Compiler Construction: Practical Introduction

Projects

Eugene Zouev
Fall Semester 2022
Innopolis University

Project List

- **Four "toy" languages**

  For a team of ~four persons each

D Dynamic *(like Javascript)*
O Object-oriented *(like Oberon-2)*
F Functional *(~CoreLisp; like Lisp & Scheme)*
I Imperative *(like Pascal/Oberon)*

- **Implementation details**

  - Compiler or interpreter
  - Implementation languages: C, C++, C#, Java
  - Tools: yacc/bison, hand-made
  - Target codes & platforms:

    - MSIL or Java bytecode
    - LLVM bitcode
    - C
    - Own VM ☺

- **One "real" language: Lua**

  For a team of ten persons

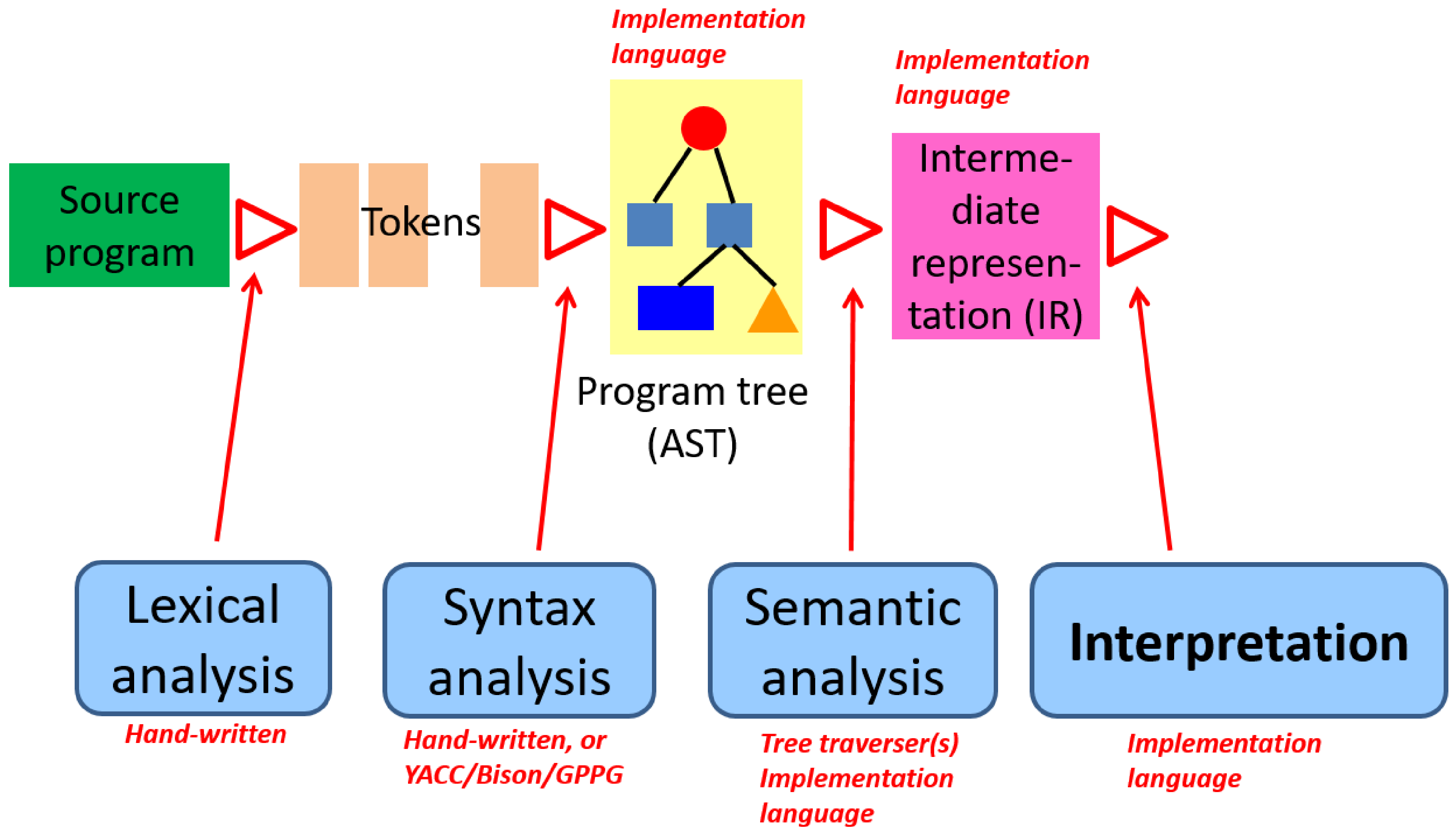
(like Lua ☺)
Hand-written Lua->LLVM compiler

Project D: Dynamic

Short language characteristics

- Object types are not specified and can change while program execution
- The language is **interpreted**
- Major notion: variable & literal (constant)
- Program structure: a sequence of declarations and statements
- Builtin types: built-in: integer, real, boolean, string
  User-defined types: array, tuple, function
- Implicit type conversions
- Statements: assignment, if/while, return, input/output

Project D: Dynamic

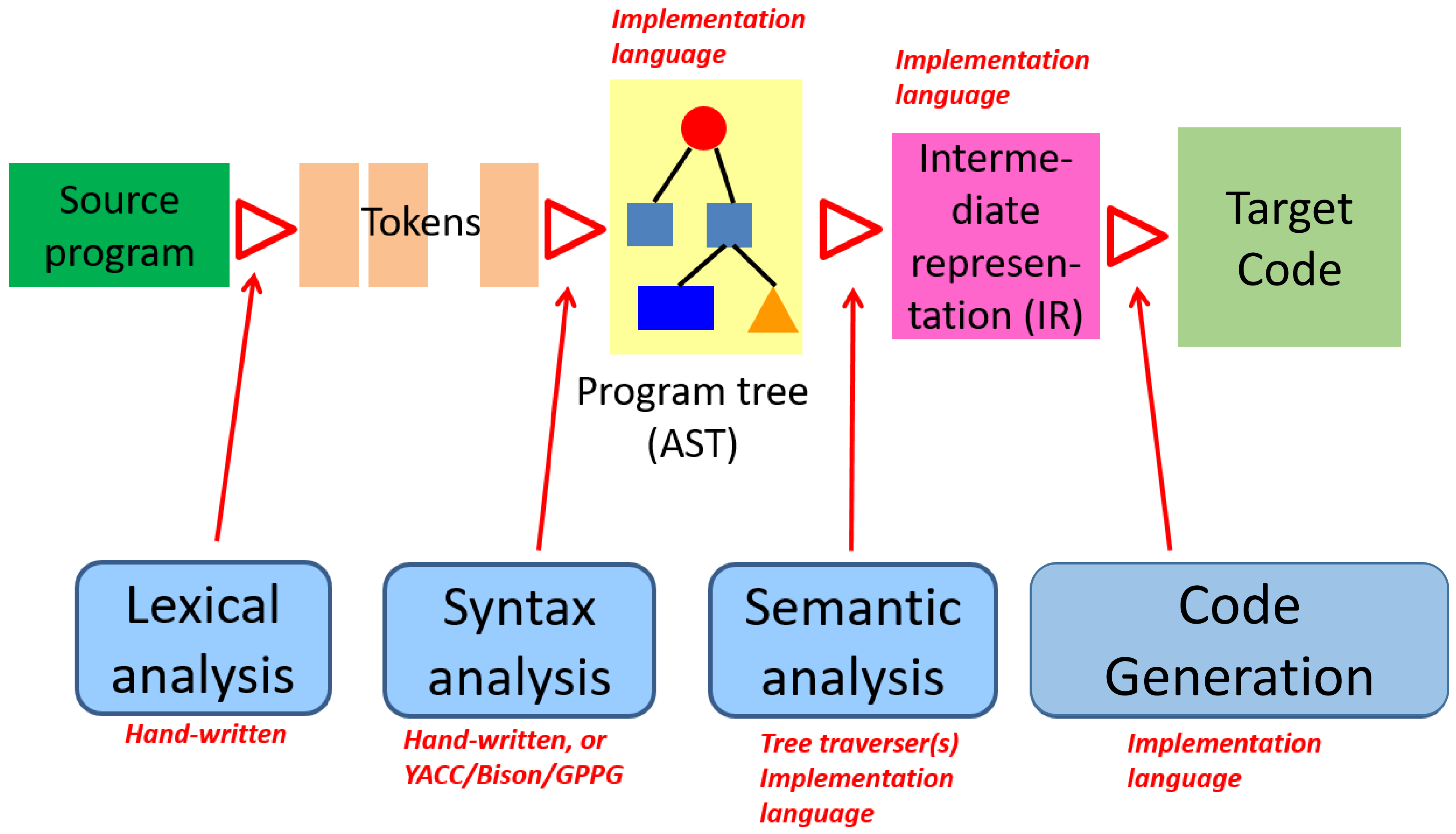


Project O: Object-Oriented

Short language characteristics

- The language is static. Object types are fixed by object declarations and cannot change while program execution.
- Object types are classes – either predefined of user-defined.
- Classes contain data declarations and method declarations.
- The single inheritance and virtual functions are supported.
- No expressions! – only function calls.
- The language is **compiled**. The target code is either an assembly language, of LLVM bitcode, or JVM bytecode, or .NET CIL.
- Program structure: a sequence of classes.
- Statements: a standard set (assignment, if/while, return, input/output)

Project O: Object-Oriented

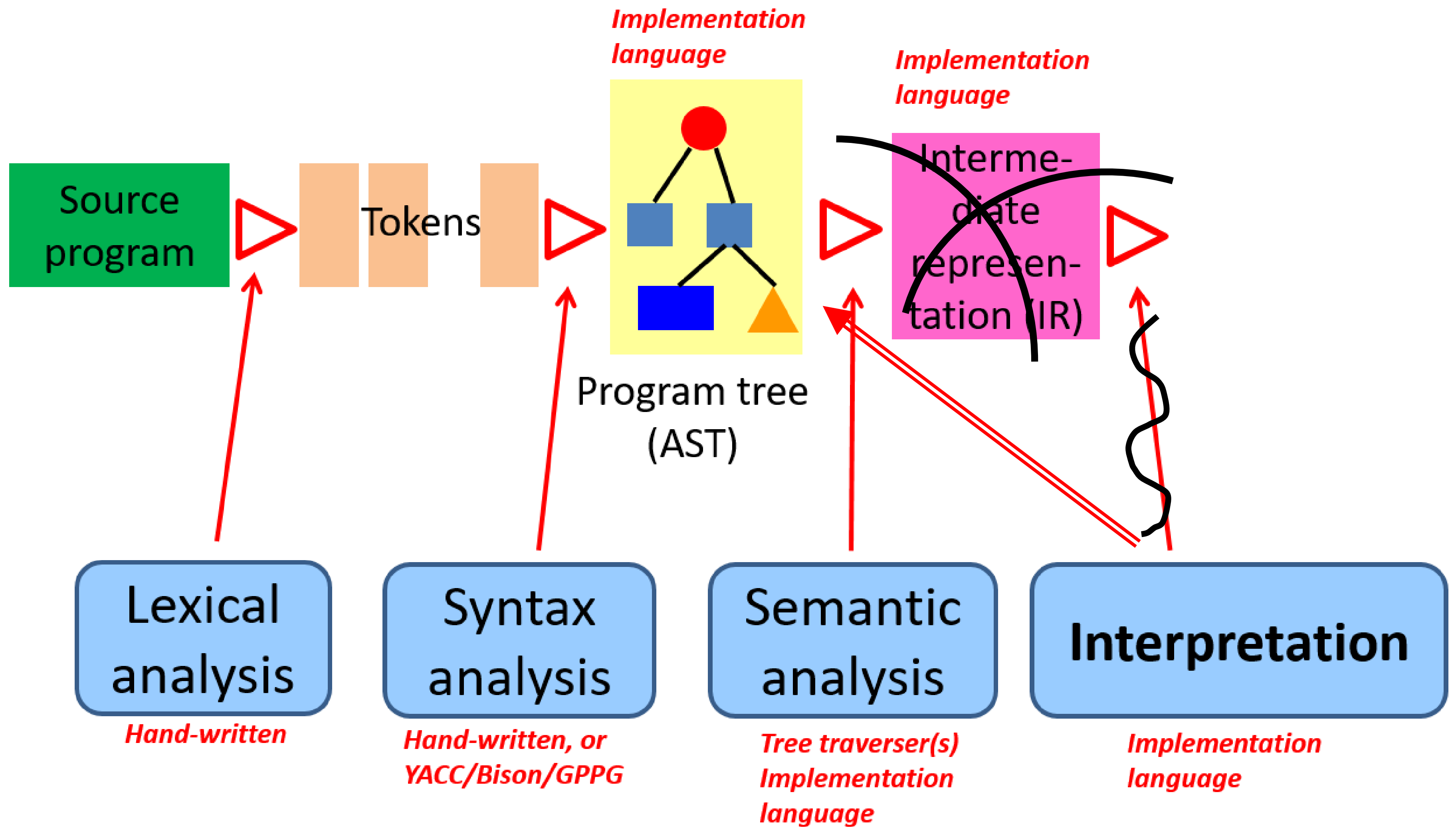


Project F: Functional

Short language characteristics

- There are three atomic predefined types: integers, reals & booleans.
- There are two basic structures: atoms and lists.
  Atom has the name and the value. The atom value can be either of a predefined type, or, or list. In some contexts atom represents itself.
  A list is an ordered sequence of atoms, literals and lists.
- Basic list access rules include taking the head of the list, the "tail" of the list, and constructing a list from its head and the "tail"..
- The language is dynamically typed and **interpreted**.
- Program structure: a sequence of lists and/or atoms
- Special forms (lists): functions, lambdas (unnamed functions), control structures.
- A minimal set of predefined functions: arithmetic, lists operations, etc.

Project F: Functional

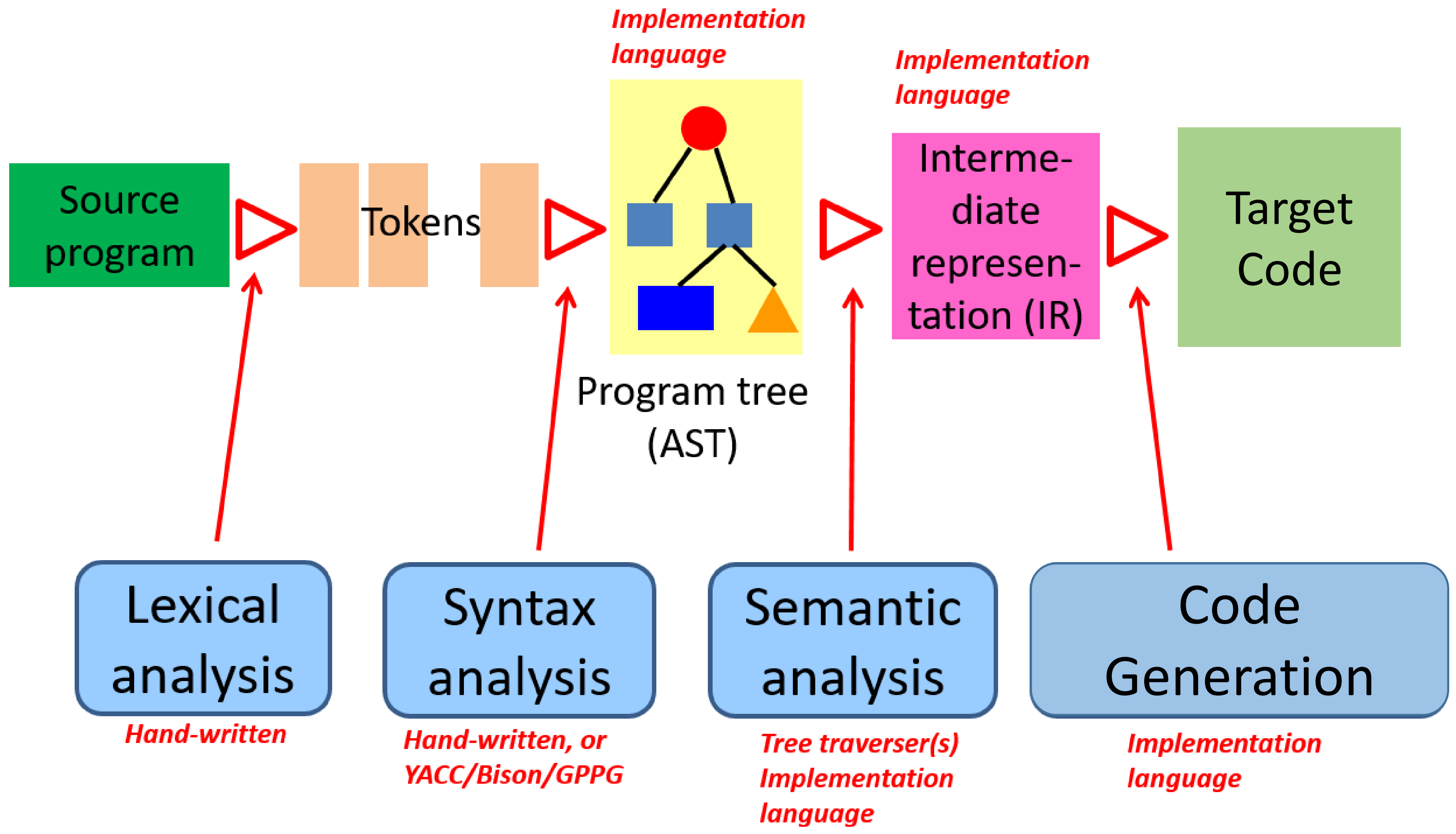


Project I: Imperative

Short language characteristics

- The language is static. Object types are fixed by object declarations and cannot change while program execution.
- There are three predefined data types: integer, real and boolean. There are two predefined data structures: structs and arrays.
- Full expression syntax with usual set of operators.
- The language is **compiled**. The target code is either an assembly language, of LLVM bitcode, or JVM bytecode, or .NET CIL.
- Program structure: a sequence of data and routine declarations.
- Statements: a standard set (assignment, if/while, return, input/output)

Project I: Imperative



Project Aul: The Lua Compiler for LLVM

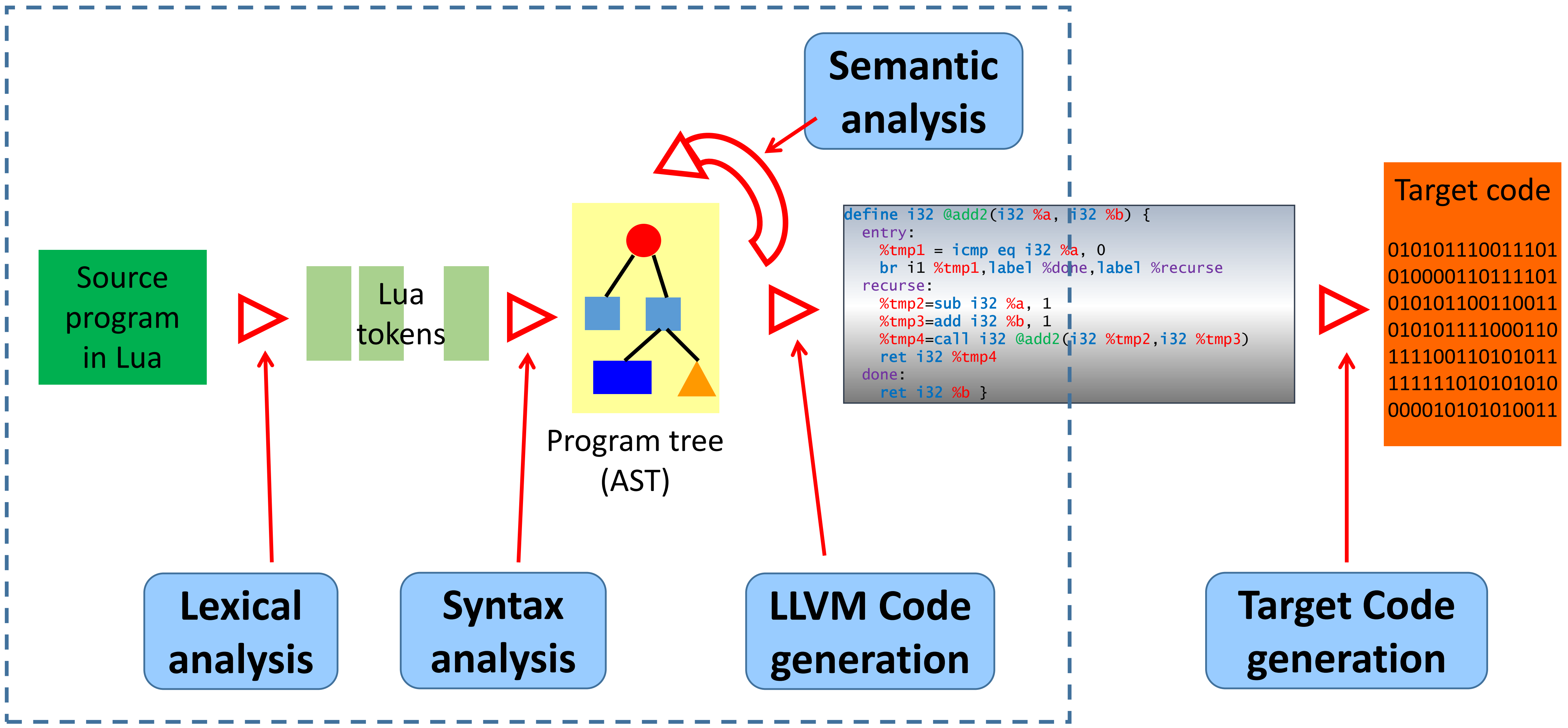
The Task Description

The project is aimed at providing users with a powerful industrial-strength tool for developing application programs in the Lua programming language. In addition, the tool would support efficient compilation of the existing code base written in Lua.

The Aul compiler translates source code written in the Lua programming language directly to the LLVM bitcode that later gets converted to the machine code for the target hardware. Thus, the interpretation phase will be avoided.

- Lua version 5.4.3
  https://www.lua.org/manual/5.4/manual.html
  https://lua.org.ru/contents_ru.html
- A **small subset** of Lua should be implemented

Project Aul: The Lua Compiler for LLVM



```
define i32 @add2(i32 %a, i32 %b) {
  entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1,label %done,label %recurse
  recurse:
    %tmp2=sub i32 %a, 1
    %tmp3=add i32 %b, 1
    %tmp4=call i32 @add2(i32 %tmp2,i32 %tmp3)
    ret i32 %tmp4
  done:
    ret i32 %b }
```

```
010101110011101
010000110111101
010101100110011
010101111000110
111100110101011
111111010101010
000010101010011
```