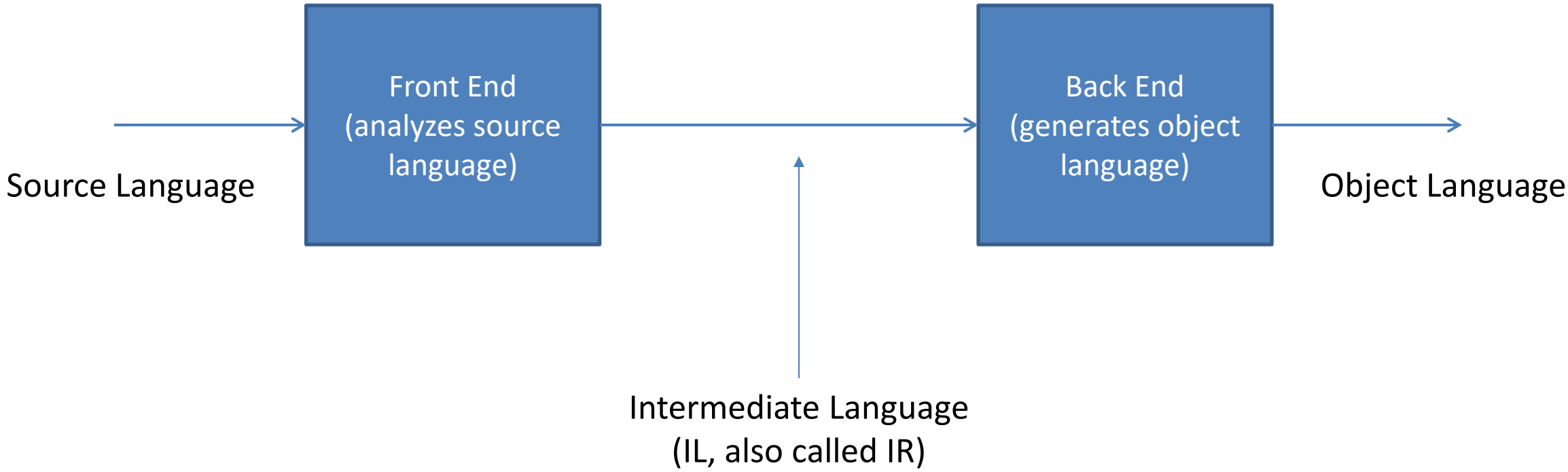
# Compiler Design

Peter Chapin
Vermont State University

# Why Study Compilers?

- Somebody has to write them. Why not you?
- *Know your tools!*
  - Compilers are fundamental. The more you know about them, the more effectively you can use them.
- Domain Specific Languages
  - Compiler technology is useful in surprising ways.
    - Processing complex configuration and command languages.
    - Transforming complex file formats.

# The Pipeline



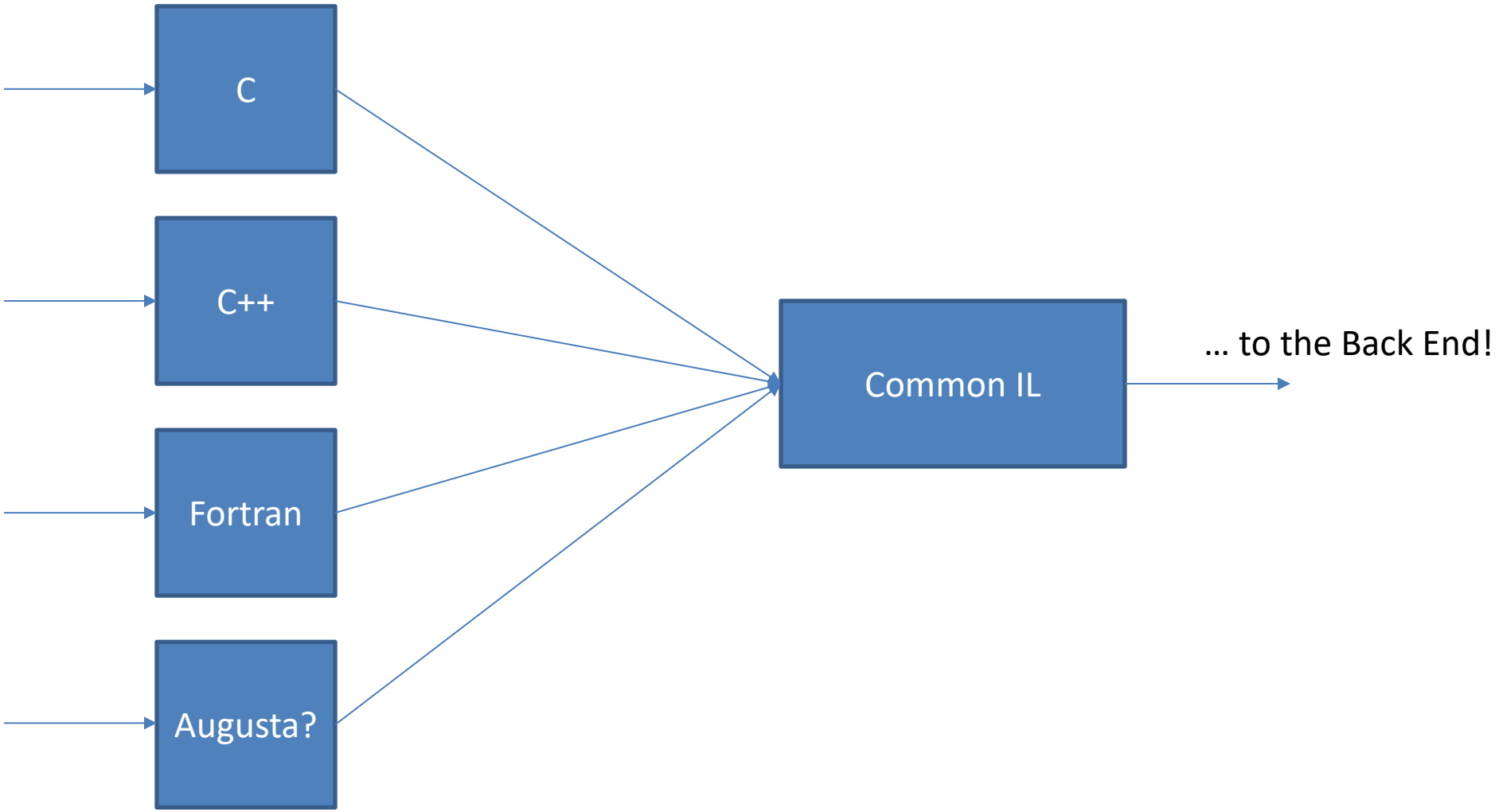
# Language Levels

- Source Language
  - Usually high-level, abstract.
  - e.g., C, Java, etc.
- Object Language
  - Usually low-level, concrete.
  - e.g., Machine language, assembly language, C?
- Intermediate Language
  - Easy for the front end to produce, easy for the back end to consume

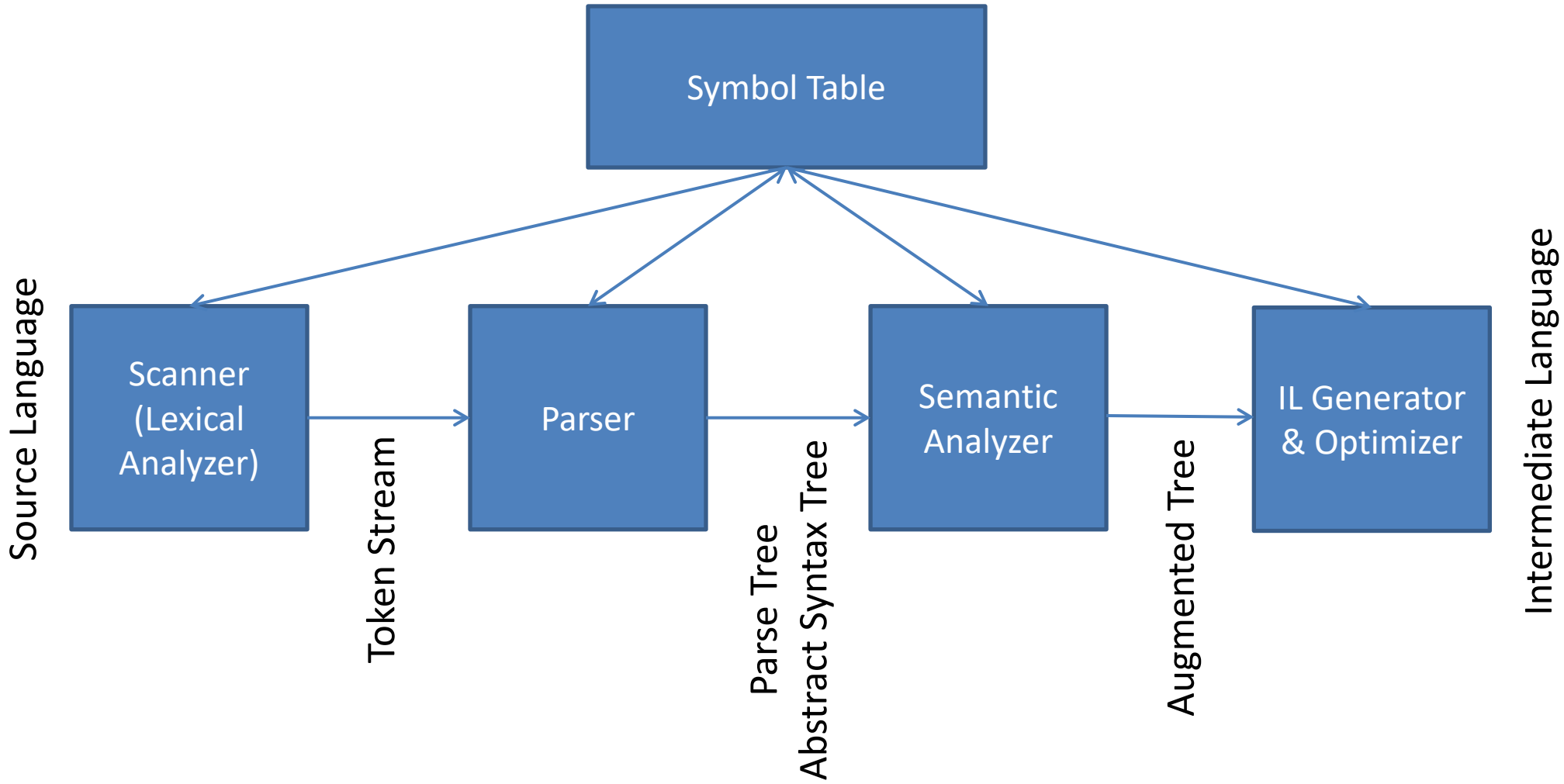
# Front End

- The front end knows the source language
  - Change the front end to compile different source languages. As long as they all generate the same IL, they can use a common back end.
  - gcc. The "GNU Compiler Collection"
    - C, C++, Objective-C, Fortran, Java, Ada, Go
    - All use the same code generator/back end (in theory).
  - Open Watcom
    - C, C++, FORTRAN
    - All use the same code generator/back end (in theory).

# Multiple Front Ends



# Front End Pipeline



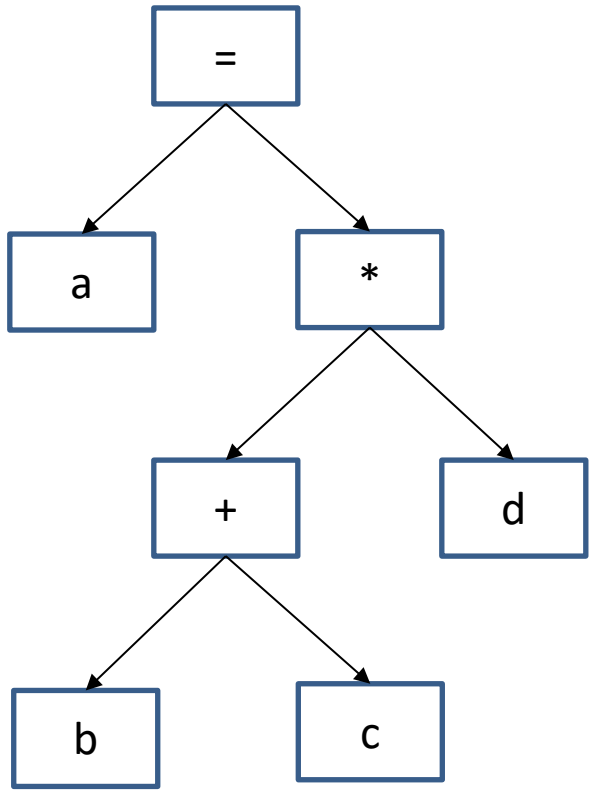
# Lexical Analysis

- Consider the declaration "`int x = 2*a + 1;`"
  - Tokens:
    - `INT`, `IDENTIFIER`, `EQUALS`, `NUMERIC_LITERAL`, `TIMES`, `IDENTIFIER`, `PLUS`, `NUMERIC_LITERAL`, `SEMICOLON`
    - The IDENTIFIER tokens have *attributes* containing the text "x" and "a"
    - The NUMERIC_LITERAL tokens have attributes containing the text "2" and "1"
  - All tokens have attributes specifying their position in the source file
    - … so that good error messages can be produced later
  - Comments and whitespace not in the token stream (typically)

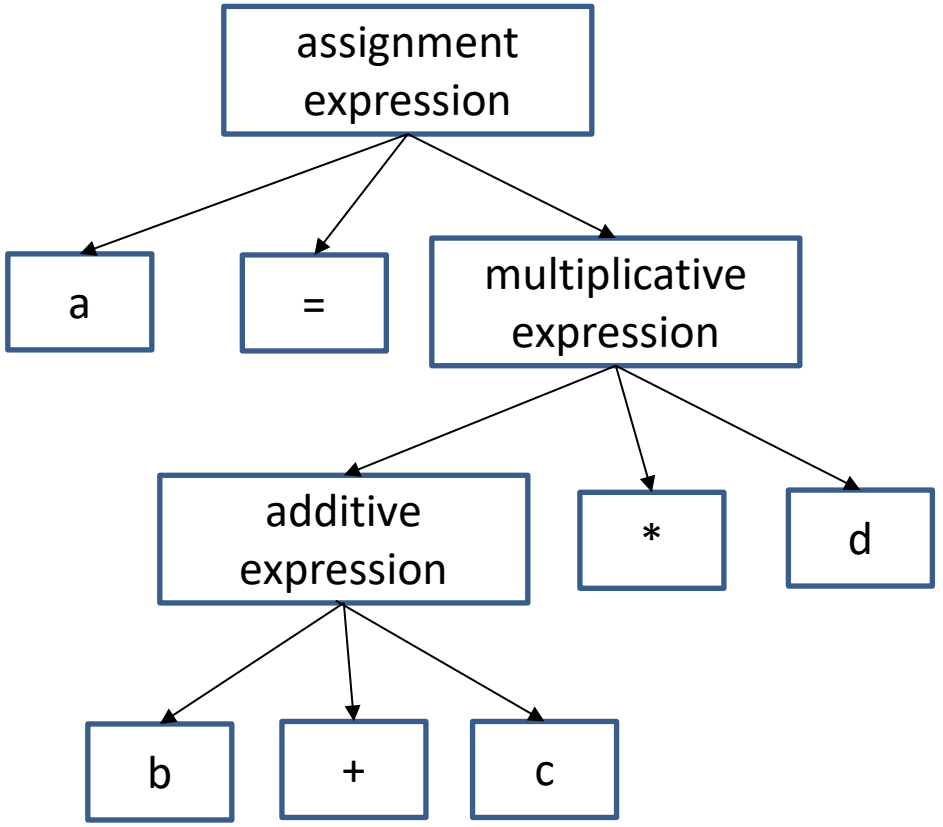
# Parsing

```
a = (b + c) * d;
```

Concrete Syntax




Abstract Syntax




Parse Tree

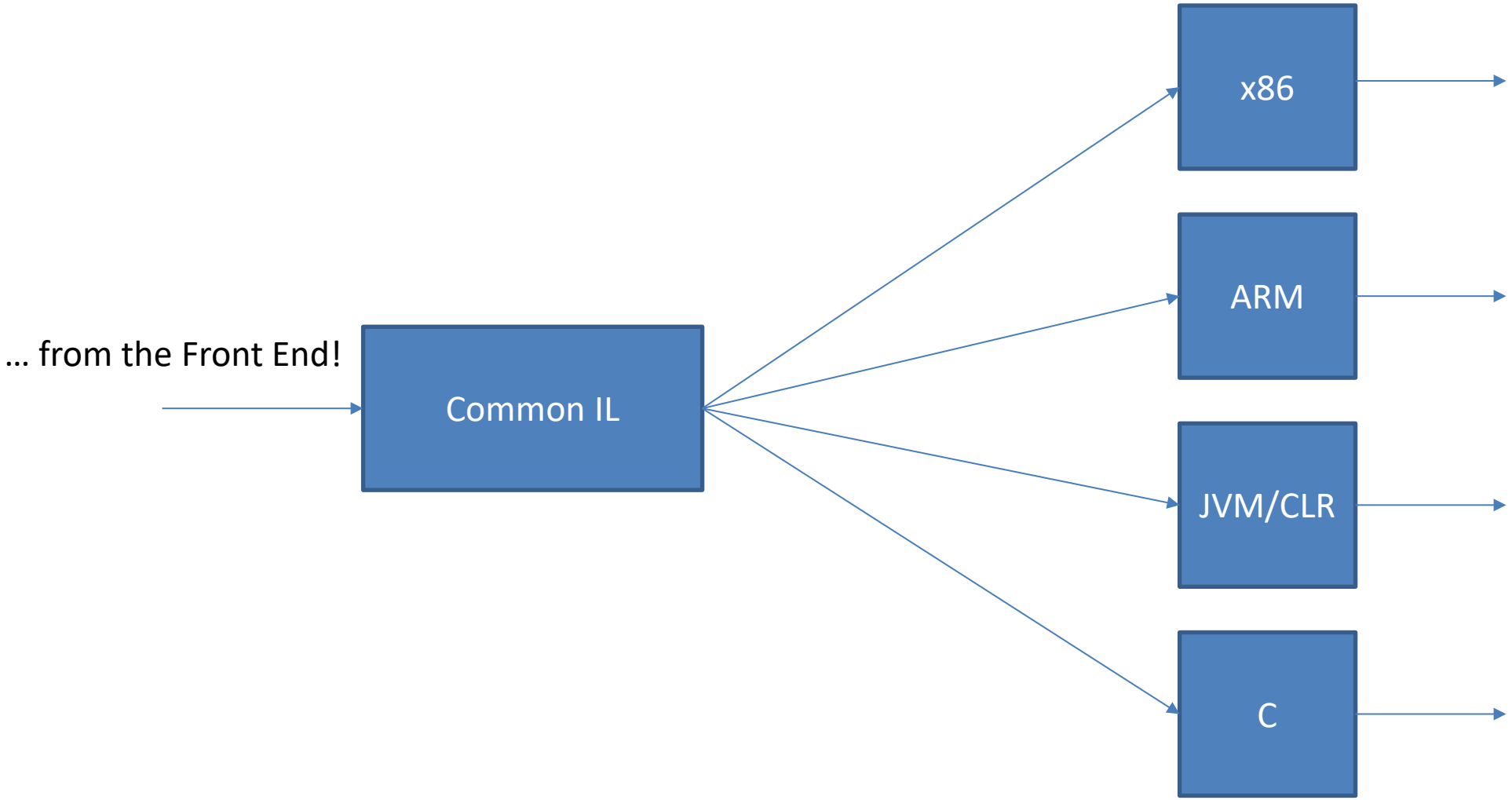
# Semantic Analysis

- Does the program make sense?
  - Type checking… a big part of semantic analysis in statically typed languages
  - All other rules of the language
    - Implicit type conversions
    - Restrictions on operators
    - Rules about how constructs can be ordered and placed, etc.
- Some semantic rules can be enforced during parsing. Some syntactic requirements are left for semantic analysis

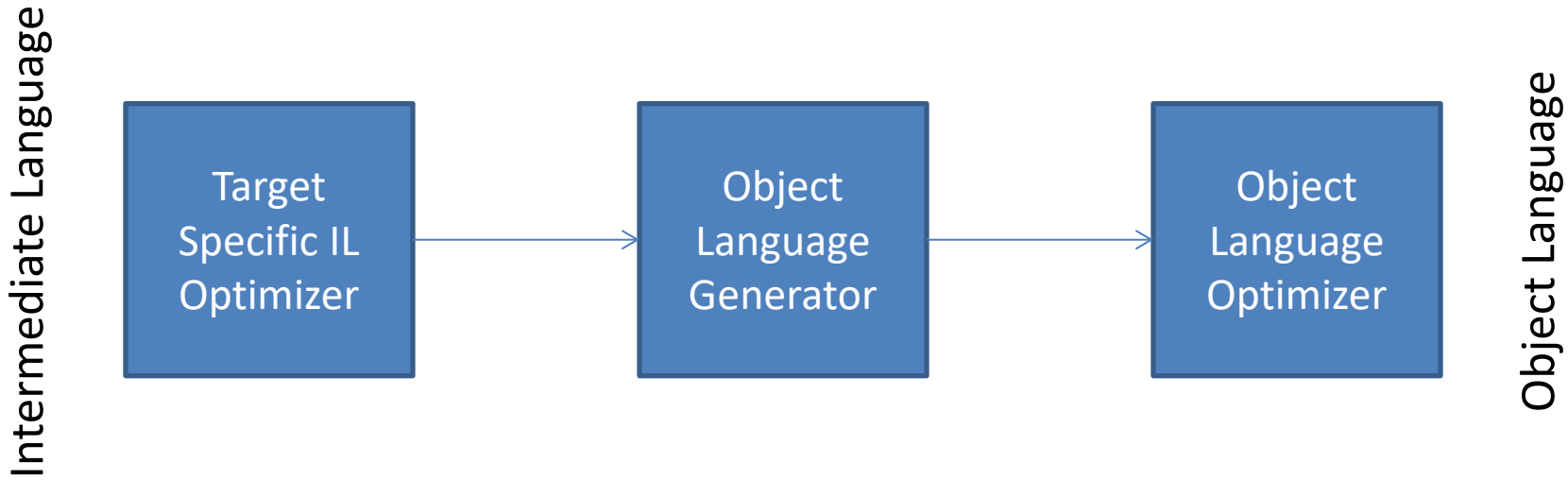
# Back End

- The back end knows the object language
  - Change the back end to target different systems. 16- and 32-bit they all accept the same IL, they can use a common front end.
  - gcc
    - Supports multiple targets by providing a separate back end for each. All source languages work on all targets (in theory).
  - Open Watcom
    - Targets 16- and 32-bit x86 processors, Alpha (not maintained), MIPS (not maintained), x86_64 under development… using multiple back ends.

# Multiple Back Ends



# Back End Pipeline



# Implementation Language

- *Common to write compilers in the language they compile*
  - Compiler writers are experts in their language… want to use it too!
  - Easier to get help from the community; language users can contribute
  - Compiler can compile itself; no dependency on another product
- *… but it is not necessary!*
  - The first compiler for a language obviously can't be in that language!
  - Some languages are more suitable for compiler construction than others
  - Community and cultural reasons may dictate the choice of implementation language (e.g., the Open Watcom Fortran compiler is written in C).

# Course Organization

- The classic compiler pipeline suggests…
  - Starting with scanners and working our way from left to right in the previous diagrams!
  - The book does this also
    - I think all compiler texts do.
- *I'm going to do something slightly different*…
  - Build a simplistic system from start to finish
  - Go back and enhance it to add features exploring other aspects of compiler design.

# Write a Compiler?

- Yes!
  - It is traditional for students to write a small compiler for a simple language in a course like this.
  - The difference is that we'll explore the whole pipeline early and study parts in more depth as we need them (and as time allows).
  - We'll use Scala as our implementation language
  - We'll use LLVM as our back end

# Augusta

- *Augusta* is a simplified subset of the Ada programming language
  - Suitable for education: difficult features of Ada are deleted.
  - Defined for several levels. August L1 has a C-like feature set
- *AGC* (pronounced “Agency”)
  - The Augusta compiler. Currently in an embryonic state.
  - See: https://github.com/pchapin/augusta.

# Scala?

- We will use Scala as our implementation language
  - Scala is a good language for compiler construction (in my opinion).
  - Scala targets the JVM, so our compiler will run anywhere Java runs
  - Scala is a good language to know (resume fodder)
- I will assume no prior experience with Scala

# Back End Options 1

- Interpreter
  - No code generation at all. AGC analyzes the program and also executes it by interpreting the source code.
- C
  - AGC outputs a C program that does the same as the input Augusta program. Use a normal C compiler to generate executable code.
  - This allows users to write Augusta programs for any platform that has a C compiler (i.e., everywhere)

# Back End Options 2

- JVM
  - AGC outputs class files containing Java byte codes that run on the JVM.
  - Since the JVM is fairly high-level, AGC might be able to use Java byte code as its IL and dispense with all the usual back-end stages.
  - Potentially would allow Augusta programs to call Java libraries
  - For ease of implementation, AGC could output JVM assembly language and then use a JVM assembler to generate the class files

# Back End Options 3

- LLVM (Low-Level Virtual Machine)
  - LLVM is at a lower level than the JVM. However, it is still easier to manage than real hardware (the LLVM machine has infinitely many registers, removing the register allocation problem).
  - Very advanced LLVM tools can do state-of-the-art optimizations
  - Potentially interact with code generated by other LLVM targeting compilers such as Clang (C), Clang++ (C++), and others
  - Might be more appropriate for a “systems” language like Augusta.
  - Allows us to explore classic code generation issues

# Back End Options 4

- Real Hardware (x86_64?)
  - This is the most realistic option.
  - To simplify implementation, AGC could output x86_64 assembly language and then use an assembler to generate the object files.
  - Allows us to consider all aspects of code generation, including register allocation
  - CONS: It’s hard work! Also, by definition, it is very target-specific.

# My Background

- What do I know?
  - Been involved in the standardization of C++; talked extensively with real-world compiler writers
  - Been part of the Open Watcom project, working on C/C++ compilers and libraries
  - Created a compiler to convert an enhanced dialect of nesC into pure nesC for my PhD work at UVM.
  - Modified the Scala compiler to add novel type-checking and code-generating facilities as a research project while at UVM

Let’s get started!