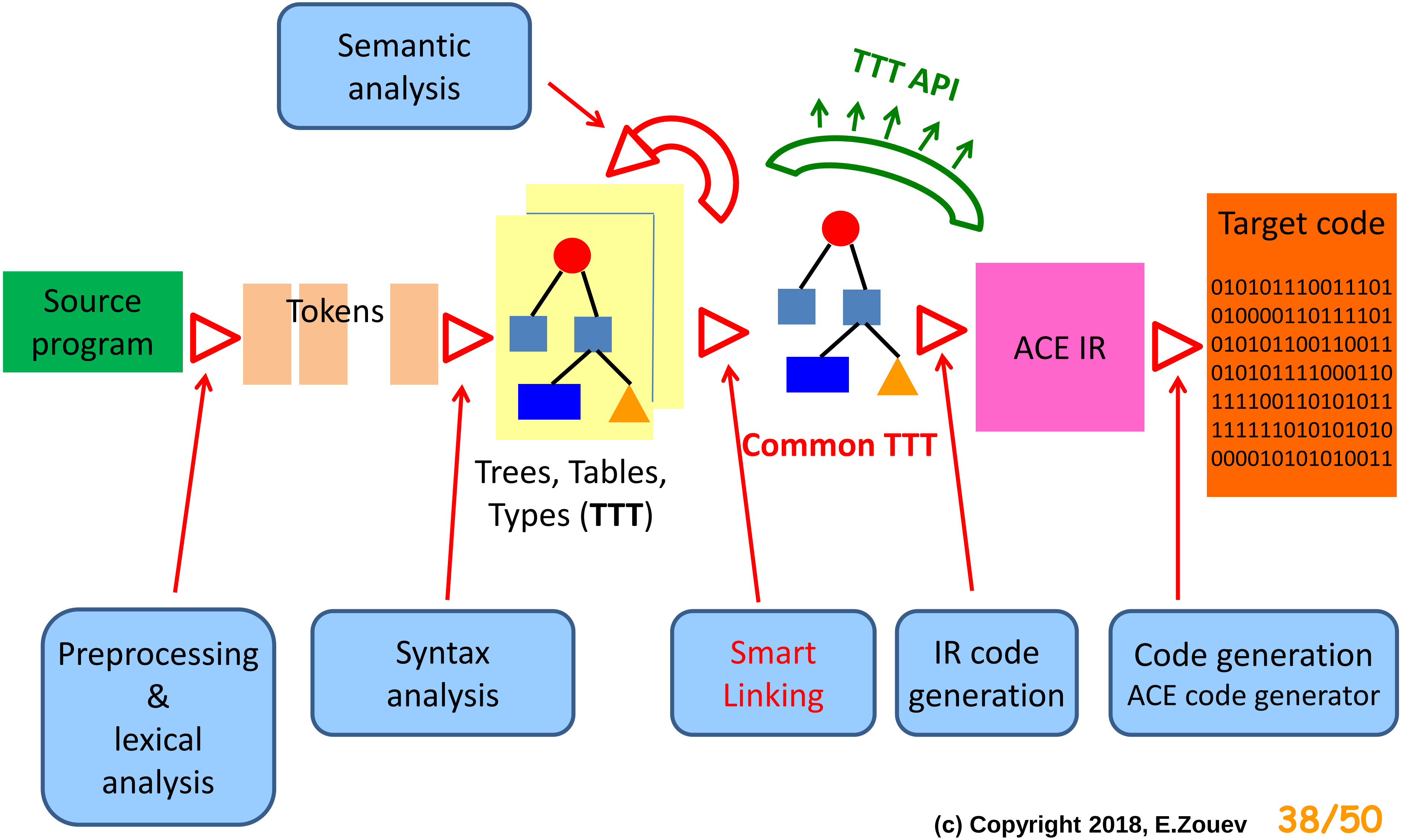
This branch seems to be dead...

Target code:
LLVM

This branch is "under construction"

Lexical, syntax & semantic analyses

Compilation Phases & Structures: Interstron C++



Semantic analysis

TTT API

Source program

Tokens

Trees, Tables, Types (**TTT**)

Common TTT

ACE IR

Target code

010101110011101
010000110111101
010101100110011
010101111000110
111100110101011
111111010101010
000010101010011

Preprocessing & lexical analysis

Syntax analysis

Smart Linking

IR code generation

Code generation
ACE code generator

(c) Copyright 2018, E.Zouev

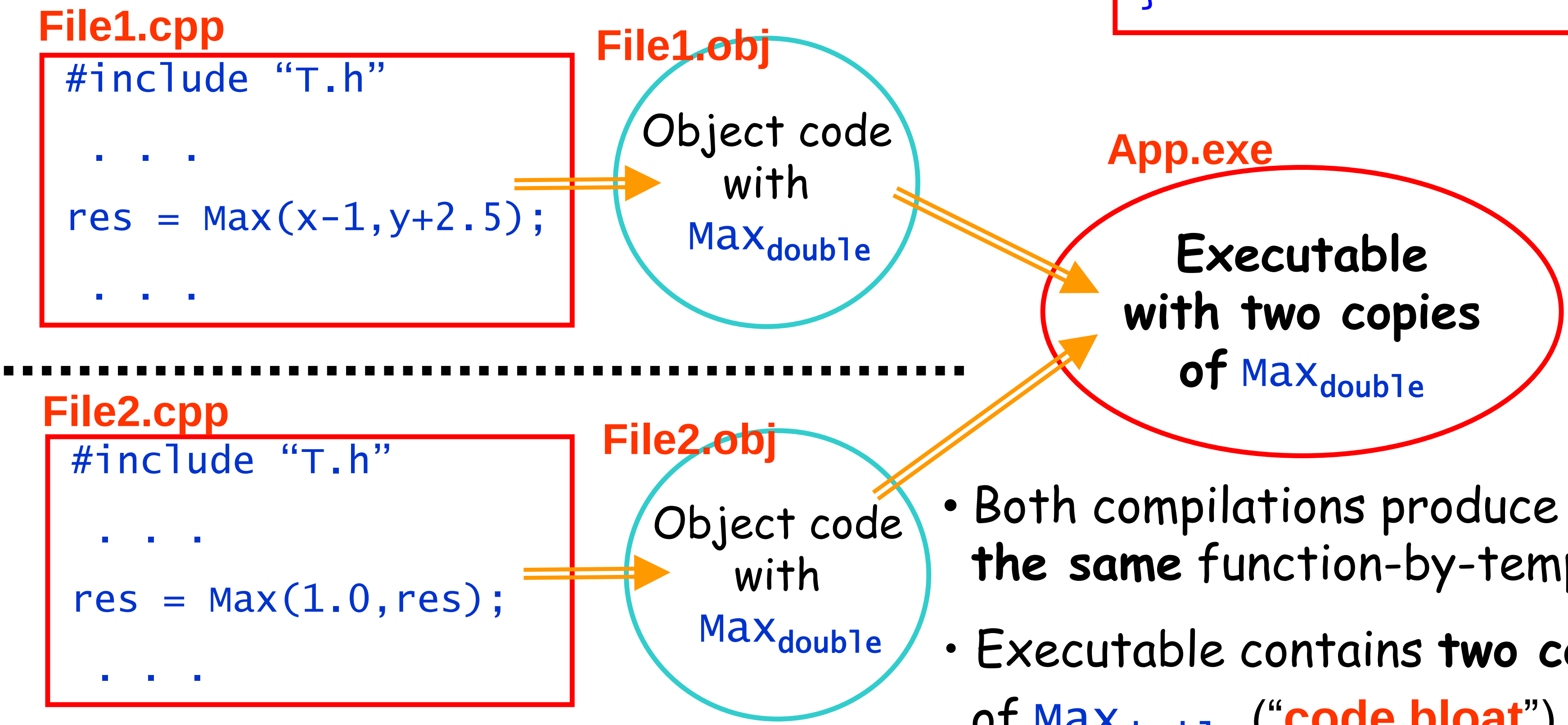
Linkage Phase & Code Bloat Problem

Template instantiation may cause **problems**.

Example: separate compilation

T.h

```
template<typename T>
T Max ( T a, T b )
{
    return a>b?a:b;
}
```



File1.cpp

```
#include “T.h”
 . . .
res = Max(x-1,y+2.5);
 . . .
```

File1.obj

Object code with $\text{Max}_{\text{double}}$

File2.cpp

```
#include “T.h”
 . . .
res = Max(1.0,res);
 . . .
```

File2.obj

Object code with $\text{Max}_{\text{double}}$

App.exe

Executable with two copies of $\text{Max}_{\text{double}}$

- Both compilations produce **the same** function-by-template
- Executable contains **two copies** of $\text{Max}_{\text{double}}$ (“**code bloat**”)

Compilation: Two-faced Janus

- Initial stages: language-dependent, and platform-agnostic:

  front-end compiler

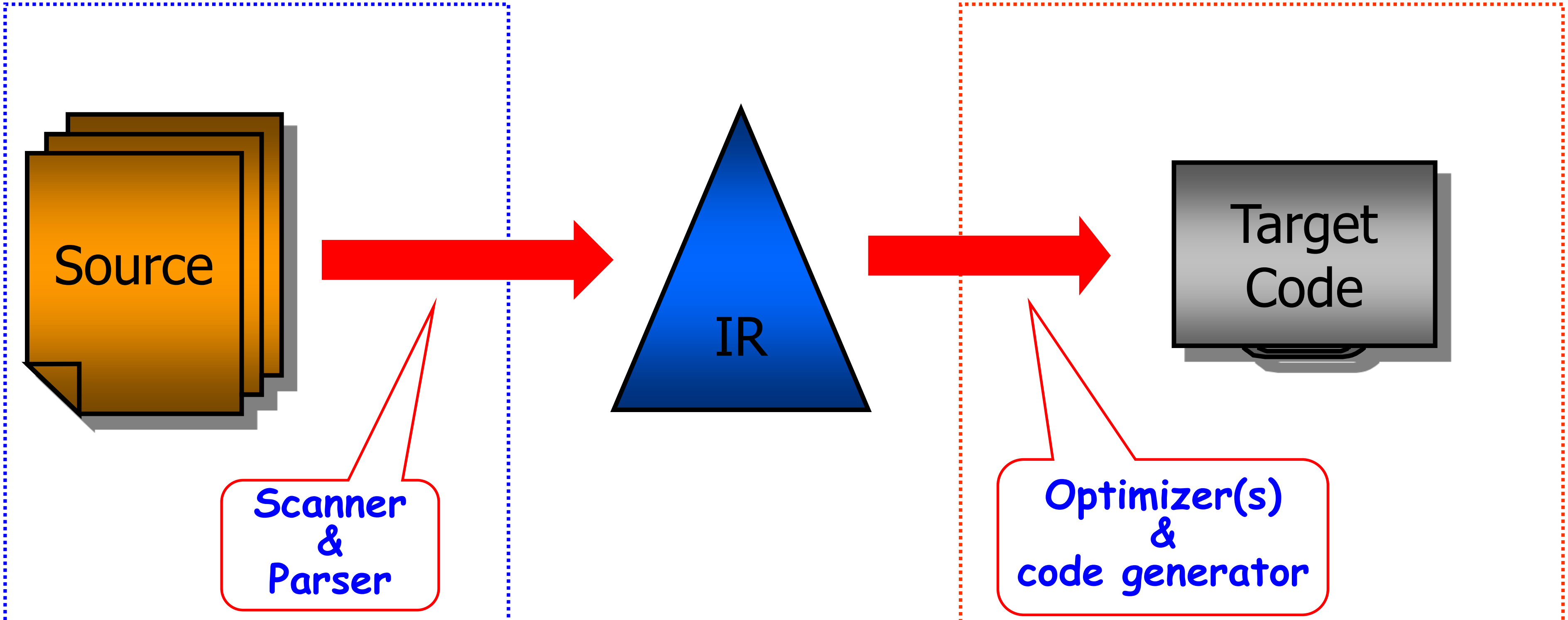
- Final stages: language-independent, and dealing with the specifics of the target machine architecture:

  back-end compiler

- Interface (“mediator”) between the two compiler components:

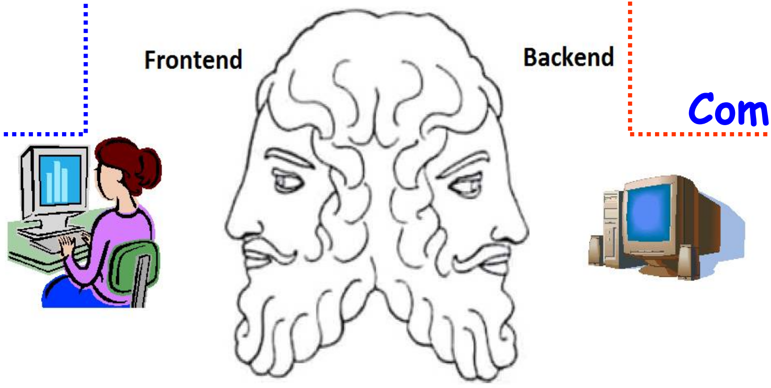
  intermediate representation

Compilation: Two-faced Janus

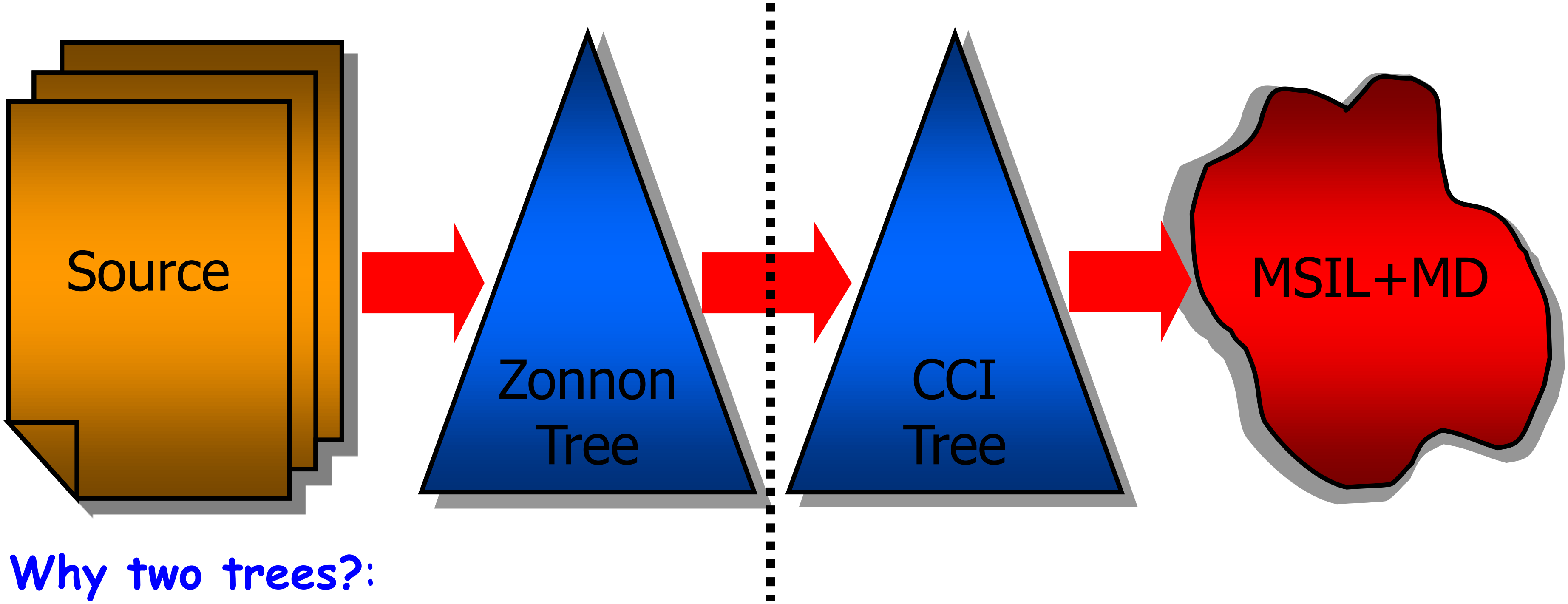


Front-end part:
Language specific &
Machine independent

Back-end part:
Target-specific
Common to all source languages



Zonnon Compilation Model

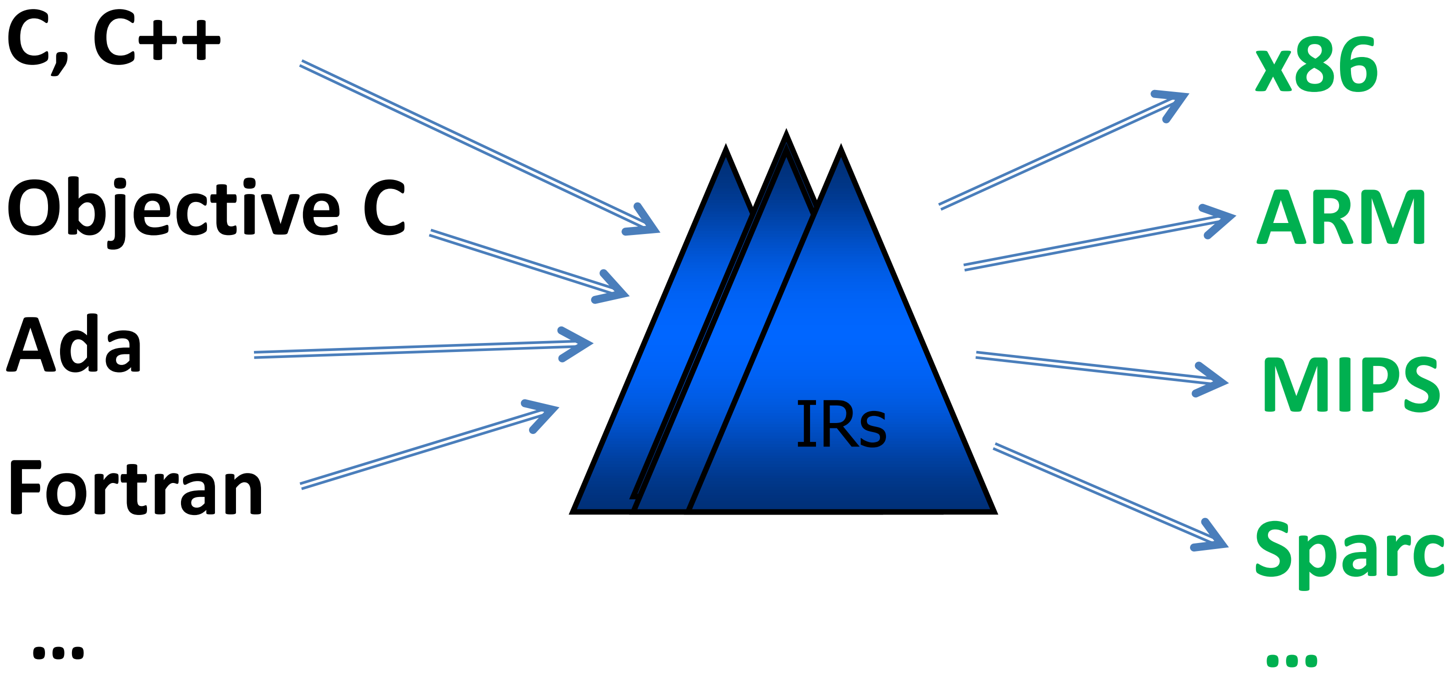


Why two trees?:

- They reflect a conceptual gap between Zonnon & CLR
- Zonnon semantic representation doesn't depend on CCI & the target platform
- Conversion Zonnnon tree -> CCI tree explicitly implements and capsulates mappings from Zonnon language notions to the CLR model.

The Advanced Architecture

- Multi-language compilation systems
- Multi-target compilation systems
- Typical example: **gcc** – GNU Compiler Collection



C, C++

Objective C

Ada

Fortran

...

IRs

x86

ARM

MIPS

Sparc

...

- A newer example: **Clang/LLVM** infrastructure

Compiler Development Technologies

- Compiler construction is quite complicated and non-trivial thing. In many cases there may be reasons for using special tools.
- In 60s-70s there were many ambitious supporting tools & systems called **compiler compilers**. They were quite hard-to-use and in general didn't provide the required quality of the resulting compilers.
- More modest but practical tools replaced "compiler compiler" monsters. They automated some compilation stages but were much more useful..
- General principle: development tool can facilitate compiler implementation, but you should clearly understand the price.
- The most popular are tools for generating **scanners** (examples are lex/flex) and **parsers**: yacc/bison and their clones and modifications, as well as ANTLR , COCO, xText) .
- Also, **code generation** can be automated by some modern infrastructures like CCI for .NET, xTend for JVM.
- Most compilers are written "by hand" ☺.

Compiler Development Technologies

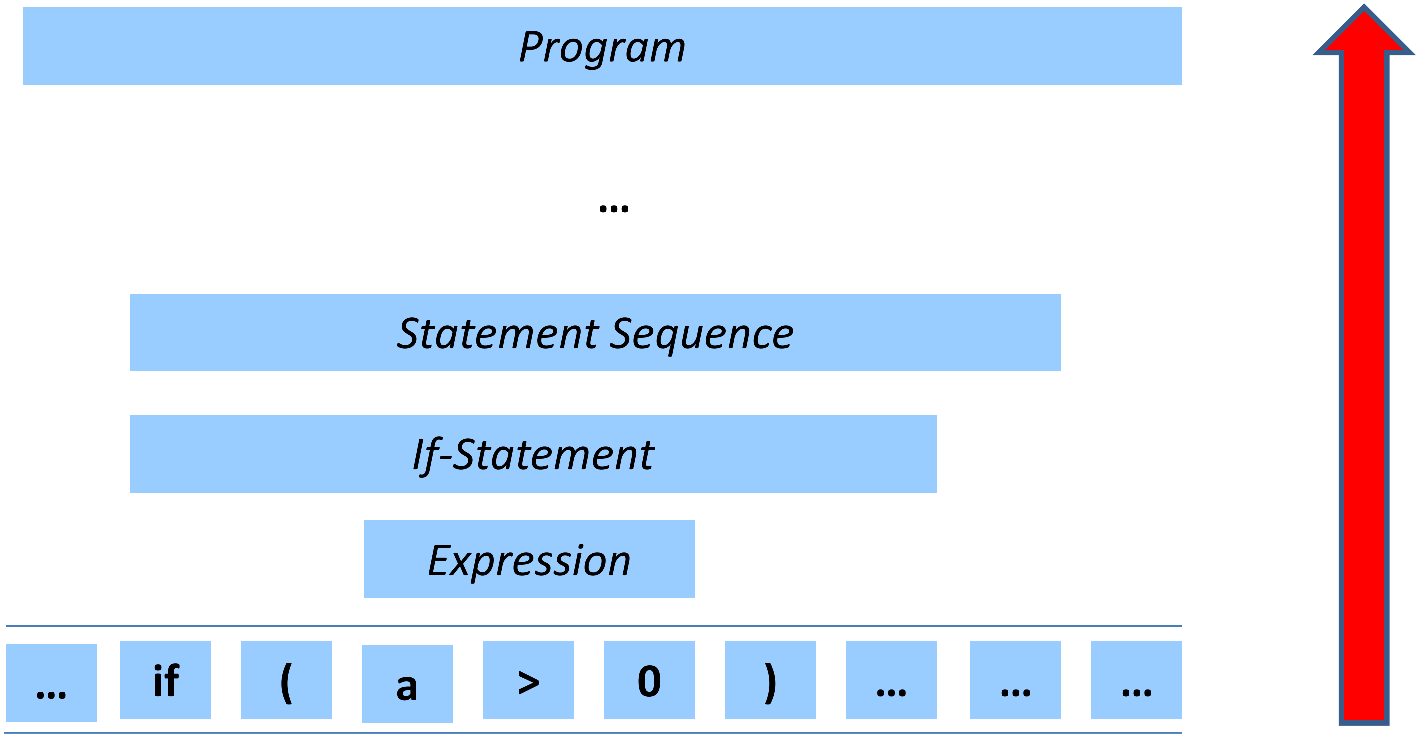
(Syntax analysis as an example.)

- Top-down or bottom-up parsing?
- «Hand-made» development or using tools?

	By hand	By tools
Top-down parsing	Most often: **Recursive descent parser**	ANTLR COCO etc.
Bottom-up parsing	Rarely (too complicated)	Yacc, Bison, Jay, GPPG, etc.

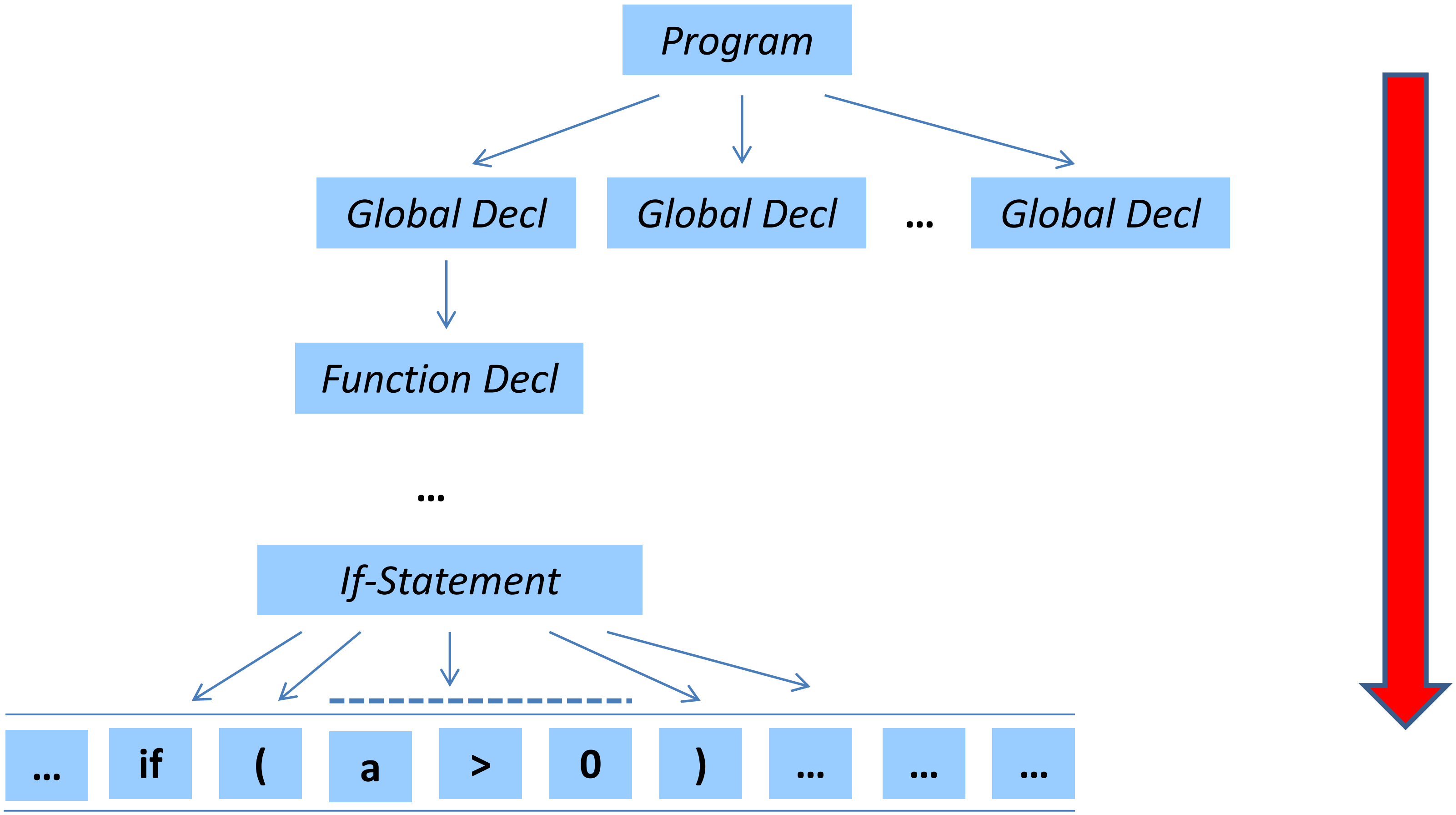
Compiler Development Technologies

Bottom-up Parsing



Compiler Development Technologies

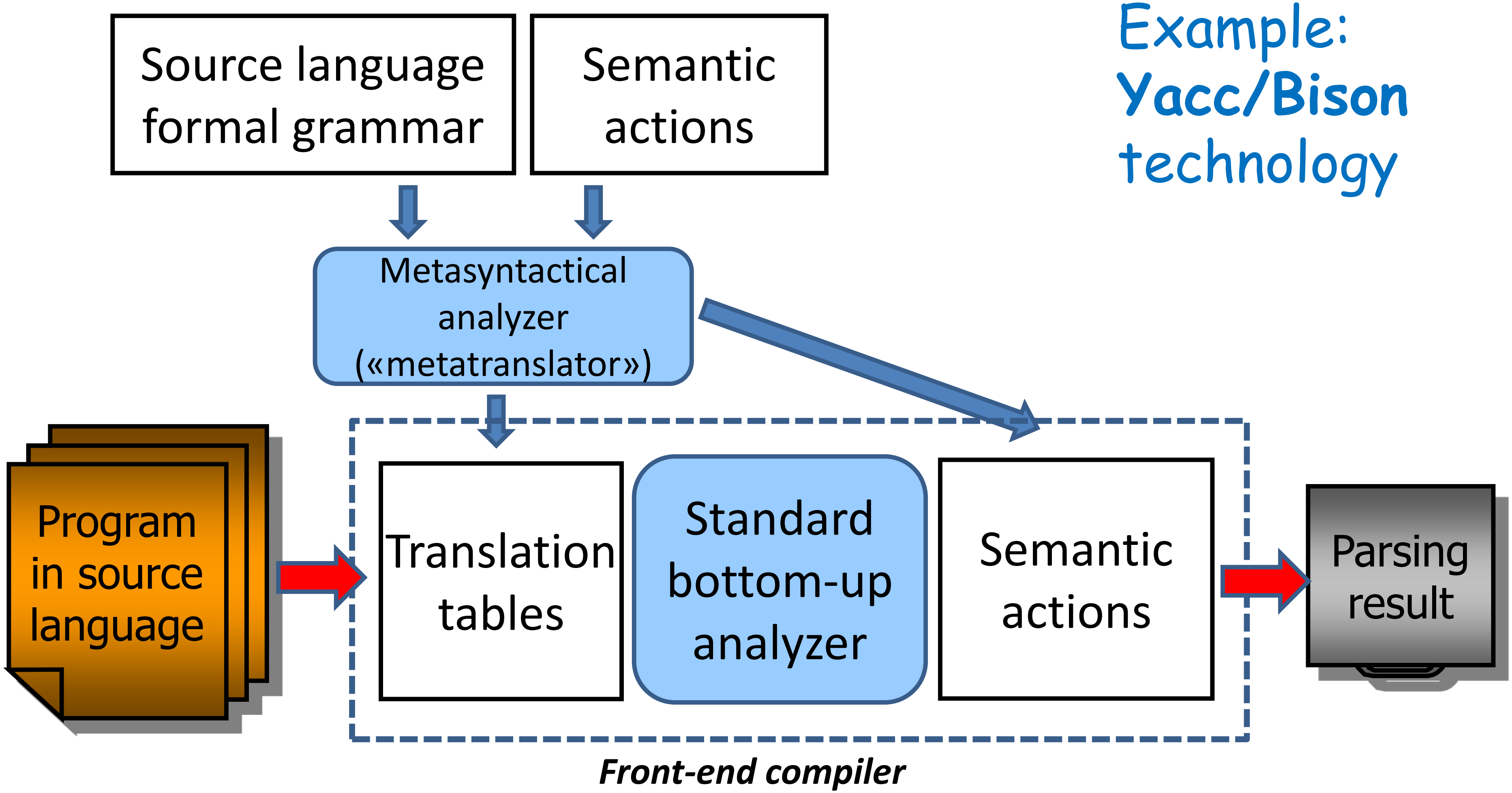
Top-Down Parsing



Compiler Development Technologies

- Top-down or bottom-up?
  - Top-down parsing is rather easy for programming.
  - Top-down parsing is less tolerant to syntax errors.
  - «Syntax» part of the compiler is bad structured (is spread across the compiler's text). It is hard to maintain and modify.
- «Hand-made» or automatically created compilers?
  - Tool may significantly speed up the development (the more language is complicated the better speed up we get).
  - Automatically generated parser is typically has poor interface with other compiler's parts.
  - The big problem is **grammar ambiguity** in many real programming languages. Often it's quite hard to redesign the grammar to avoid ambiguities.

Compiler Development Technologies


Source language formal grammar

Semantic actions

Metasyntactical analyzer («metatranslator»)

Program in source language

Translation tables

Standard bottom-up analyzer

Semantic actions

Parsing result

Front-end compiler

Example: **Yacc/Bison** technology

Program Optimization

- On the stage of lexical and syntax analyses

  Big spectrum of optimization techniques.

- On the stage of semantic analysis (while processing AST).

  A series of subsequent AST traversing actions. Each traversing can contain some optimizations which are heavily depend on the language semantics.

- On the code generation stage (machine dependent optimizations).

  Depend on the target architecture and machine instruction set.

- On the linking stage.

  Example: struggling against "code bloat" in C++.