

Compiler Construction: Introduction

The Evolution of the Compiler Architecture

Eugene Zouev
Spring Semester, 2023
Innopolis University

Compiler Architecture: Outline

- Compilation task.
Compilation in a narrow & in a wide sense
- Advanced architecture, or How to turn compiler inside out?
- Program semantic representation
- Compiler integration into an IDE

Main Problems

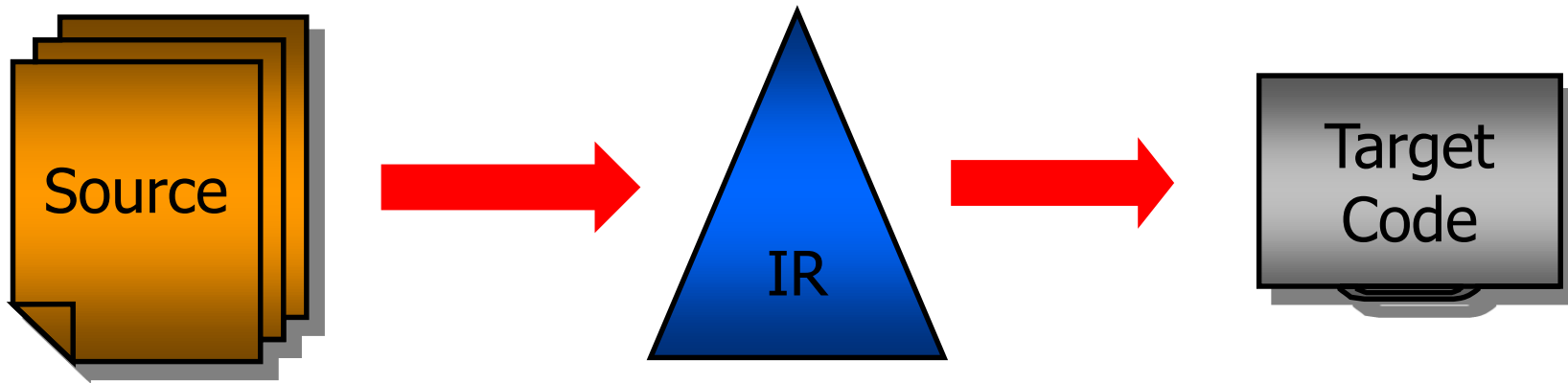
- Unsatisfactory language design 😞😞
- Efficient code generation
- Program analysis
Program understanding
- Integration compilers into integrated development environments (IDEs)

Main Problems

- Unsatisfactory language design 😞😞
Out of the scope of the talk...
- Efficient code generation
Is solved by conventional compilers
- Program analysis
Program understanding
- Integration compilers into integrated development environments (IDEs)

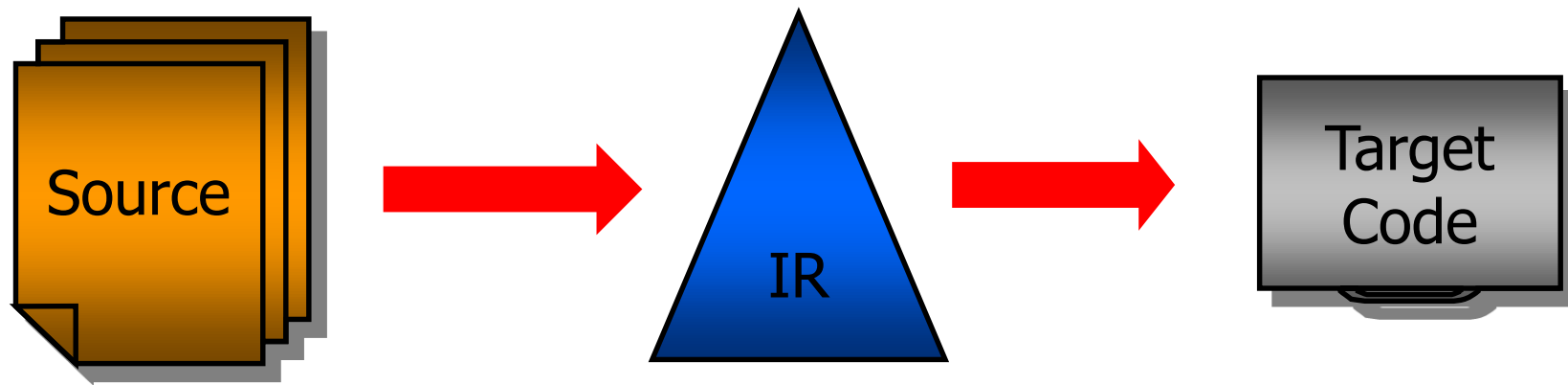
Compilation in a "narrow sense"

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



Compilation in a "narrow sense"

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



This is my
key point

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

This is my
key point

Why compilation “in a 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.
- ...

This is my
key point

Why compilation “in a 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.
- ...

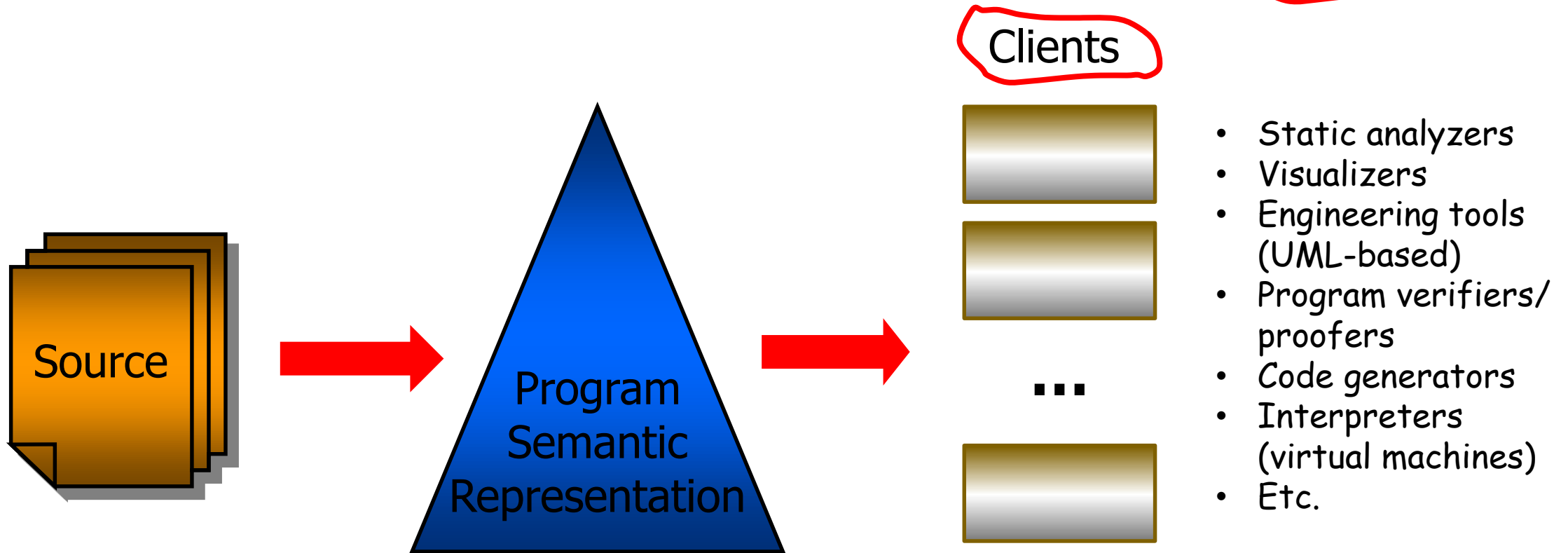
This is my
key point

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.

This is my key point



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 possibilities for improving their functionality.
- By creating **open infrastructures (APIs)** providing access to programs' semantics.

Open Infrastructures: Related Projects

ASIS

Ada Semantic Interface Specification (for Ada95):
the ISO standard

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

An open compiler infrastructure for source-to-source
transformations

Pivot (for C++)

B.Stroustrup & Dos Reis; "General infrastructure for
transformation and static analysis of C++ programs"

CCI (for .NET)

Program infrastructure for compiler construction for .NET

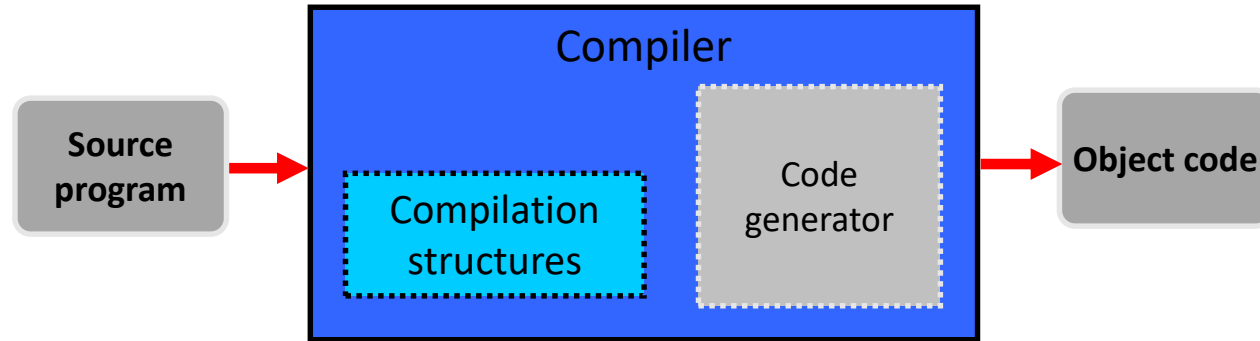
llvm/clang

C/C++ compiler & API for program analysis

Roslyn for .NET

The Evolution of Compiler Architecture

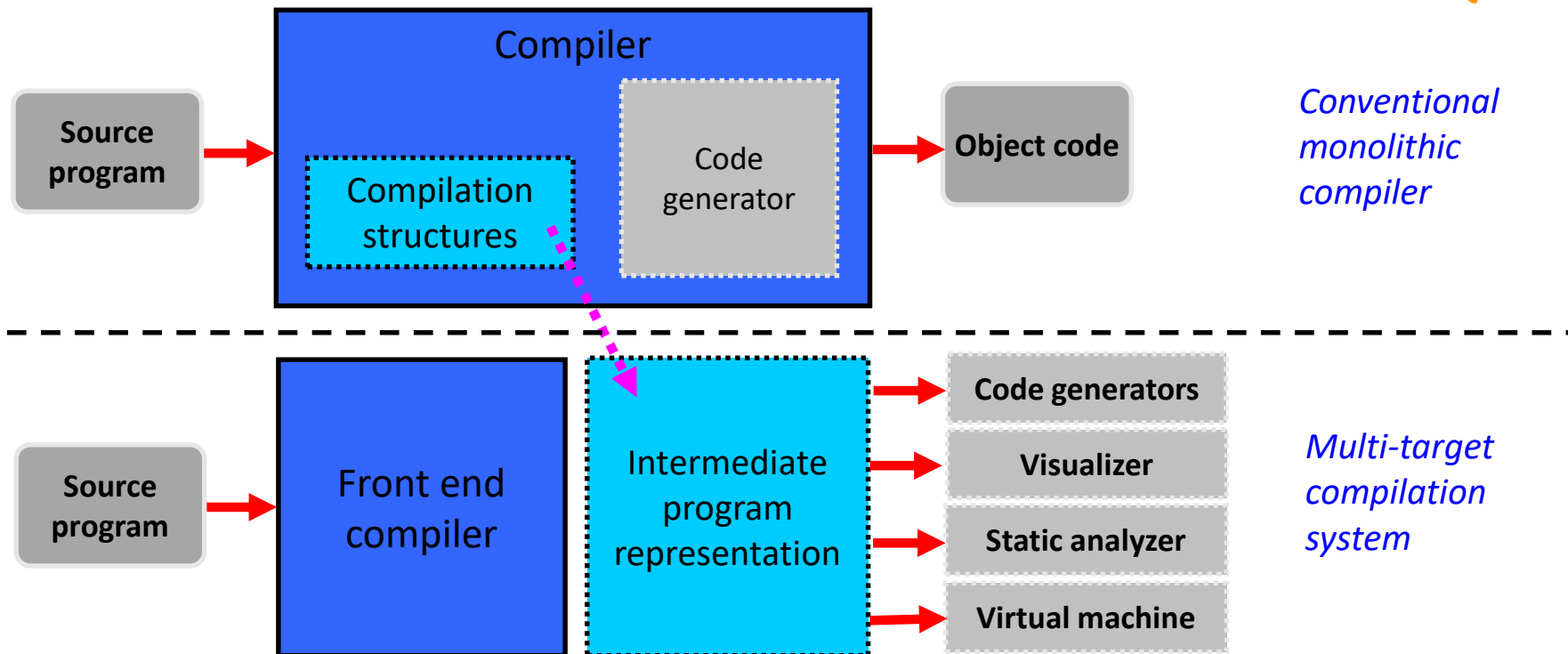
(on one slide)



*Conventional
monolithic
compiler*

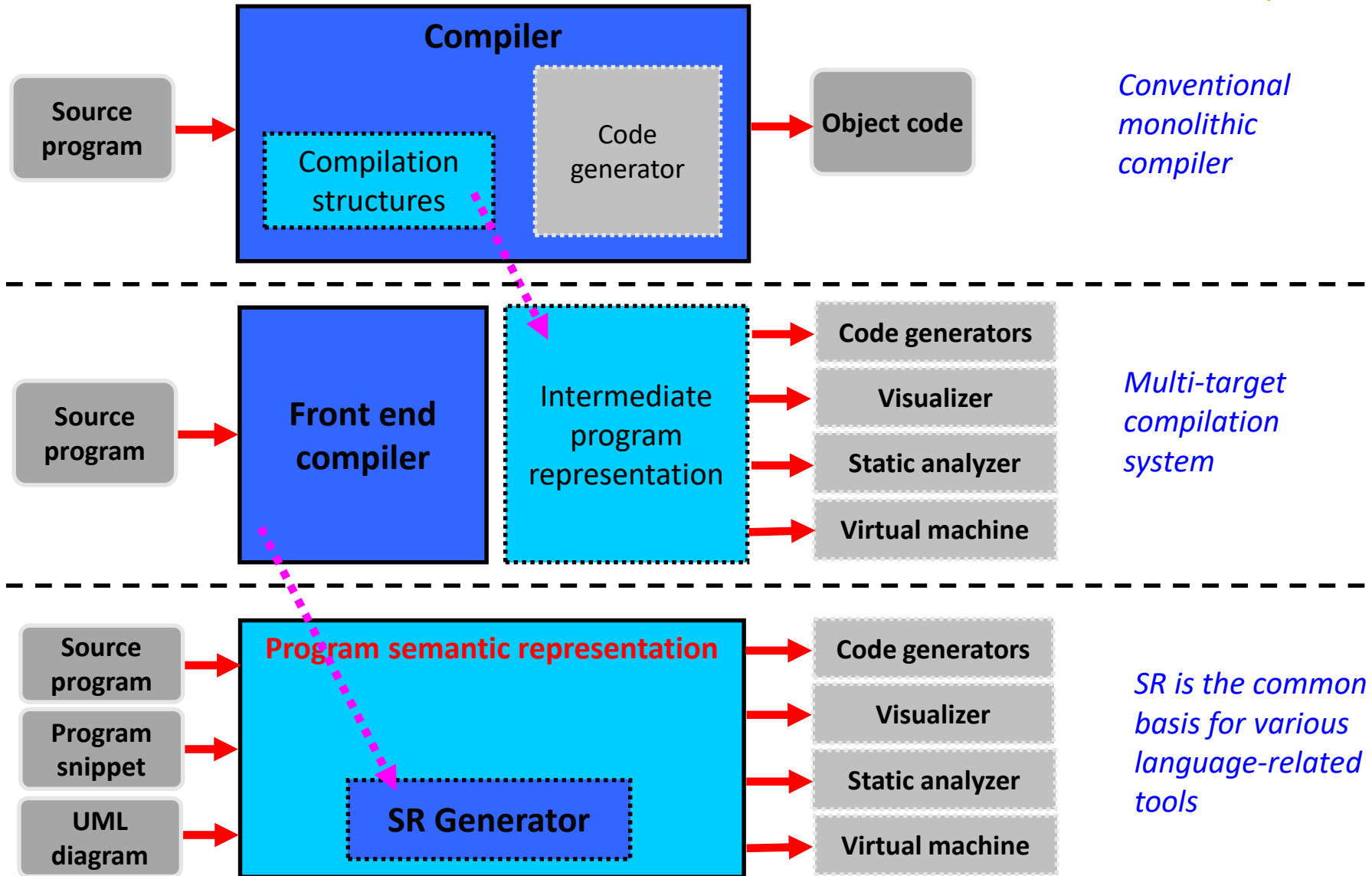
The Evolution of Compiler Architecture

(on one slide)

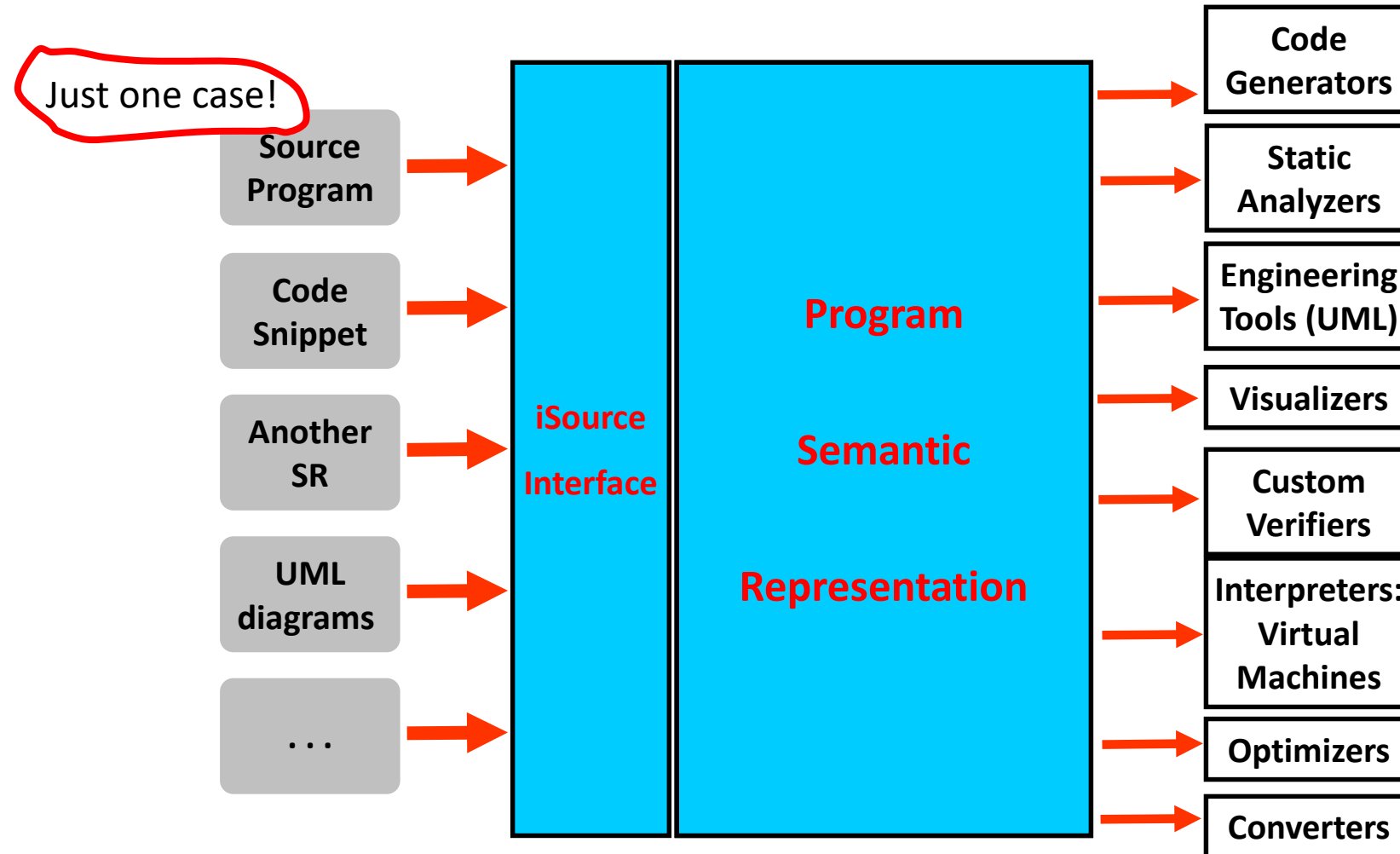


The Evolution of Compiler Architecture

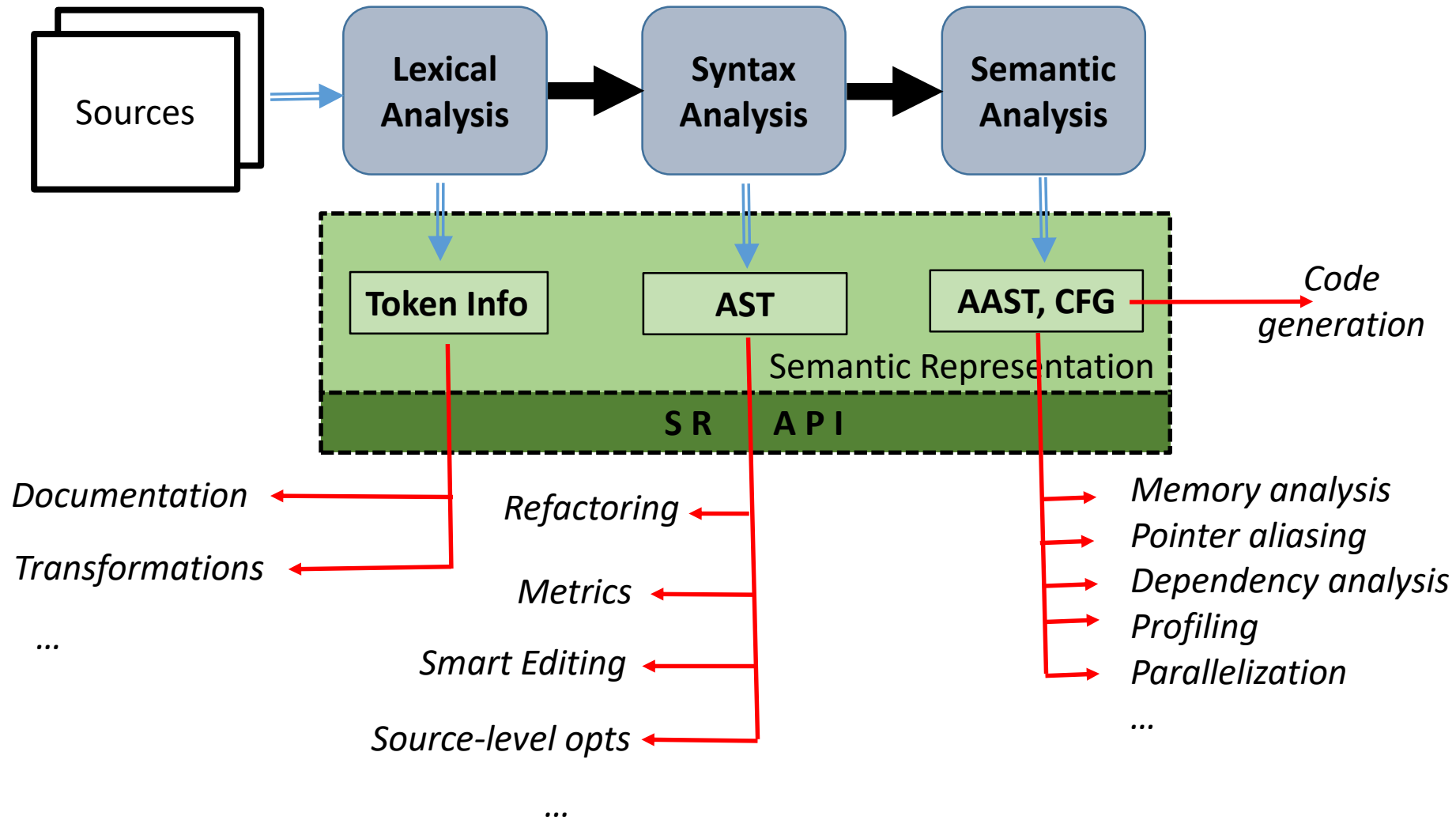
(on one slide)



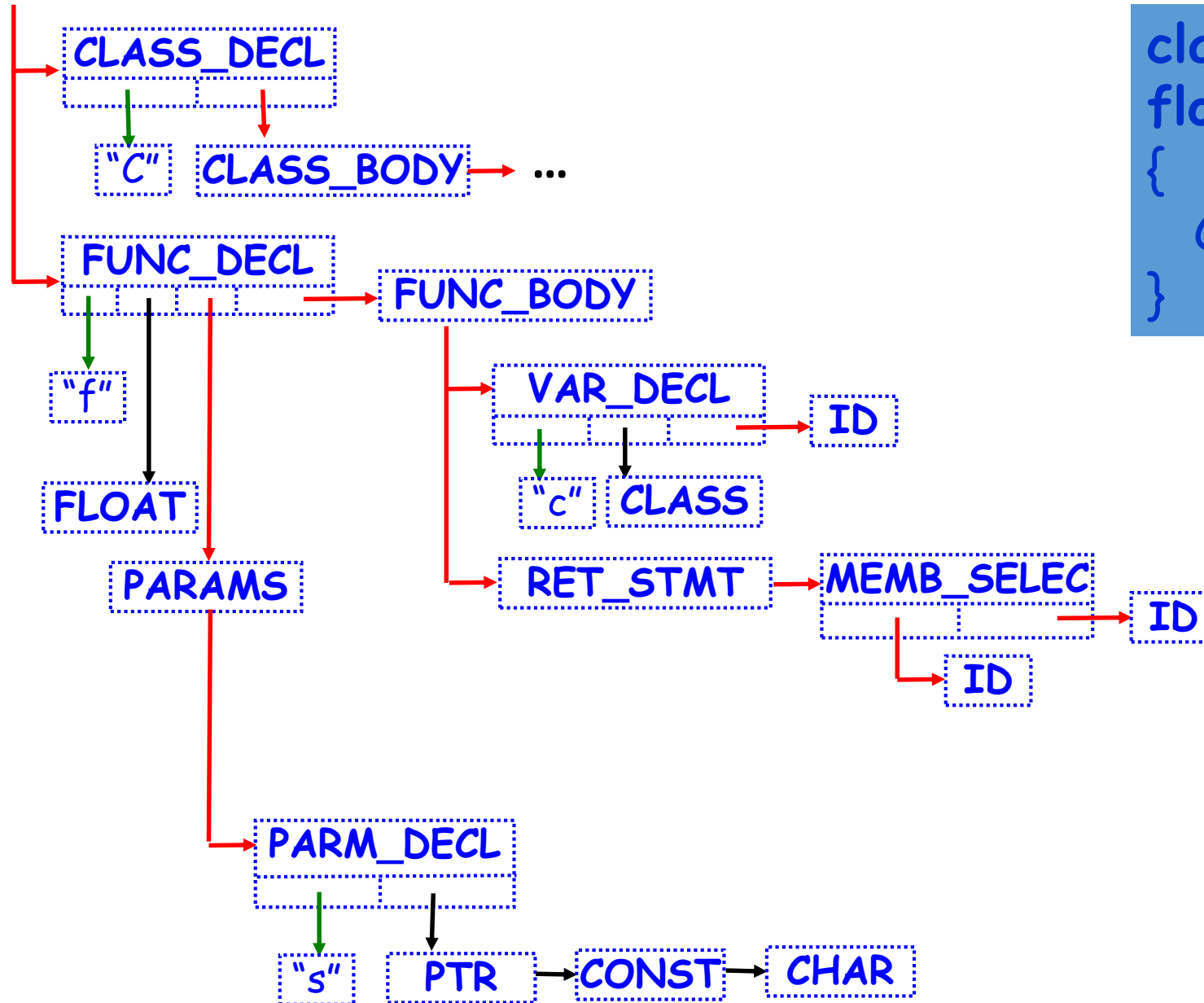
Compilation in a "wide sense": the conceptual view



Semantic Representation in More Details



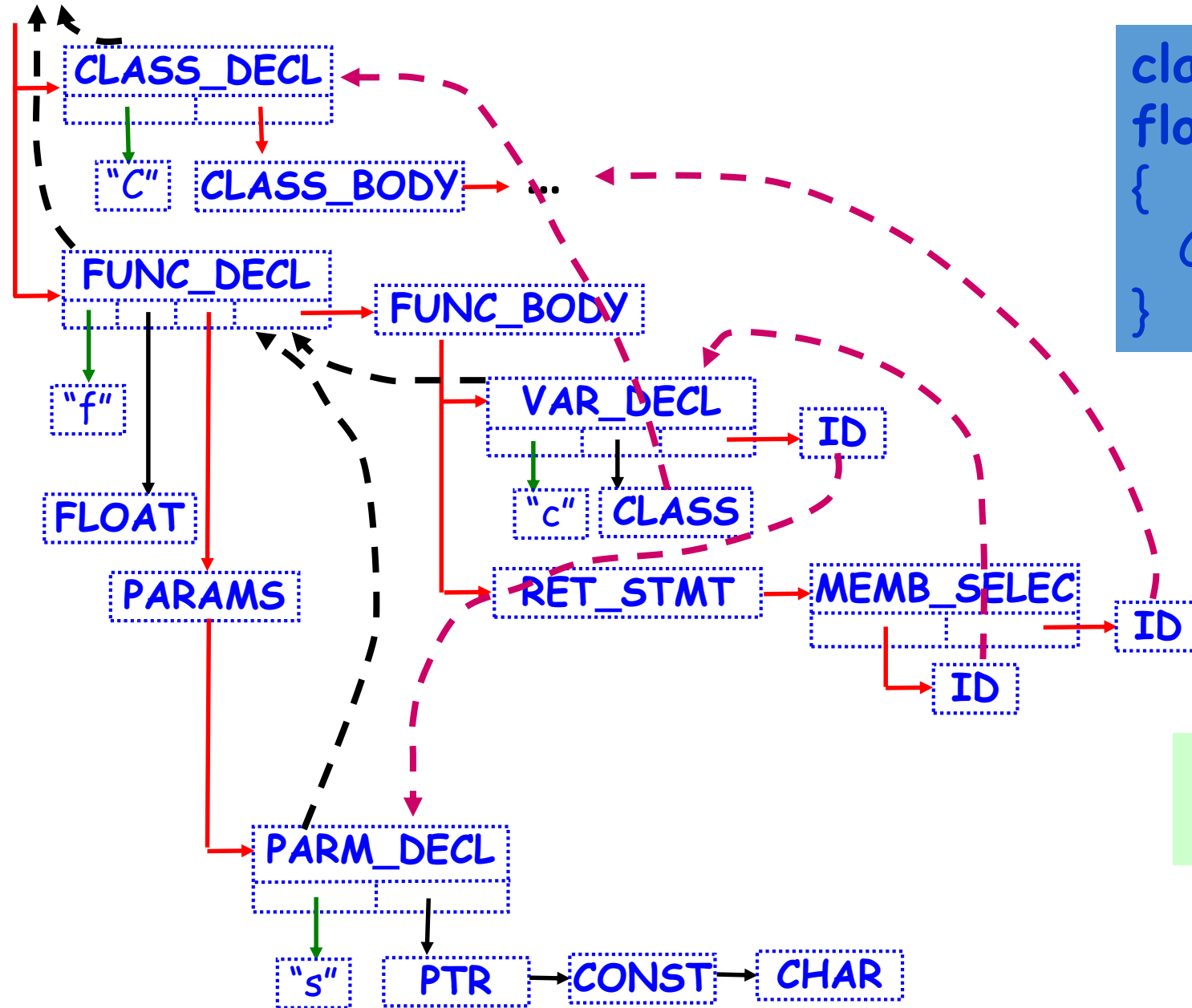
Annotated AST as the Basis of SR (1)



```
class C { ... };
float f(const char* s)
{
    C c(s); return c.m;
}
```

- Structural relations
- Type information
- Various attributes

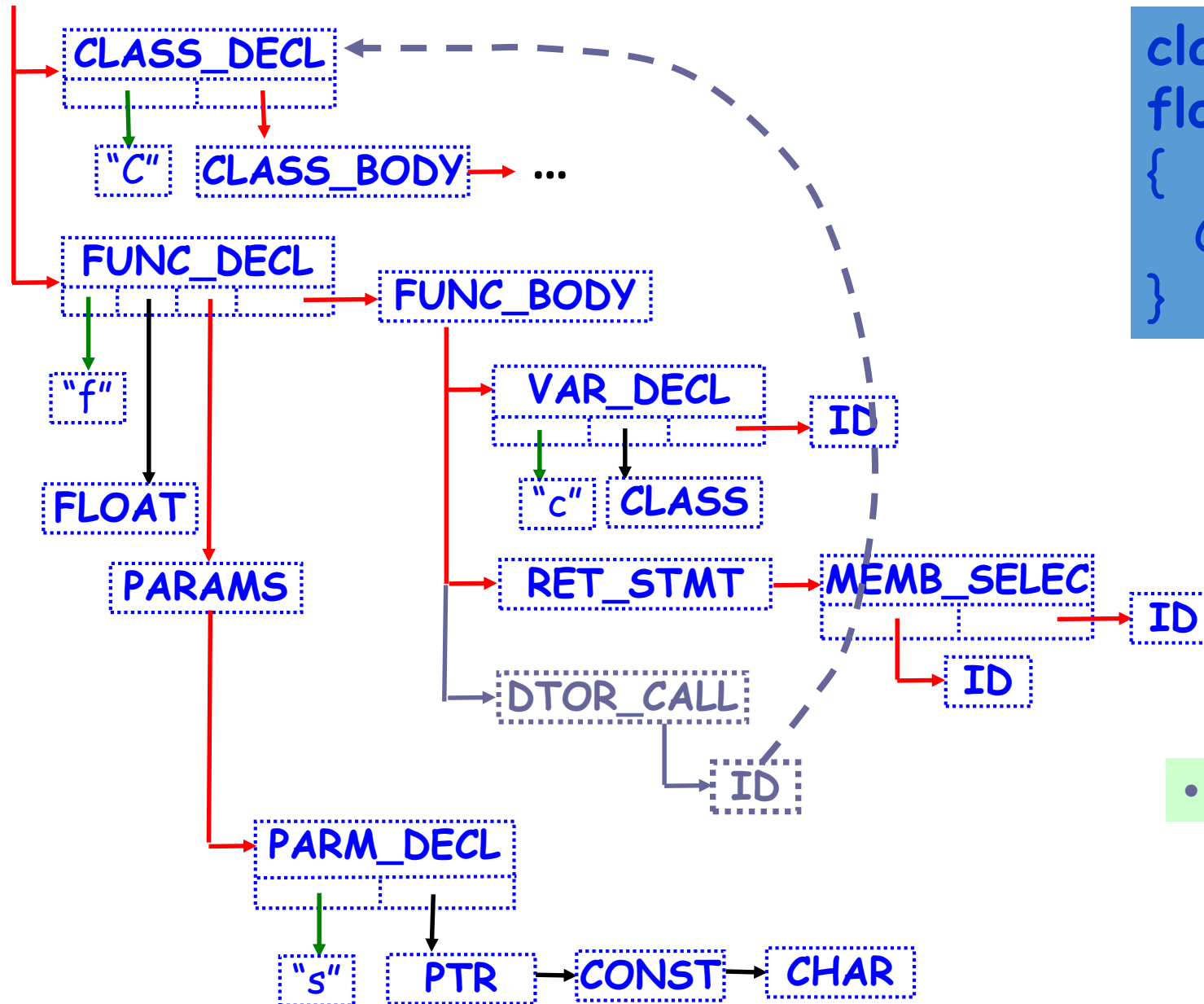
Annotated AST as the Basis of SR (2)



```
class C { ... };  
float f(const char* s)  
{  
    C c(s); return c.m;  
}
```

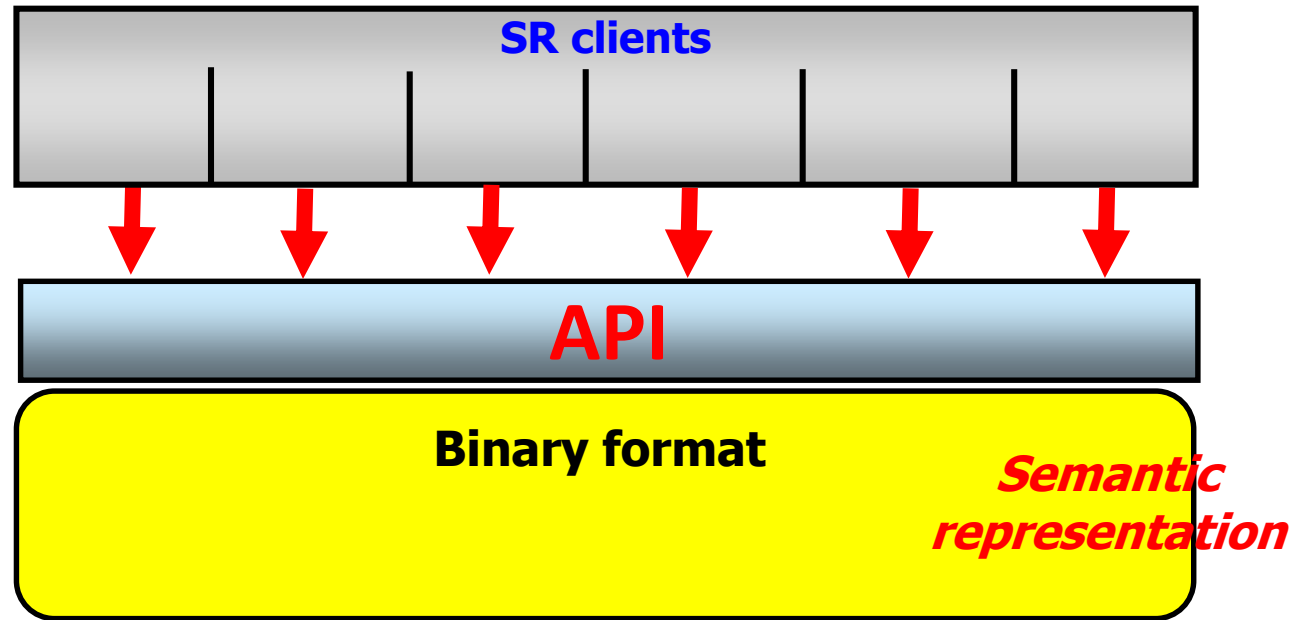
- Semantic relationships
- Scoping rules

Annotated AST as the Basis of SR (3)

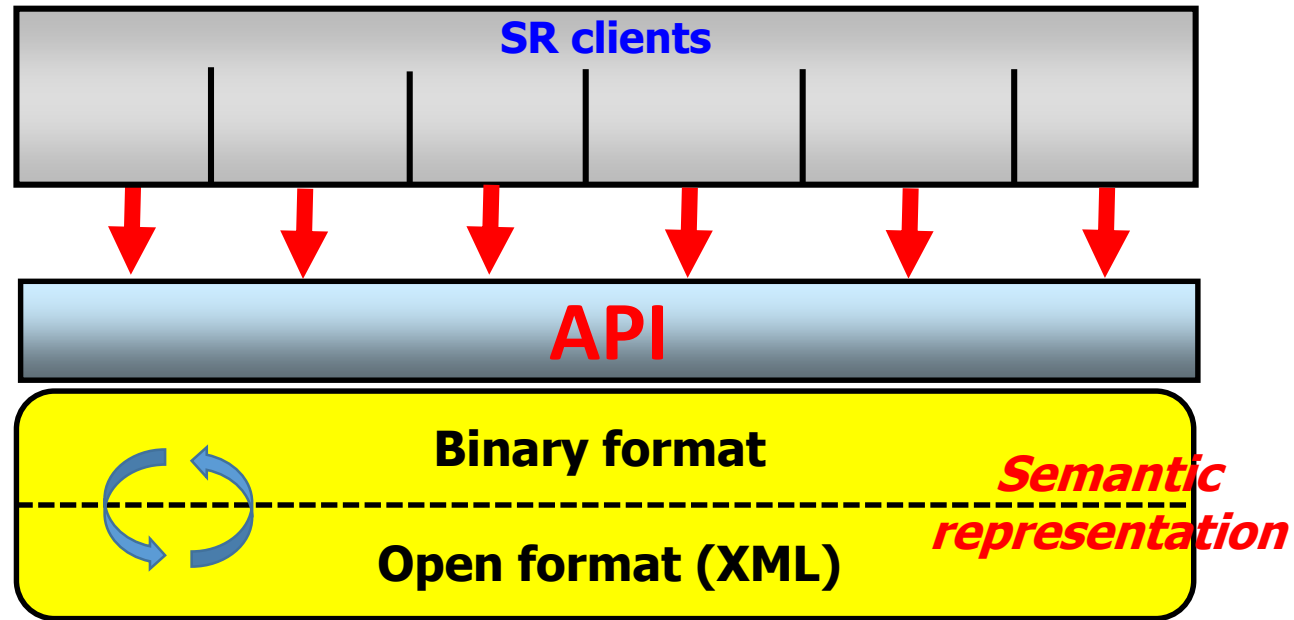


```
class C { ... };
float f(const char* s)
{
    C c(s); return c.m;
}
```

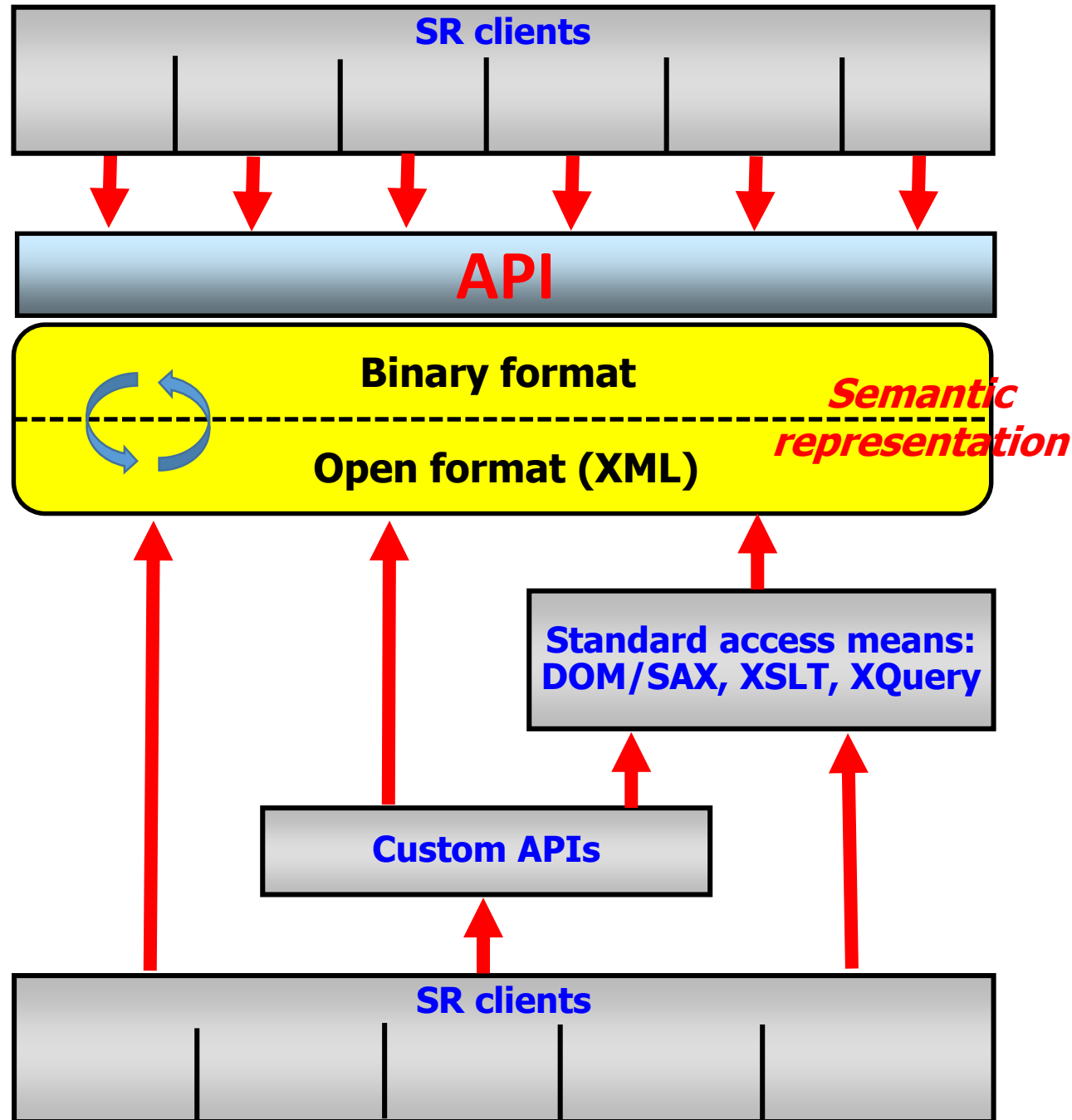
- Hidden semantics



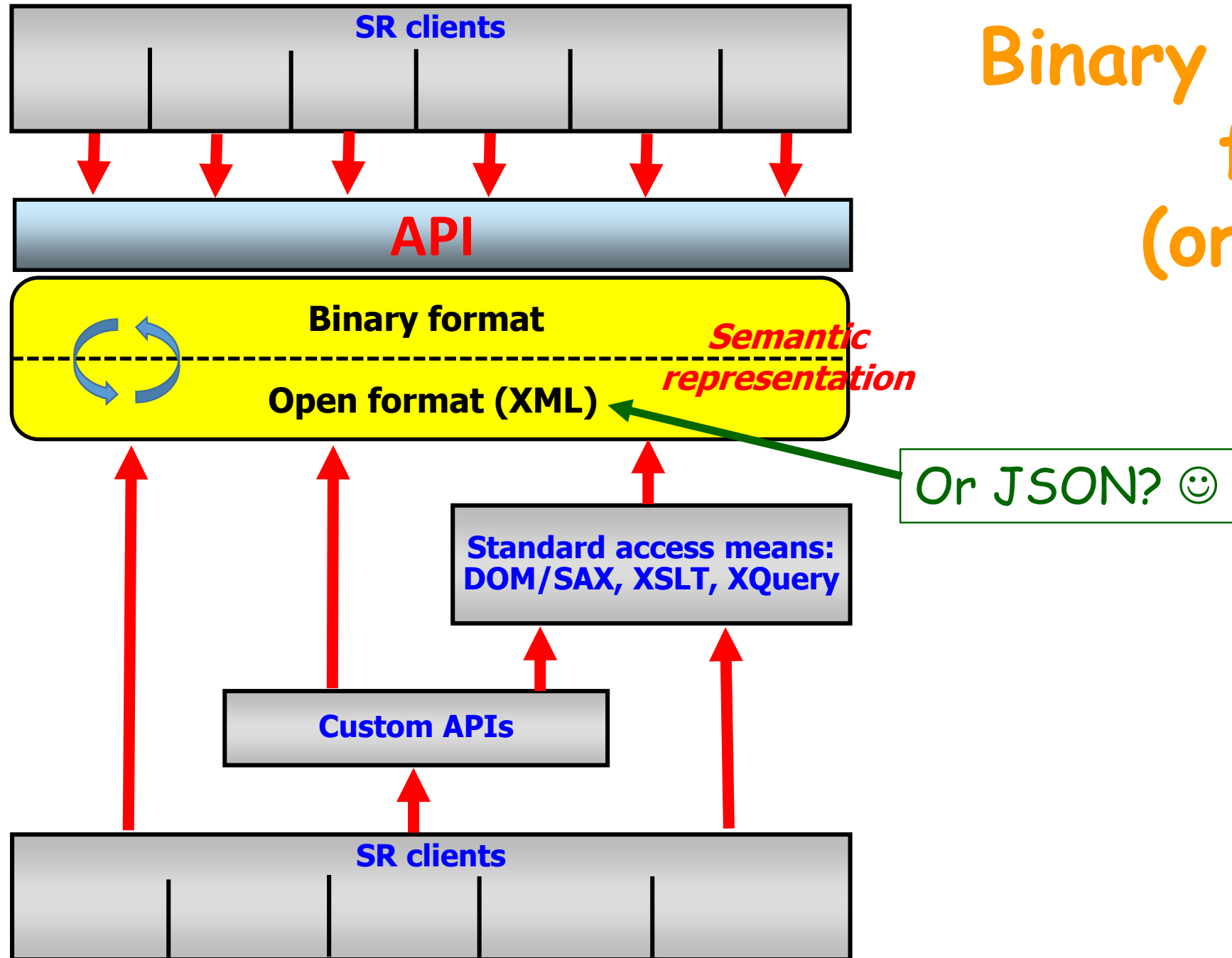
Binary or open
format?
(or both?)



Binary or open
format?
(or both?)



Binary or open
format?
(or both?)



Binary or open
format?
(or both?)

Or JSON? ☺

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 manipulating

Why XML?

← Or JSON? 😊

- 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 manipulating

All this applies to
JSON as well 😊

SR example in XML format

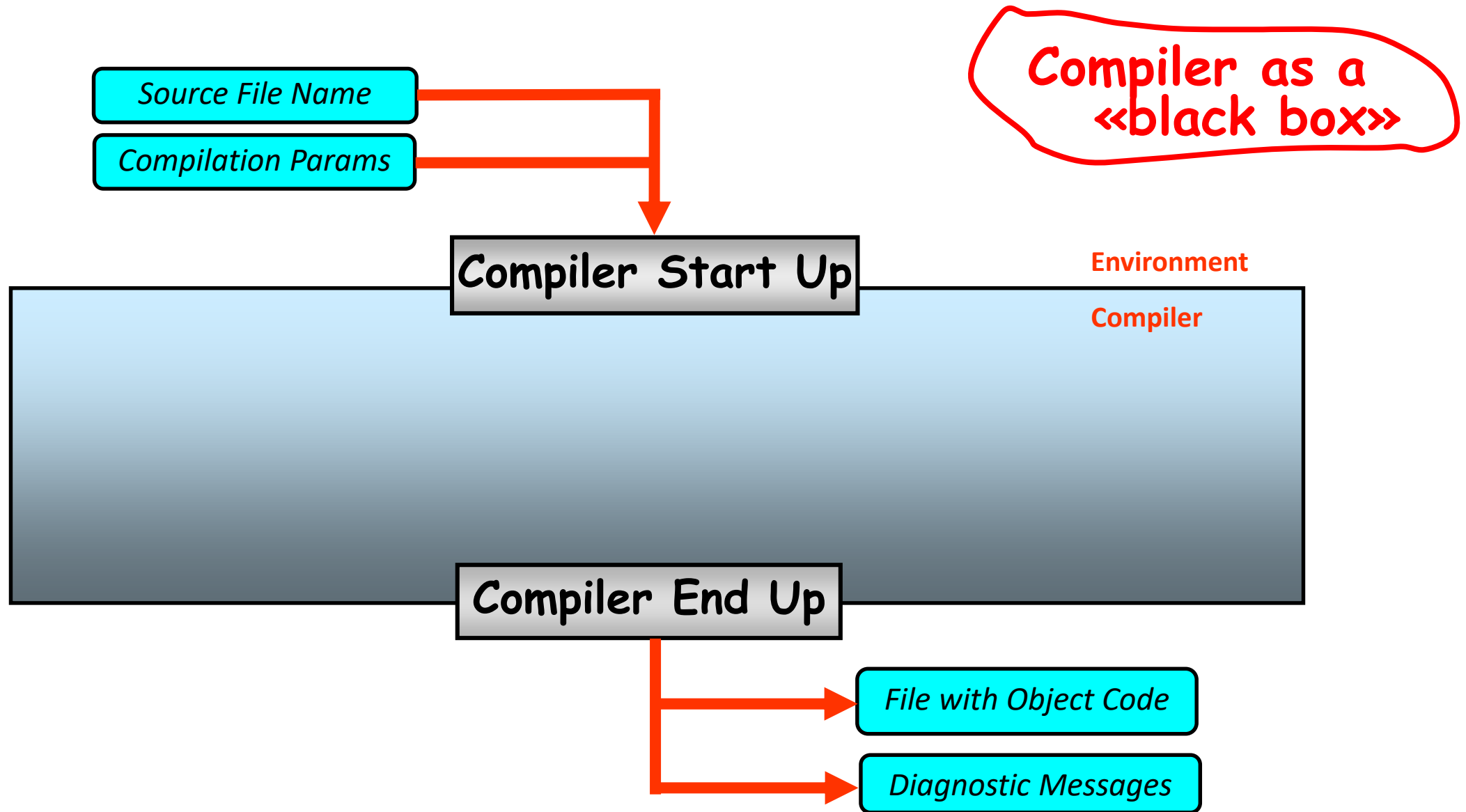
```
<while-statement ln="1" col="1">  
  <condition>  
    <expression ln="1" col="7"> ... </expression>  
  </condition>  
  <compound-statement>  
    <assignment-expression ln="2" col="4">  
      <name ln="2" col="4">x</name>  
      <expression ln="2" col="9"> ... </expression>  
    </assignment>  
    <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>
```

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

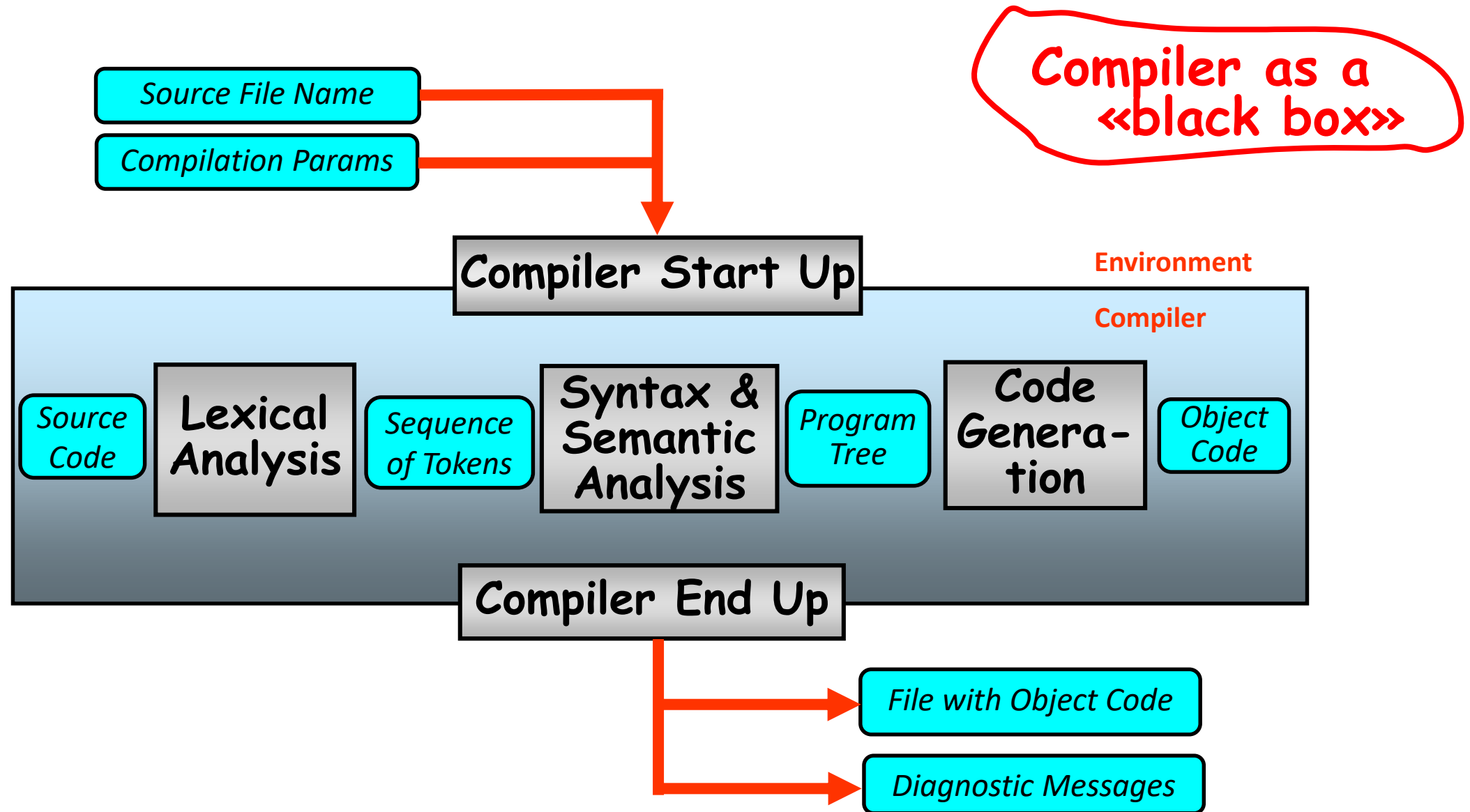
simplified

Compilers in IDEs & Compiler Architecture

Compilation: Conventional Approach



Compilation: Conventional Approach



How the editor knows
that these are types?

Compiler Integration (1)

Example: integration
the C# compiler into
Visual Studio

```
while ( true )
{
    token = s.expect(TokenCode.Identifier);
    var id = IDENTIFIER.create(token);
    result.add(id);
    id.parent = result;

    token = s.get();|
    if ( token.code == TokenCode.Comma ) { s.forget(); continue; }
    break;
}
token = s.expect(TokenCode.RightBracket);
result.setSpan(begin,token.span);

return result;
}

#endregion
```

Compiler Integration (1)

How the editor knows that these are types?

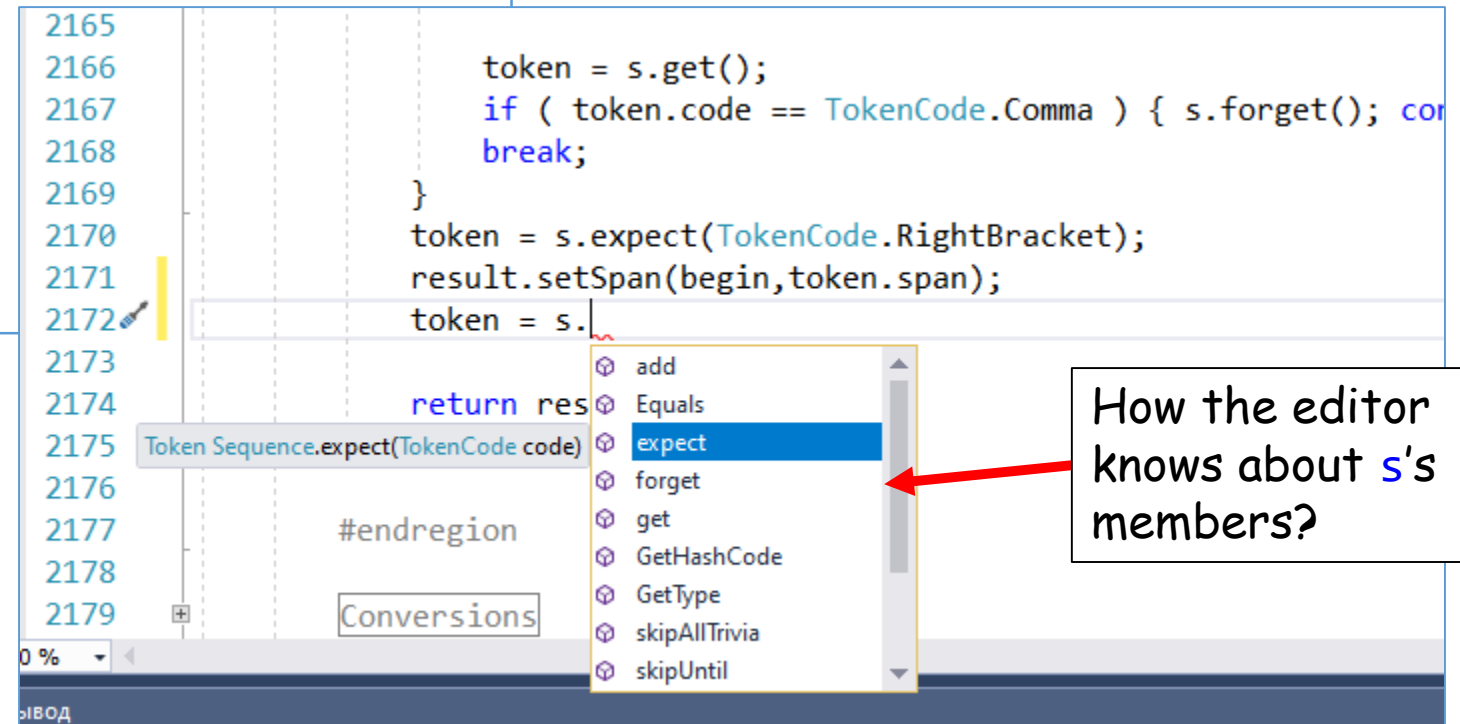
```
while ( true )
{
    token = s.expect(TokenCode.Identifier);
    var id = IDENTIFIER.create(token);
    result.add(id);
    id.parent = result;

    token = s.get();
    if ( token.code == TokenCode.Comma ) { s.forget(); continue; }
    break;
}
token = s.expect(TokenCode.RightBracket);
result.setSpan(begin,token.span);

return result;
}
```

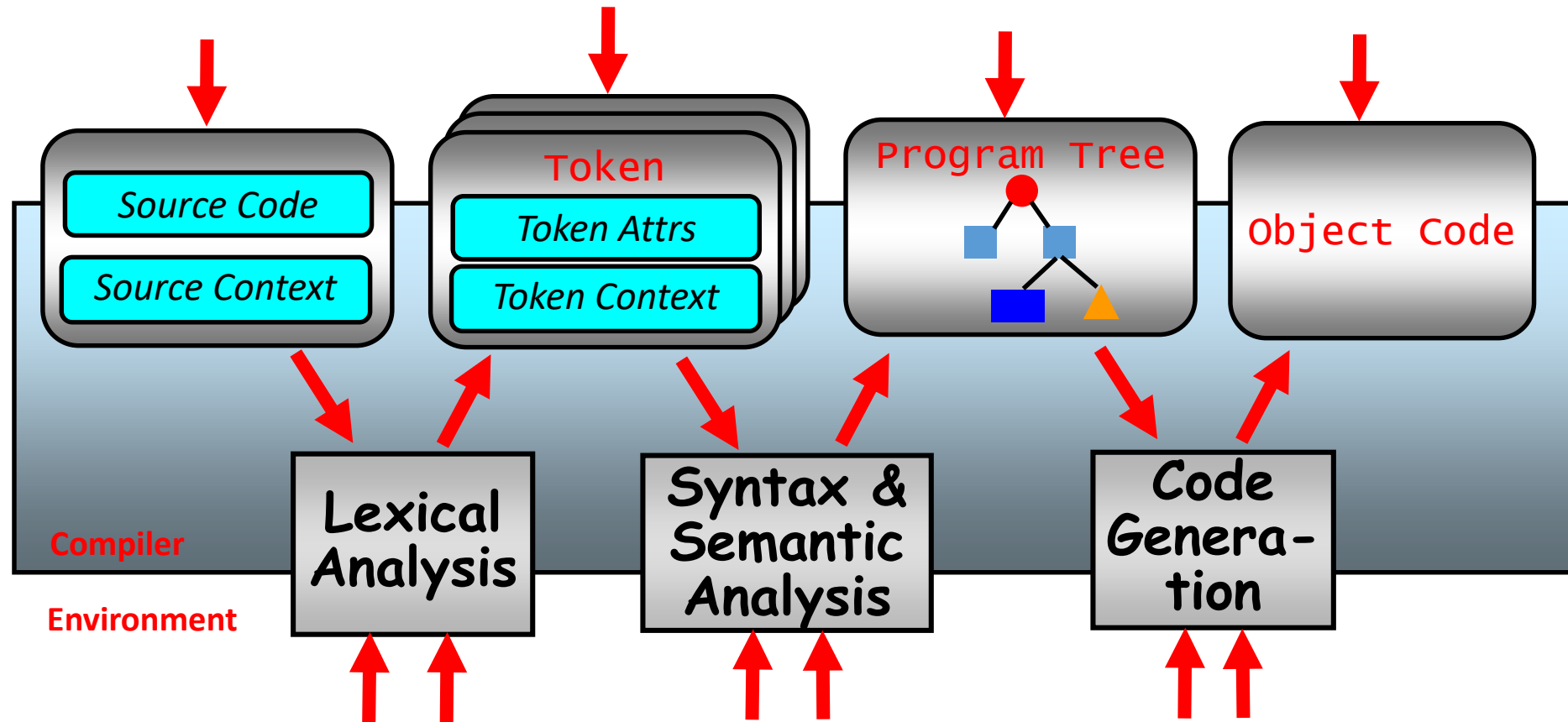
#endregion

Example: integration
the C# compiler into
Visual Studio



How the editor
knows about s's
members?

Compiler Integration (2)



Compiler as a collection of resources

Compiler Integration (3)

