# System Software Crash Couse

Samsung Research Russia
Moscow 2019

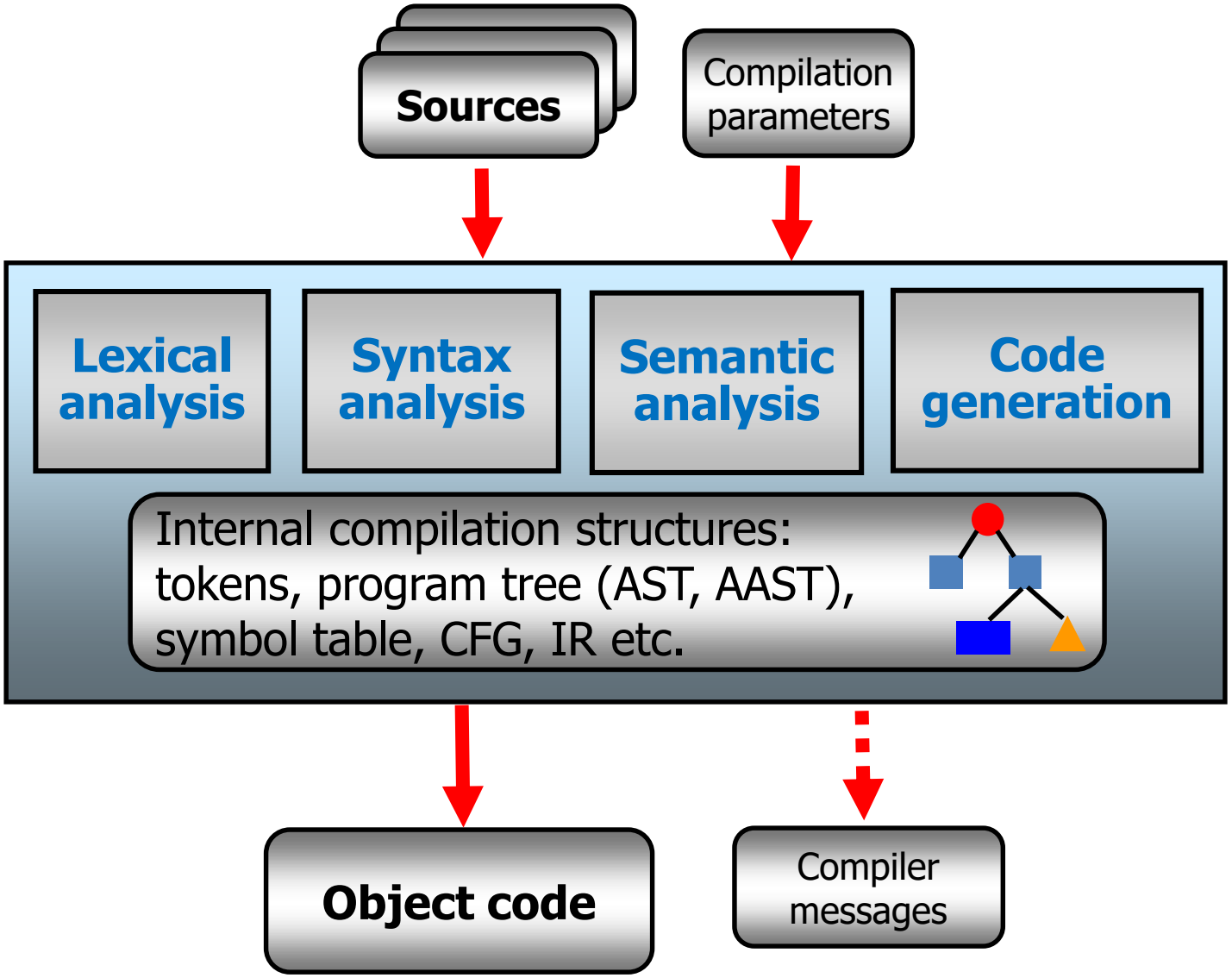
## Block C Compiler Construction
## 13. The Evolution of the Compiler Architecture

Eugene Zouev

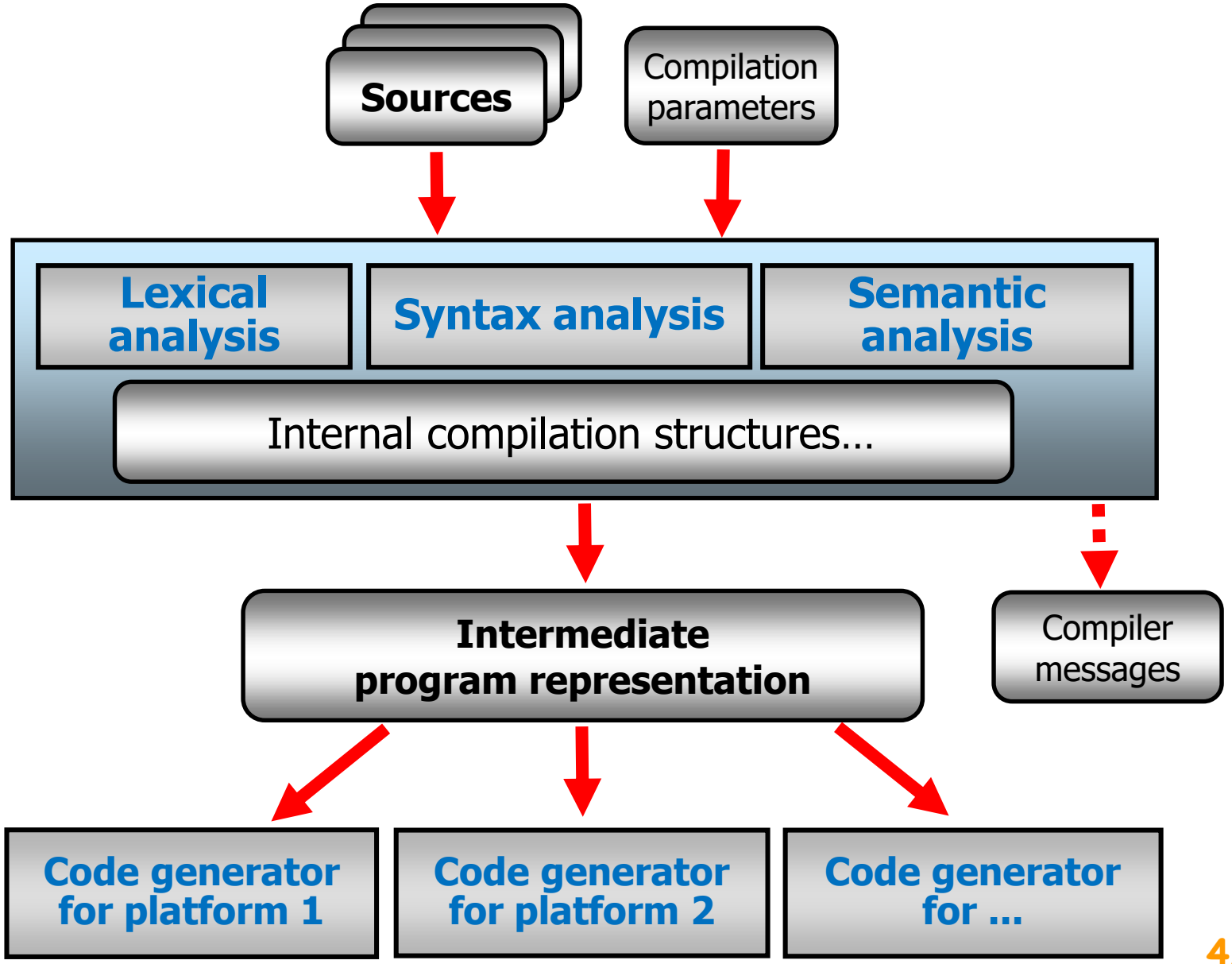
# Outline

- «Conventional» compiler
- Multi-platform architecture
- Compilation task. Compilation in a narrow & in a wide sense
- Advanced architecture, or How to turn compiler inside out?
- Program semantic representation: API or an open format?
- Compiler integration into an IDE

# Conventional Compiler



# Multi-platform Architecture



# Main Problems

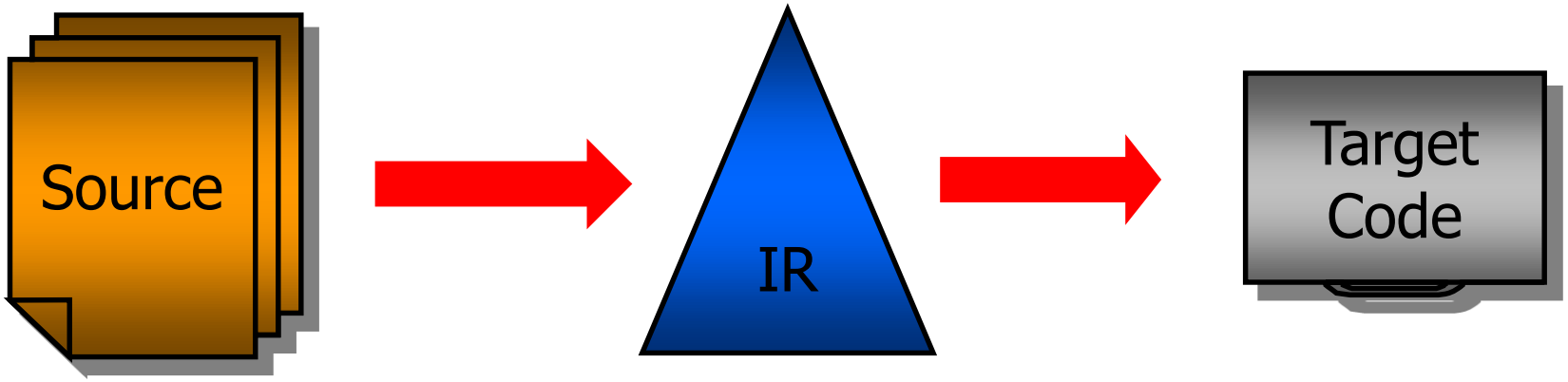
- **Efficient code generation**
  *Is solved by conventional compilers*

- **Program analysis**

- **Integration compilers & other language-related tools into integrated development environments (IDEs)**

# Compilation in a "narrow sense"

- Analysis of a program for producing semantically equivalent machine code for direct execution.




**However**, producing machine code is not the single compilation task – and often is **even not the most important one**!

# Why compilation "in narrow sense" is not enough?

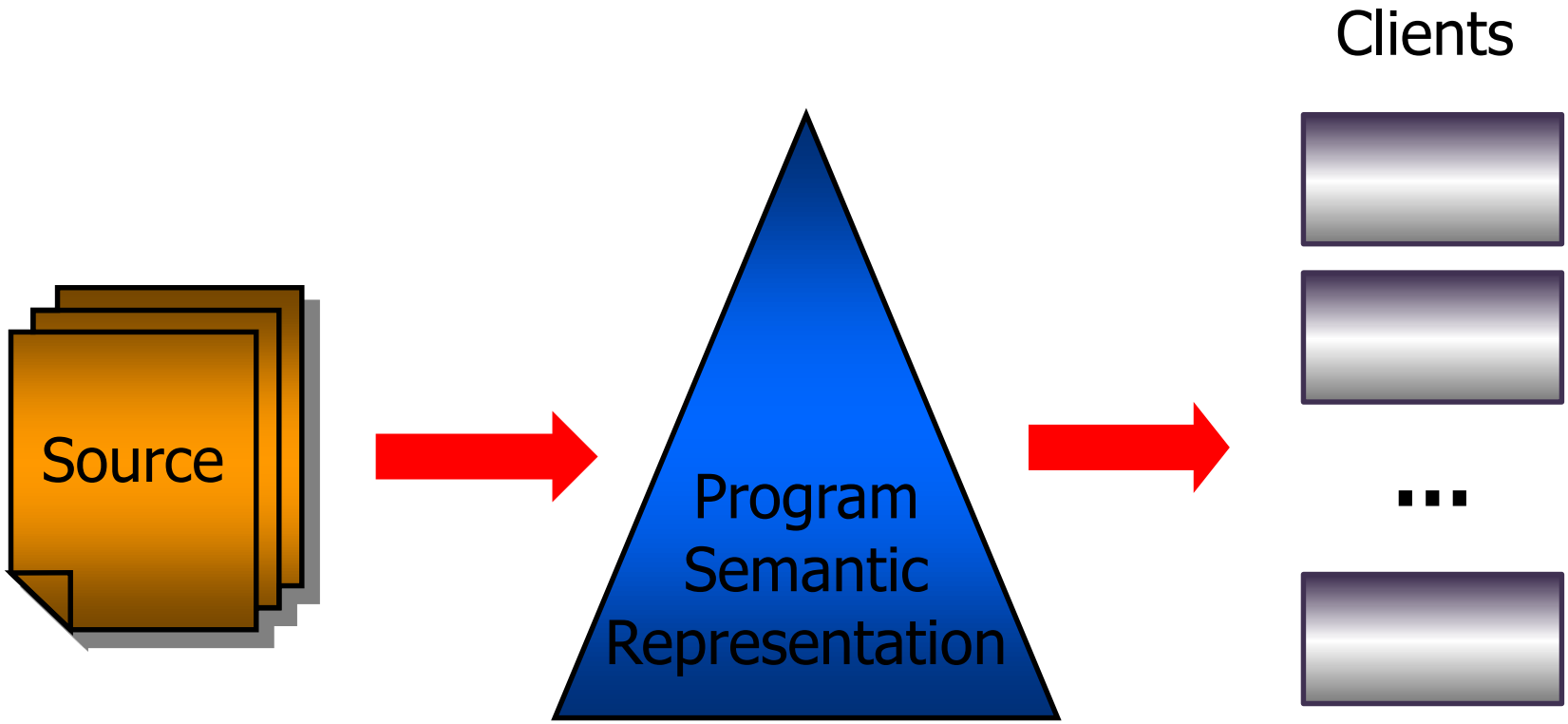
There are many actual tasks ("challenges") **not related to producing executable code**:

- **Legacy code reengineering**; source-to-source translation; automatic program generation.
- Maintenance and improving existing programs; **refactoring**.
- **Program static analysis**: detailed diagnostics without executing code; eliminating vulnerabilities, potentially problematic code; "dead code" eliminating; source level optimizations.
- **«Understanding»** programs; visualization: creating UML diagrams (reverse engineering), XREF diagrams; metrics calculation.
- **Testing**; creating test coverages.
- Program **verification**: formal correctness proving; abstract interpretation; partial interpretation.
- ...

To solve these and similar tasks **we need full information about program semantics**.

# Compilation in a "wide sense"

- Analysis of a program for getting full information about **all its features**.



# How these problems are solved?

- By creating **specific software tools & systems**: from simple utilities like **lint** for C (or **lint++** for C++) to powerful systems like **Klocwork**, **Fortify**, or **Coverity**.

  Typically, they are either command-line utilities, or "hermetic" systems without any possibilities for improving their functionality.

- By creating **open infrastructures** (APIs) providing access to programs' semantics.

# Open infrastructures: some examples

**ASIS**

**A**da **S**emantic **I**nterface **S**pecification (for Ada95, Ada2005, …): ISO standard.

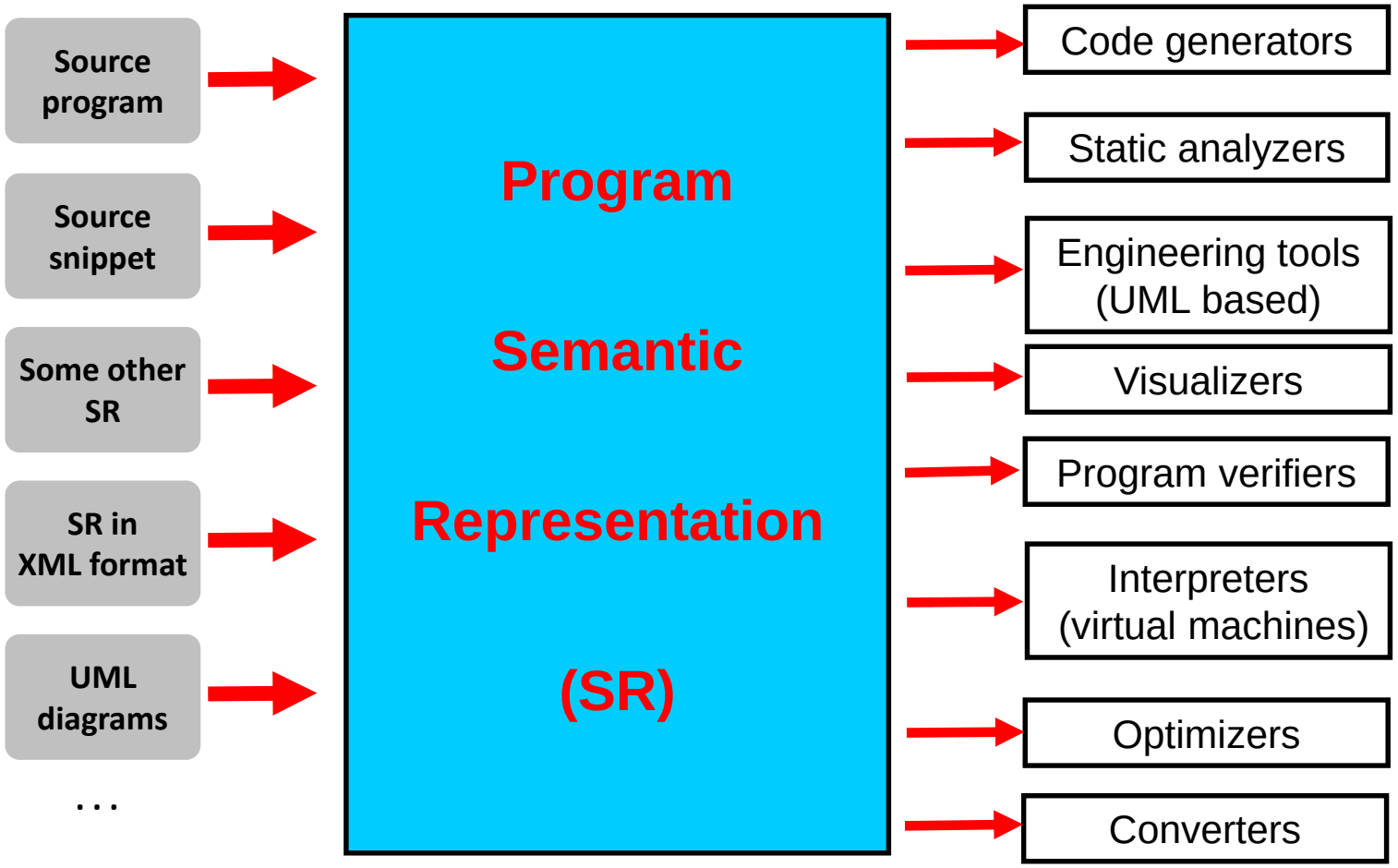
**SAGE - SAGE II - ROSE** (for C/C++, HPF…)

Open infrastructure for source-to-source transformations.

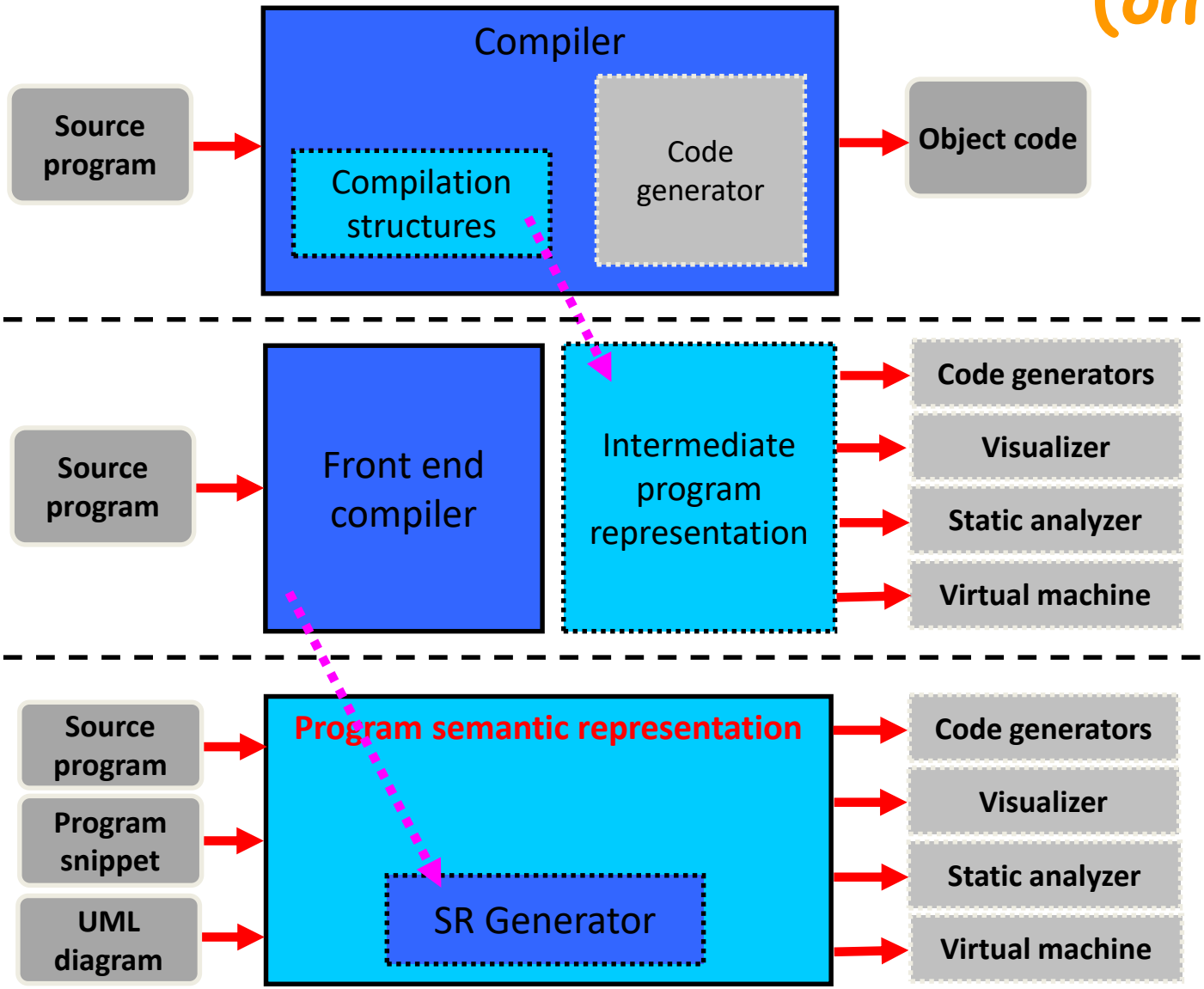
**Pivot** (for C++)

Stroustrup & Dos Reis; "Common infrastructure for C++ program conversions and static analysis".

# Compilation in a "wide sense": conceptual view



# The evolution of comp.architecture (on one slide)

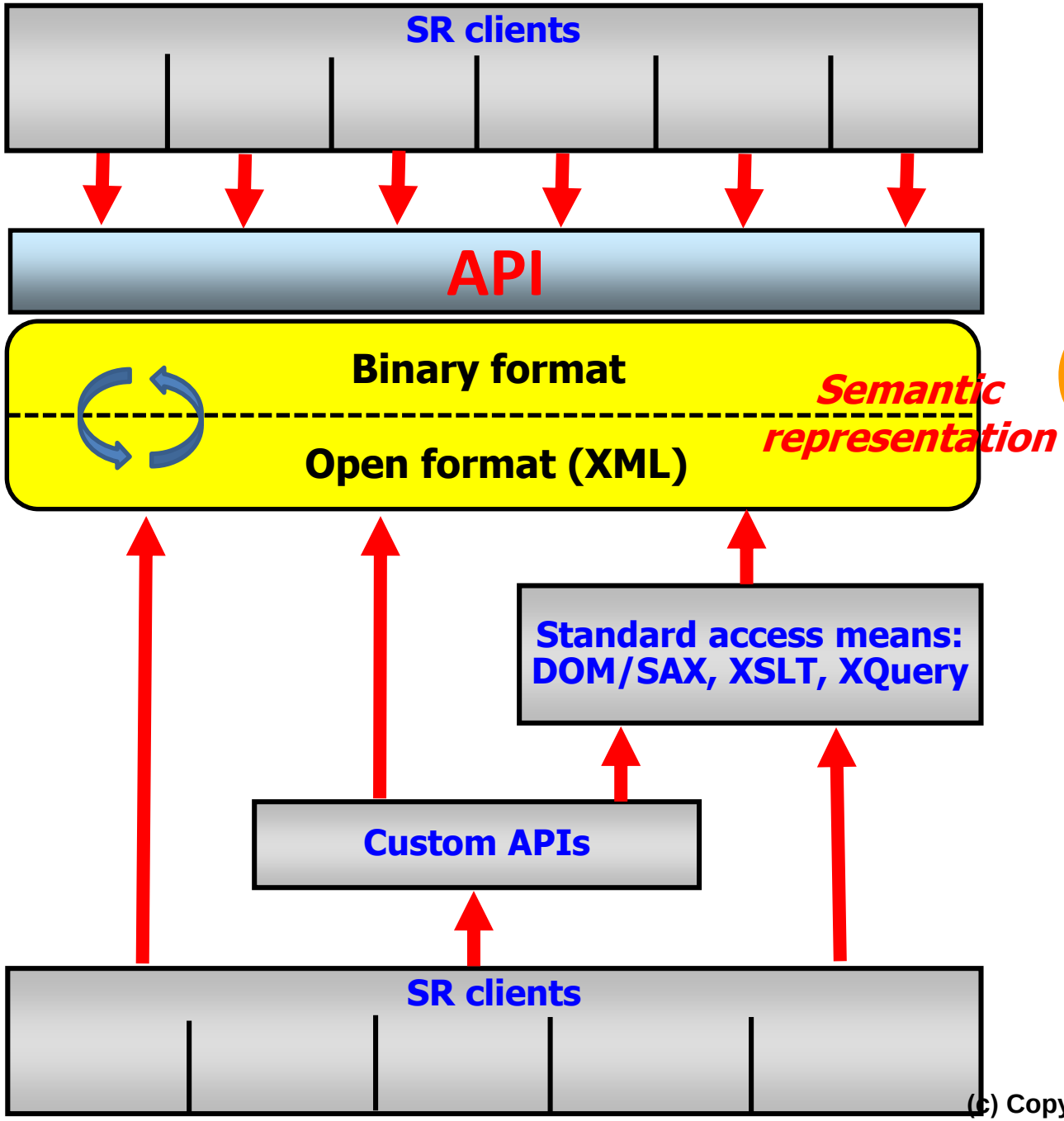



*Conventional monolithic compiler*

*Multi-target compilation system*

*SR is the common basis for various language-related tools*

# Binary or open format? (or both?)



# Why XML?

- Open format
- Extendable (XML is the *metalanguage* actually)
- Extremely simple representation model
- De facto standard
- Plenty of available XML-based technologies (XQuery, XSLT etc.) and tools for XML manupulating

# SR example in XML format

```
while ... {
    x = ...;
    P(...);
}
```

```
<while-statement ln="1" col="1">
    <condition>
        <expression ln="1" col="7"> ... </expression>
    <condition>
    <compound-statement>
        <assignment-expression ln="2" col="4">
            <name ln="2" col="4">x</name>
            <expression ln="2" col="9"> ... </expression>
        </assignment>
        <call ln="3" col="4">
            <name ln="3" col="4">P</name>
            <argument-list>
                <expression ln="3" col="5"> ... </expression>
            </argument-list>
        </call>
    </compound-statement>
</while-statement>
```

simplified

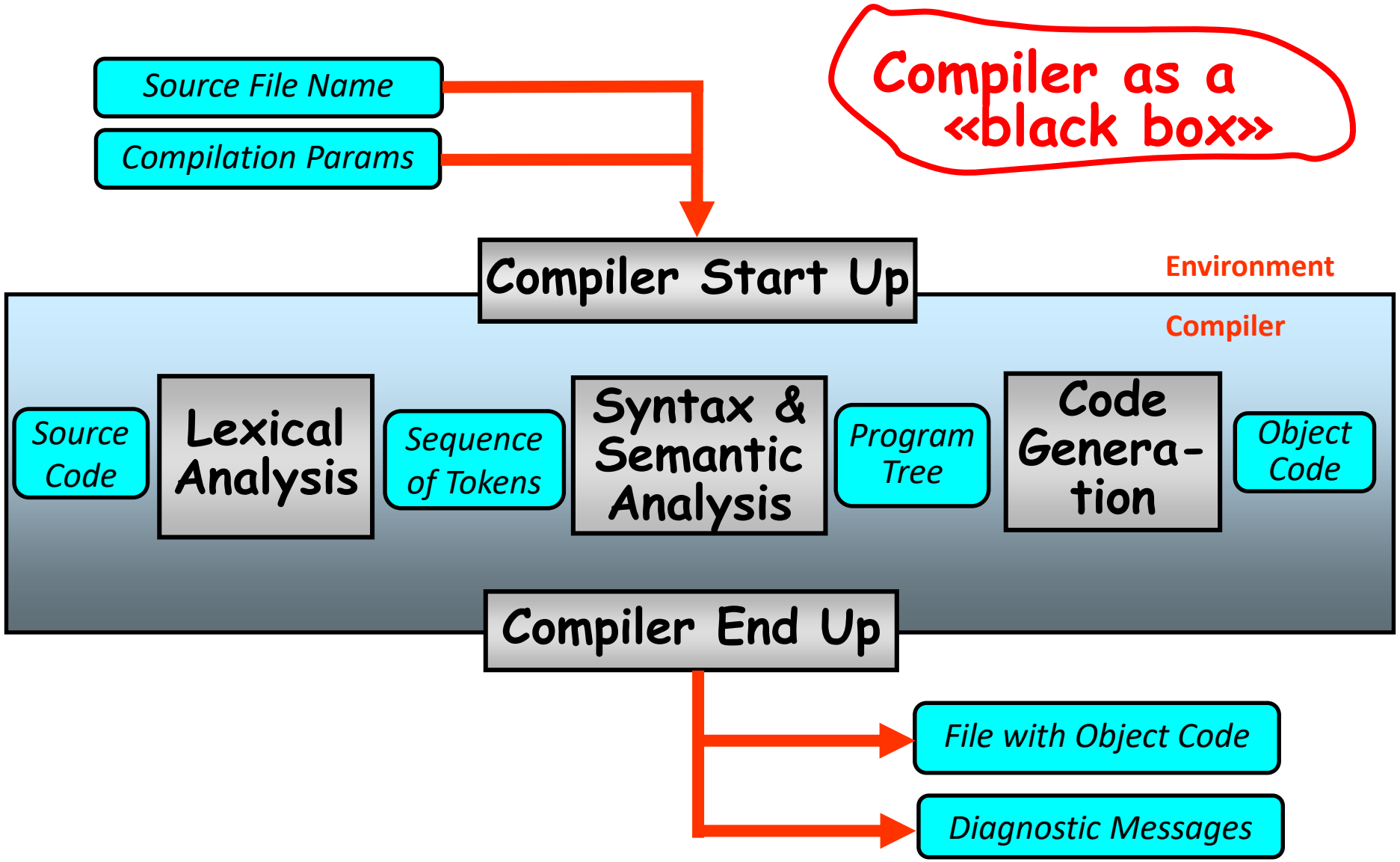
# Main Problems

- **Efficient code generation**
  *Is solved by conventional compilers*

- **Program analysis**

- **Integration compilers & other language-related tools into integrated development environments (IDEs)**

# Compilation: Conventional Approach



# The Task: Compiler Integration

**IDE components**

**Project Manager**

**Text Editor**

**Semantic Support ("Intellisense")**

**Debugger**

- Type identification
- Auto-completion
- Background compilation

**Features to be supported by compiler**

- Language sources identification

- Syntax Highlighting
- Automatic text formatting
- Smart text browsing { ➔ }
- Error checking while typing
- Tooltip-like diagnostics & info
- Outlining: collapsing parts of the source;

- Type member lists for classes and variables of class types
- Lists of overloaded methods
- Lists of method parameters

- Expression evaluation
- Conditional breakpoints

# Compiler Integration (1)

## Example: Semantic information from the C# compiler using by Visual Studio

Program.cs | Program.cs | CompoundSource.cs | Declarations.cs | Scanner.cs | Program.

ansic2xml_API.Program | open()

```
60
61          public void open()
62          {
63              // Initialize text templates for diagnostic messages.
64              Errors.init();
65              // Initialize program tree.
66              ENTITY.initTree(errors,options.initial);
67
68              // Initializing scanner and parser
69              Scanner scanner = null;
70              if ( options.preprocess ) scanner = new Preprocessor(source,options,errors);
71              else                      scanner = new Scanner(source,options,errors);
72
73              Parser parser = new Parser(scanner,options,errors);
74
75              // Start processing the source
76              try
77              {
78                  program = parser.parseProgram();
79              }
80              catch (Excepti
81              {
82                  throw new
83              }
84
```

void Parser.parseProgram(ANSIC.PROGRAM program)

Ошибка:
Ни одна из перегрузок метода "parseProgram" не принимает "0" аргументов

# Compiler Integration (2)

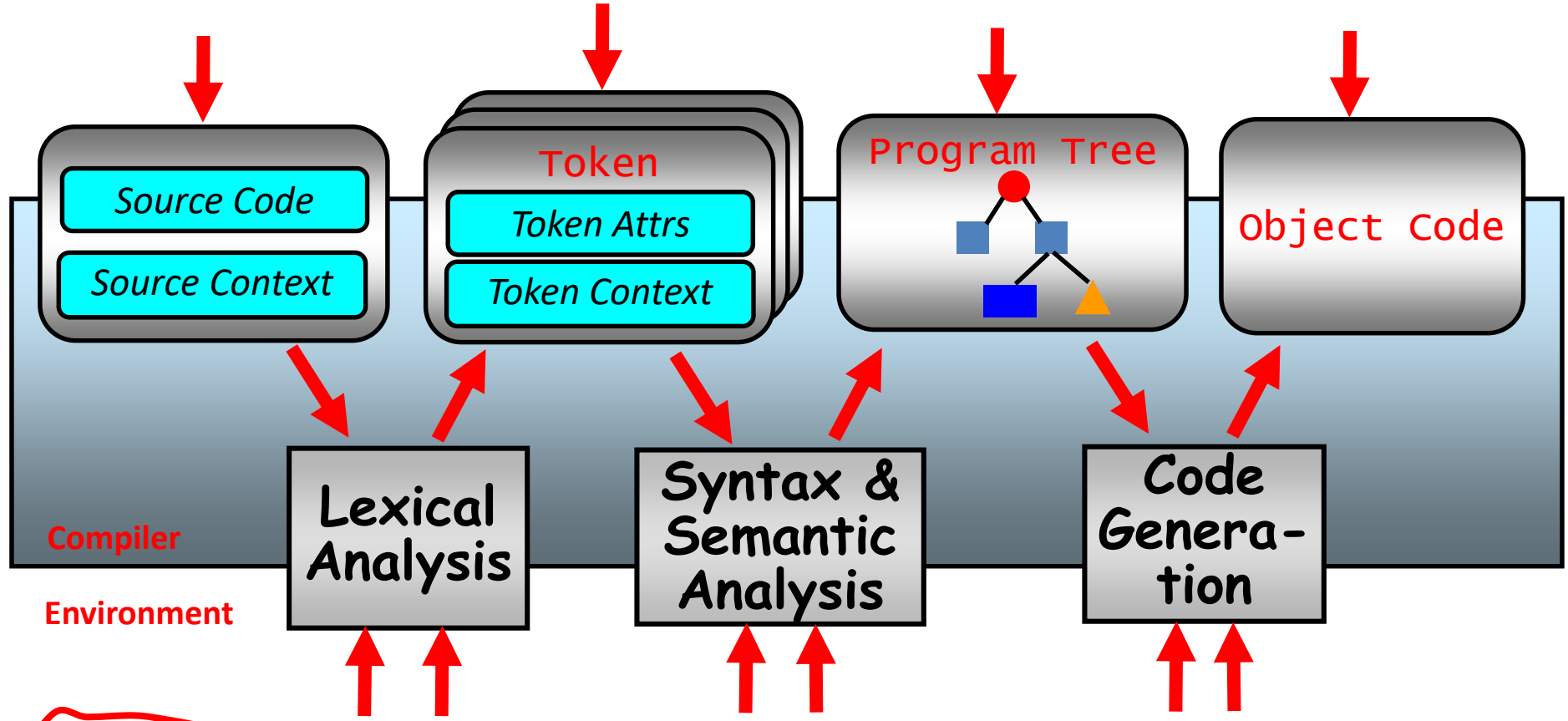
## Example: Semantic information from the C# compiler using by Visual Studio

```
// For that, taking tokens from the buffer until STRINGIZE_END appears.

Token previous = null;
Span begin = null, end = null;
string literal = "";
while ( true )
{
    token = this.get(); // Recursion 'cause we need macro processing
    forget();

    if ( begin == null )
        begin = token.sp
    if ( token.code ==                                  D ) break;

    if ( previous != n
        literal += " ";

    literal += token.image;
    previous = token;
    end = token.span;
}
if ( literal == "" )
{
    // There was an empty sequence STRINGIZE, STRINGIZE_END.
```

commentSpan
modifySpan
span

Span TokenBase.span
Textual coordinates of the original token (line/position of the beginning and the end of the token).

# Compiler Integration (3)




Compiler as a collection of resources

# Compiler Integration (4)