Compiler Design

What is a Compiler?

- A *compiler* is a program that transforms code written in one (source) language into another (target) language
  - Goal: obtain an executable program
  - Typically, from a high-level programming language like C or Java to a low-level language like machine code or JVM bytecode

- *Transpiler*: from high-level language to high-level language, e.g., from Java to Javascript

- *Interpreter*: program that reads the source code and directly executes it, no executable program created

Properties of a Compiler

- *Syntactic correctness*: compiler should accept syntactically valid source code and reject malformed source code
- *Semantic correctness*: behavior of generated target code should match expected behavior of source code
- *Efficiency of output*: target code should be fast and memory efficient
- *Efficiency of translation*: should be fast and memory efficient, even for large source code
- Useful error messages, warnings,...

```
rtmap.cpp: In function `int main()': rtmap.cpp:19: invalid conversion
from `int' to ` std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:19: initializing argument 1 of `std::_Rb_tree_iterator<_Val,
_Ref, _Ptr>::_Rb_tree_iterator (std::_Rb_tree_node<_Val>*) [with _Val =
std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr
= std::pair<const int, double>*]' rtmap.cpp:20: invalid conversion from
`int' to ` std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:20: initializing argument 1 of `std::_Rb_tree_iterator<_Val,
_Ref, _Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*) [with _Val =
std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr
= std::pair<const int, double>*]'
```

Languages

- Of course, writing a correct compiler requires that we know
  - the *syntax* of the source language, i.e., "How does source code look like?"
    - Sometimes difficult. Example C:

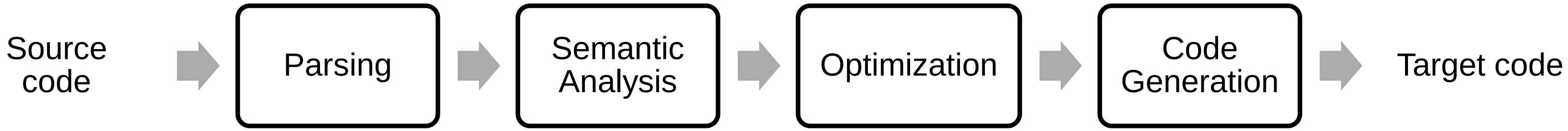
      Does a+++b mean (a++)+b or a+(++b) ?

  - the *semantics* of the source language, i.e., "What does the code mean?"
    Example: Is the function f() called in this code if a==false?

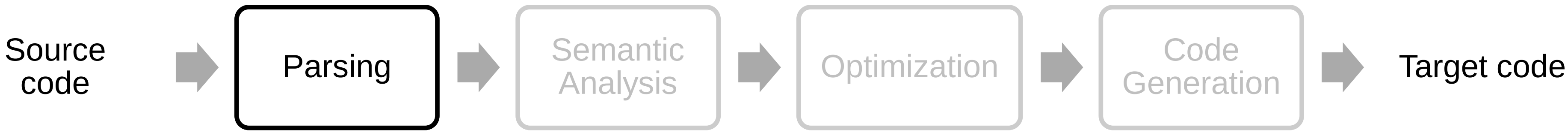
    ```
    if(a && f(x)==2) { ....
    ```

Compiler pipeline

- Languages and compilers have been intensively studied in the past 60 years and theories and best practices have been developed
- In many compilers, the translation happens in multiple phases

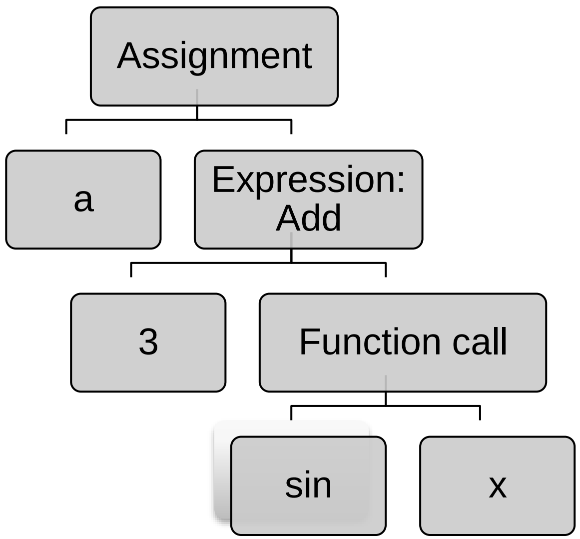


Parsing (Syntax Analysis)



- A parser transforms the source code into a data structure that is easier to handle for semantic analysis, typically a tree ("Syntax tree")

a = 3 + sin(x) →



- Parsing also checks whether the source code follows the syntax of the source language. For example, "a 3 = + sin)x" gives a syntax error

Lexing

Source code → **Lexing** → Parsing → Semantic Analysis

- Often, the parsing is preceded by another phase, the lexing
- Lexing splits the source code into a sequence of symbols, getting rid of whitespaces, new lines, etc.

*<tab>*a=*<space><space>*3*<newline>*
+sin(x)*<newline>*

→

Symbol 1: Identifier “a”
Symbol 2: Assignment operator
Symbol 3: Number “3”
Symbol 4: Arithmetic operator “+”
Symbol 5: Identifier “sin”
Symbol 6: Opening parenthesis
Symbol 7: Identifier “x”
Symbol 8: Closing parenthesis

- The *lexer* (also called *scanner*) is dump and simple
- Advantages: Simplifies the parsing, lexer is very optimized

Not only compilers...

- Lexing and parsing is also used outside compilers
  - In interpreters

```
Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 b
Type "help", "copyright", "credits" or "license" for more information.
>>> 3*4
12
```

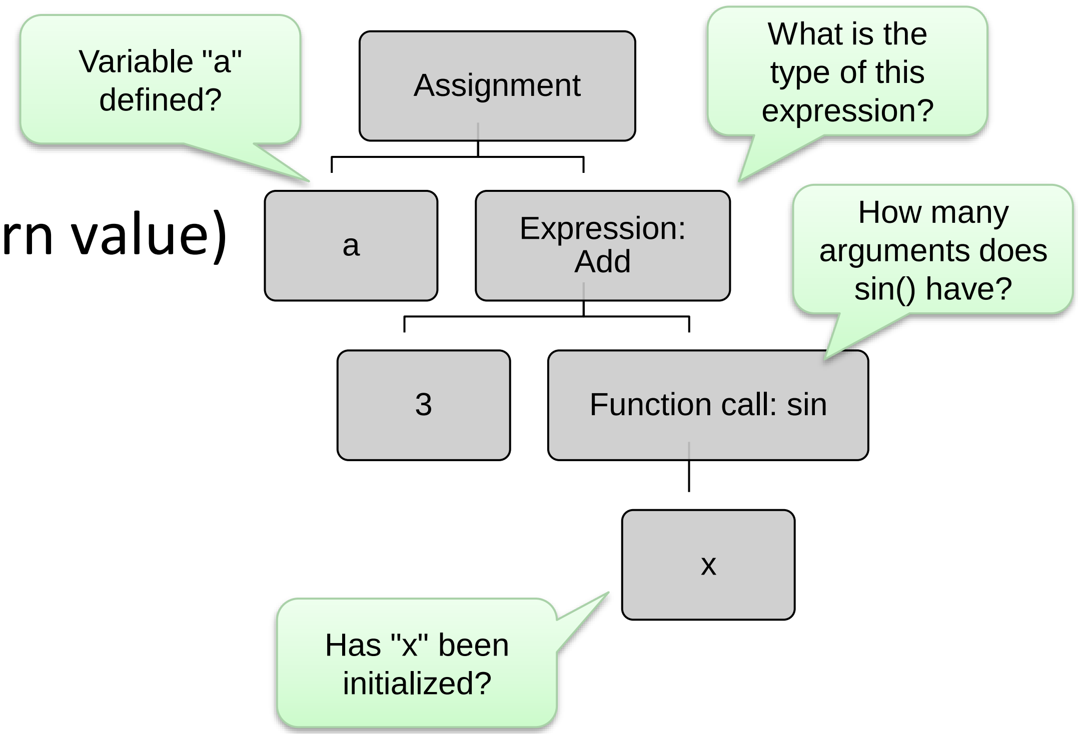
  - To read structured file formats (XML, HTML, JSON,...)
  - For syntax highlighting in editors

```
int n = matrix.length;
// range
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (matrix[i][j] <= 0 || matrix[i][j] > n*n) {
            return false;
        }
    }
}
```

Semantic Analysis

Source code → Parsing → **Semantic Analysis** → Optimization → Code Generation → Target code

- Takes the syntax tree from the parser and analyzes it:
  - Learns which new types, variables, functions are defined in the code
  - Type checking
  - Flow checking (functions without return value)
  - ...



Optimization

Source code → Parsing → Semantic Analysis → **Optimization** → Code Generation → Target code

- Very important for achieving good performance
- Many optimizations! A few of them:
  - Precalculate results during compile-time

```
int x = 3;
int y = 7*x;          // <- replace by y=21
```

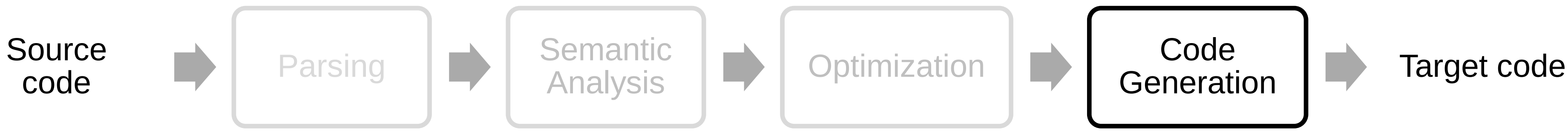
  - Eliminate common expressions

```
int x = 3+z;
int y = (3+z)*2;    // <- replace by y=x*2
```

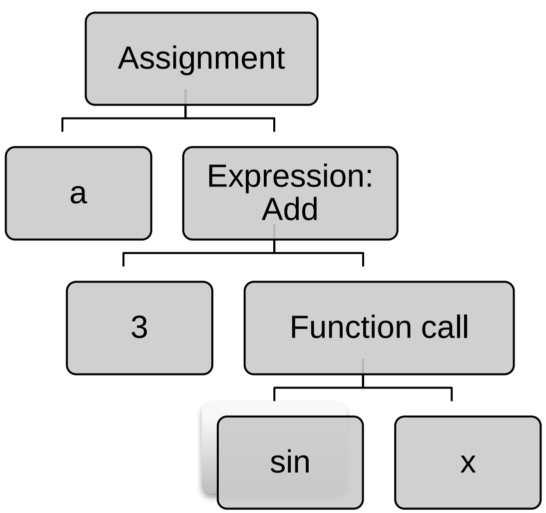
  - Invariant code motion:

```
for(int i=0;i<10;i++) {
        int x = 3;     // <- move out of loop
```

Code Generation



- Generate target code from the optimized syntax tree
  - Assembly or machine code for binary executables
  - Bytecode for VMs (e.g., Java)



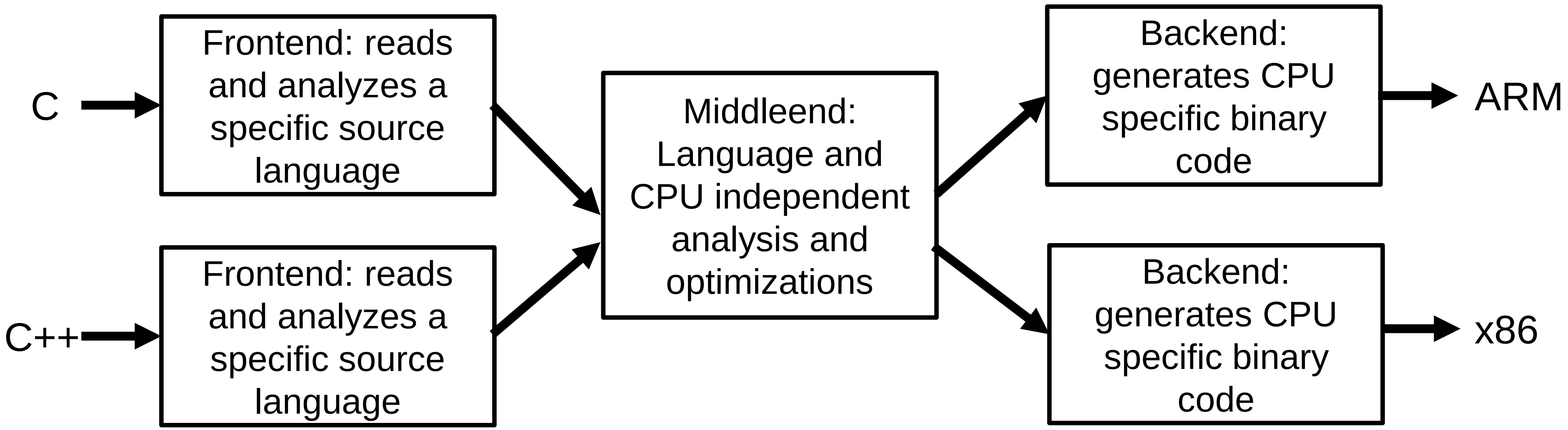

```
push 0x10000      // x
call 0x2027A      // sin
loadconst 3
add
store 0x10004     // -> a
```

- Requires selecting the best sequence of instructions for the target CPU or VM

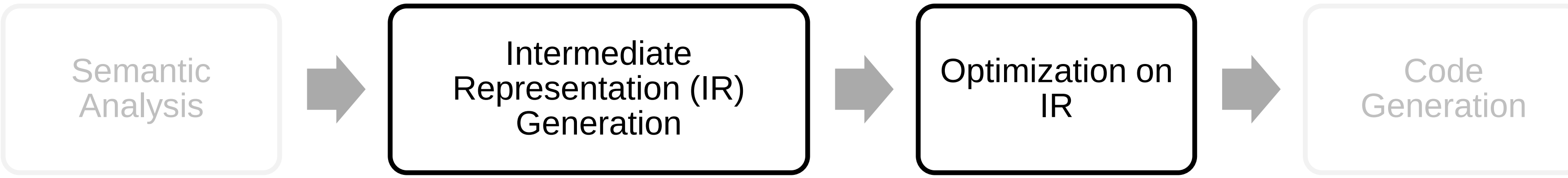
Compiler design

- The pipeline described so far is useful for compilers with specific source and target language:

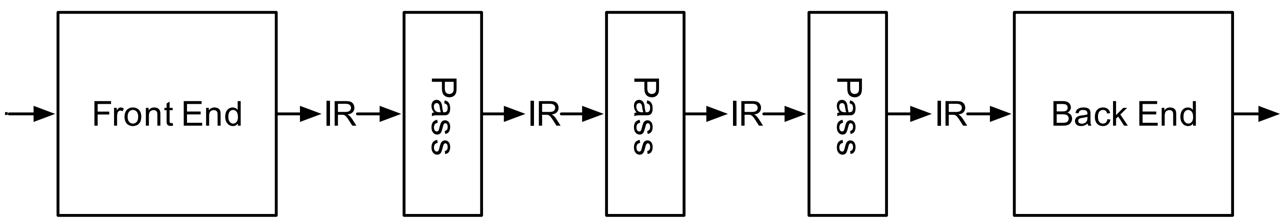
- The three-stage compiler architecture is more flexible:



Intermediate Representation in Three-Stage Architecture

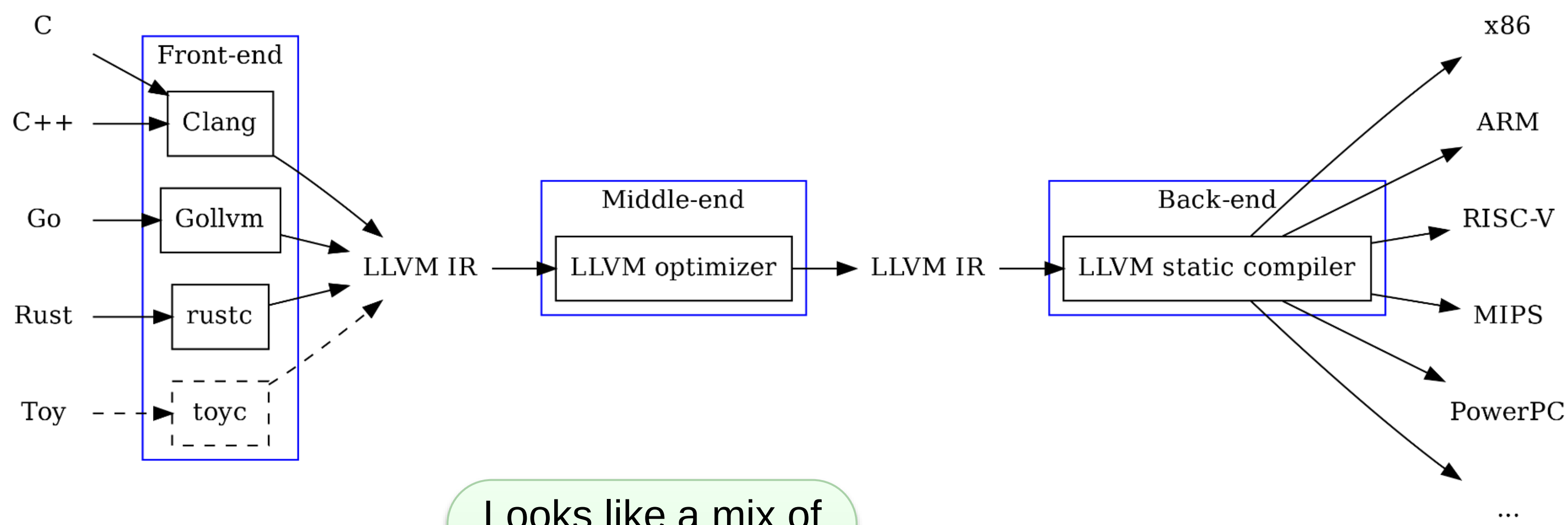


- Compilers like gcc transform the syntax tree into an intermediate representation (IR) before applying complex optimizations
- Advantage: optimizations are independent of source language and target language
- Idea: all the different optimizations are transformations of an IR to another IR (of course without changing the meaning of the compiled program)



Source: https://www.cs.cornell.edu/~asampson/blog/llvm.html

Example: LLVM compiler framework




```
int f(int a, int b) {
    return a + 2*b;
}

int main() {
    return f(10, 20);
}
```

Looks like a mix of C and assembly. But does not depend on specific CPU

```
define i32 @f(i32 %a, i32 %b) {
    %1 = mul i32 2, %b
    %2 = add i32 %a, %1
    ret i32 %2
}

define i32 @main() {
    %1 = call i32 @f(i32 10, i32 20)
    ret i32 %1
}
```

Source: https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/