Introduction & Overview

Fan Long

University of Toronto

UNIVERSITY OF TORONTO

Course Information

- Instructor: Fan Long
- Contact Info: fanl@cs.toronto.edu
- Office Hours: Monday 15:15-16:15 EST (online) + Special sessions before assignment deadline
- Lectures: Monday 13:00-15:00 EST (online zoom with recording)
- Tutorial: Tuesday 10:00-11:00 EST (online zoom with recording)
- References: Charles Fischer, Ron Cytron and Richard LeBlanc Jr. , Crafting a Compiler, Addison-Wesley 2009

  LLVM Infrastructure websites https://llvm.org

UNIVERSITY OF TORONTO

Course Information

- Marking: MarkUS, link TBD
- Web Page: https://q.utoronto.ca/courses/196799
- Bulletin Board: https://piazza.com/utoronto.ca/winter2021/csc4882107/
- Slides and Handouts:

  Will be posted in Quercus

Important Changes of CSC488

- Restructured course content to focus on **LLVM**
- The new course project is based on **C++** rather than Java
- The project is designed for **individuals** rather than groups

- Lecture + Tutorial format: Zoom session + recorded clips online
- No mid-term exam and only a lightweight take-home final exam

Course Project

- Design and implement a small compiler for MiniC (a toy language)
- The compiler will be based on **LLVM** and therefore be written in **C++**
- Project has **6** phases/assignments
- Code templates will be given for each assignment
- Work **individually** and **independently** to finish the project
- Roughly **1k-2k** lines of code in total for all assignments
- Project contributes to **75%** of the final mark. **Start early!**

Cource Project Requirement

- A PC with Linux environment or a virtual machine that runs Ubuntu 20.04
  - On the first assignment, you will build/install ANTLR4, LLVM 11.0, and Clang 11.0 to setup your project environment.
  - Mac OS may work as well but it is not recommended.
  - Windows is strongly not recommended.

- **C++** skills are very useful. We will have tutorials to help on that.
- Because LLVM infrastructure is C++ based, it is almost impossible to use other programming languages. Our code template is also in C++.

UNIVERSITY OF TORONTO

Project Assignments & Marking

• Assignment 1 (4%)	Prepare environment
• Assignment 2 (10%)	Revise grammar and build parser
• Assignment 3 (11%)	Build AST Tree
• Assignment 4 (12%)	Symbol tables and semantic checking
• Assignment 5 (22%)	LLVM IR generation
• Assignment 6 (16%)	IR optimization
• Final Exam (25%)	

Course Schedule

• Jan 11,	First class
• Jan 25,	Assignment 1 Due
• Feb 1,	Assignment 2 Due
• Feb 11,	Assignment 3 Due
• Feb 15,	Reading Week, no class
• Feb 22,	Assignment 4 Due
• March 15,	Assignment 5 Due
• April 1,	Assignment 6 Due
• April 13-27,	Final Exam

UNIVERSITY OF TORONTO

Course Content

- Introduction
- Parsing Techniques (Lexical and Syntax Analysis)
- AST Trees and Symbol Tables
- Semantic Analysis
- LLVM IR
- IR Code Generation
- Optimizations
- Runtime & Backend Code Generation

Course Project Submission Policies

- Everyone has a grace period of **96 hours** for late for the semester.
- For late beyond the grace period, **2%** penalty is applied per hour
- Sample solutions and test cases will be posted **4 days** after the submission deadline so **no late submission is allowed** after this point.
- If an exception is indeed required, we may approve to shift the mark of the missed submission to other assignments. We will calculate your mark based on your average scores on other assignments.
- However, the **maximum** you can obain in this way is **75%** of the missed assignment. The only exception for this rule is student who add this course and request to shift weights for early assignments.
- You must complete at least **2 out of last four assignments** to receive score in this course.

Course Project Submission Policies

- A student may attempt a second submission within **7 days after the initial deadline** to fix bugs based on the released test cases. Fixed cases will allow the student to retain 75% of marks lost on the cases.
- The second submission must be modifications on the student own code base (not copying sample solutions) and contain descriptions on the root cause of the bugs.
- The assignments are incremental, i.e., future assignments depend on previous ones.
- The student has the freedom to choose continue future assignments based on its own code base or the released sample code.

UNIVERSITY OF TORONTO

Course Project Submission Policies

- Discussion is encouraged, but plagirism is not tolerated.
- You are encouraged to share your thoughts and ideas, but not code.
- Offenders will receive zero on the corresponding assignment.

- **Please refrain from posting your code or sample code online**, even after the submission deadline, we may reuse the course project in future years.

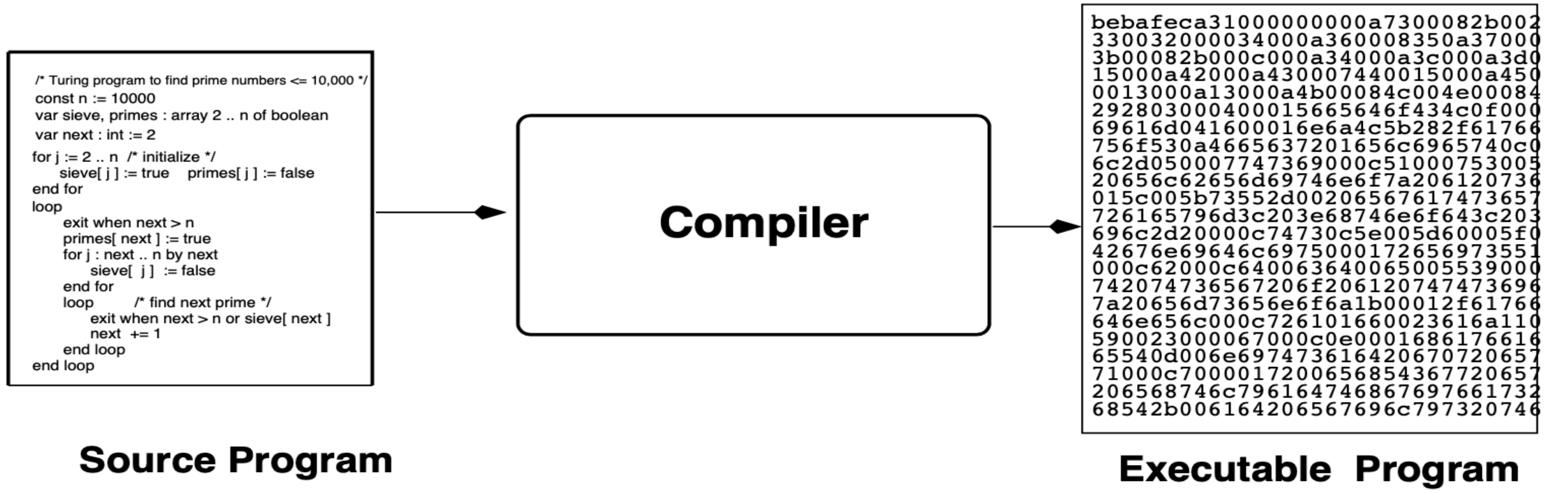
UNIVERSITY OF TORONTO

Compiler Technology is Everywhere

- Compiler techniques are used in many places besides compilers
- Anywhere that complicated structured text needs to be processed
  - Command script interpreters, e.g. bash, Perl, Python
  - HTML processing, e.g., web browsers, servers
  - Interpreters for JavaScript, Flash
  - Query processing: Twitter uses the ANTLR parser for query processing billions of queries per day.
  - Program analysis, e.g. verification, validation
  - Software testing, e.g. test case coverage analysis

What Do Compilers Do?

- Check source program for correctness
  - Well formed lexically – i.e. spell check
  - Well formed syntactically – i.e., grammar check
  - Passes semantic checks – i.e., type correctness and usage correctness
- Transform **source program** into an executable **object program**



```
/* Turing program to find prime numbers <= 10,000 */
const n := 10000
var sieve, primes : array 2 .. n of boolean
var next : int := 2
for j := 2 .. n  /* initialize */
     sieve[ j ] := true    primes[ j ] := false
end for
loop
     exit when next > n
     primes[ next ] := true
     for j : next .. n by next
          sieve[  j ]  := false
     end for
     loop          /* find next prime */
          exit when next > n or sieve[ next ]
          next  += 1
     end loop
end loop
```

```
bebafeca31000000000a7300082b002
330032000034000a360008350a37000
3b00082b000c000a34000a3c000a3d0
15000a42000a43000744001500a450
0013000a13000a4b00084c004e00084
2928030004000156665646f434c0f000
69616d041600016e6a4c5b282f61766
756f530a4665637201656c6965740c0
6c2d050007747369000c51000753005
20656c62656d69746e6f7a206120736
015c005b73552d002065676174736 57
726165796d3c203e68746e6f643c203
696c2d20000c74730c5e005d60005f0
42676e69646c6975000172656973551
000c62000c64006364006500553900 0
742074736567206f206120747473696
7a20656d73656e6f6a1b00012f61766
646e656c000c72610166002361 6a110
5900230000067000c0e0001686176616
65540d006e6974736164206707206 57
71000c70000172006568543677206 57
206568746c79616474686769766173 2
68542b00616420656 7696c797320746
```

Useful Background for Compiler Implementors

- Computer organization (CSC 258H)
- Software engineering (CSC 207H, CSC 301H, CSC 302H, CSC 410H)
- Software Tools (CSC 209H)
- File and data structures (CSC 263H/ CSC 265H)
- Programming languages (CSC 324H)
- Operating systems (CSC 369H)
- Compiler implementation (CSC 488H, ECE 489H)

UNIVERSITY OF TORONTO

Compiler Writing Requires Analytic Skills

- The compiler implementor(s) design the mapping from the source language to the target machine (e.g, x86, ARM, JVM).
- Must be able to analyze a programming language for potential problems. Determine if language can be processed during lexical analysis, syntax analysis, semantic analysis and code generation.
- Must be able to analyze target machine and determine best way to implement each construct in the programming language.

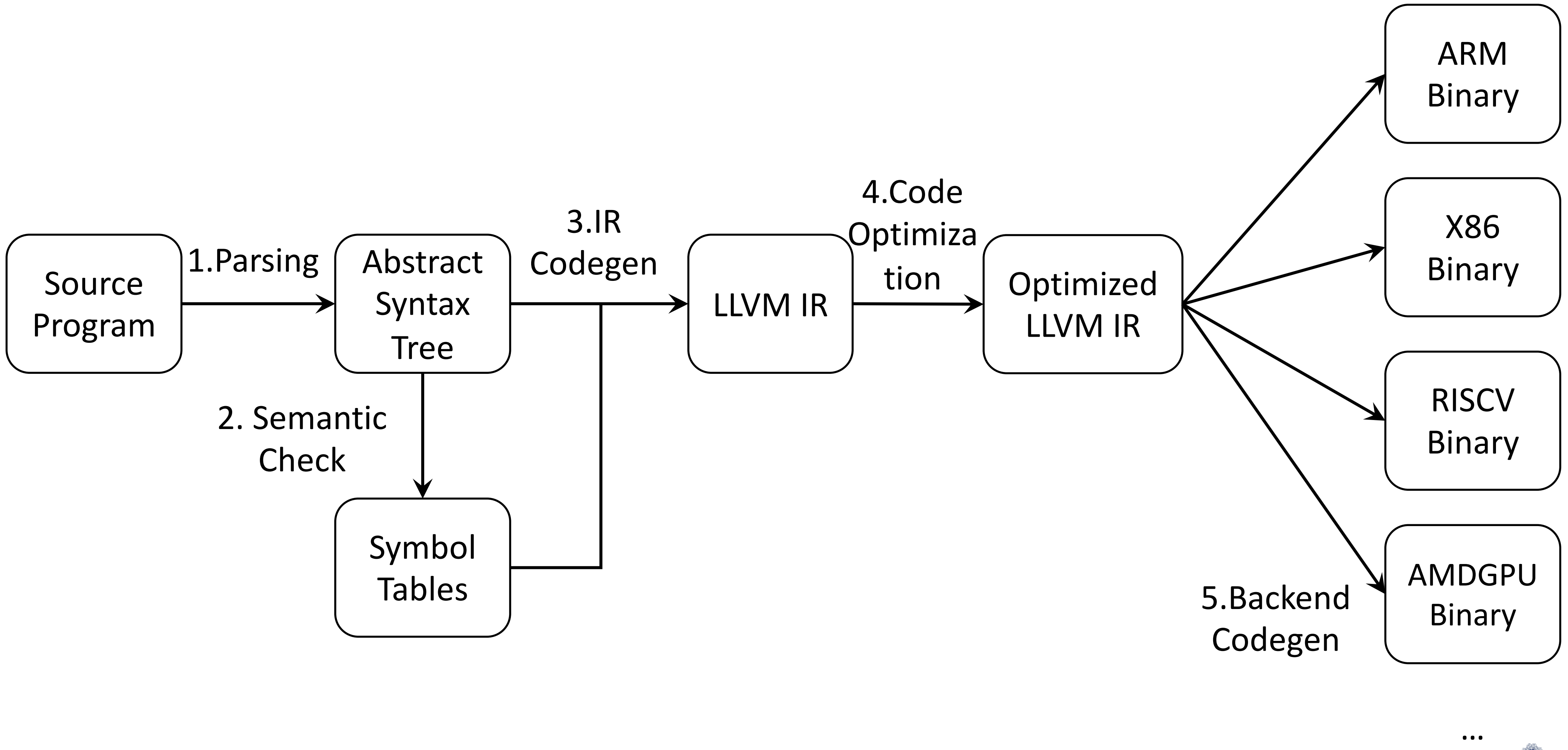
Characteristics of an Ideal Compiler

- User Interface
  - Precise and clear diagnostic messages
  - Easy to use processing options
- Correctly implements the entire language
- Detects all *statically* detectable errors
- Generates highly optimal code
- Compiles quickly using modest system resources
- Good software engineering practice
  - Well modularized, well documented, thoroughly tested, etc.

LLVM Compiler Infrastructure

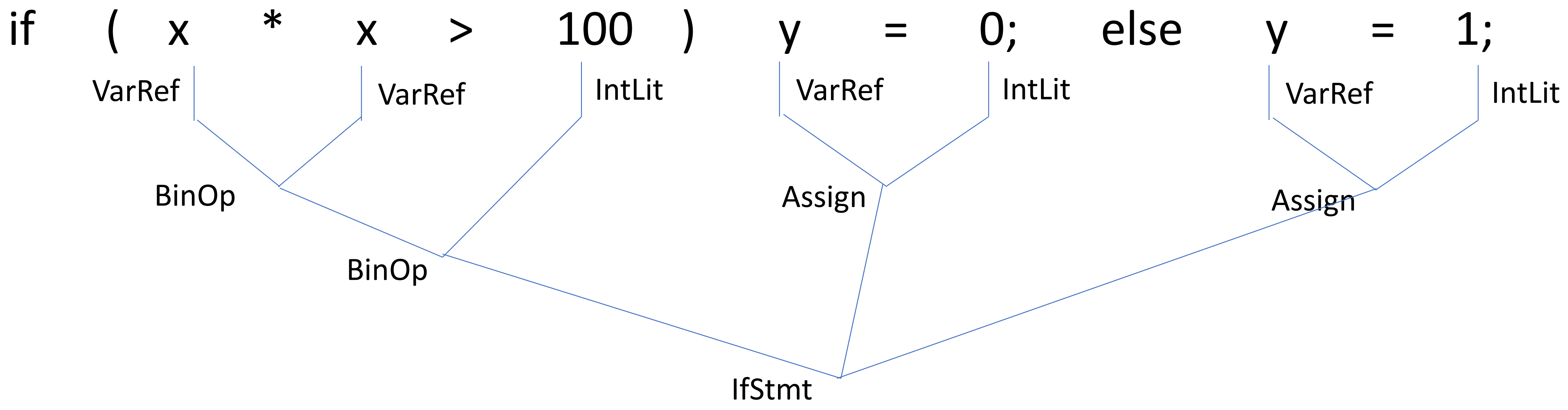
- Collection of industrial strength compiler technology
  - A powerful intermediate representation LLVM IR
  - Optimizer and code generator
  - Multiple backends for different architecture targets
- Open source project with many contributors
  - Industry, research groups, individuals
  - De-facto standard of building modern compilers
- Clang: C/C++ compiler built on top of LLVM

Typical LLVM Compiler Workflow



Example: Source code & AST

- Source Code



- AST

Example: LLVM IR

```
  %4 = load i32, i32* %x, align 4
  %5 = load i32, i32* %x, align 4
  %6 = mul nsw i32 %4, %5
  %7 = icmp  sgt i32 %6, 100
  br i1 %7, label %then, label %else

then:
  store i32 0, i32* %y, align 4
  br label %out

else:
  store i32 1, i32* %y, align 4
  br label %out

out:
```

- Virtual registers and labels start with “%”
- The number of virtual registers are unlimited
- Each register is assigned only once
- Registers can be either named or unnamed (only numbered).
- Low level instructions like **load**, **store**, **icmp**, **mul**, **br**, etc.

Example: LLVM IR Optimization

```
  %4 = load i32, i32* %x, align 4
  %5 = load i32, i32* %x, align 4
  %6 = mul nsw i32 %4, %5
  %7 = icmp  sgt i32 %6, 100
  br i1 %7, label %then, label %else

then:
  store i32 0, i32* %y, align 4
  br label %out

else:
  store i32 1, i32* %y, align 4
  br label %out

out:
```

$\rightarrow$

```
  %4 = load i32, i32* %x, align 4

  %6 = mul nsw i32 %4, %4
```

- %4 and %5 always have equal values.
- We can eliminate **load** and **%5**

Example: X86 Code Generation

```
  %4 = load i32, i32* %x, align 4
  %5 = load i32, i32* %x, align 4
  %6 = mul nsw i32 %4, %5
  %7 = icmp  sgt i32 %6, 100
  br i1 %7, label %then, label %else

then:
  store i32 0, i32* %y, align 4
  br label %out

else:
  store i32 1, i32* %y, align 4
  br label %out

out:
```

```
  movl    -8(%rbp), %eax
  imull   -8(%rbp), %eax
  cmpl    $100, %eax
  jle     LBB0_2
movl      $0, -12(%rbp)
  jmp     LBB0_3
LBB0_2:
  movl    $1, -12(%rbp)
LBB0_3:
```

Advantages of LLVM IR and Infrastructure

- Can leverage existing optimization passes on LLVM IR
- Can quickly build compilers that generate **fast** code
- Can leverage existing backend implementations
- Can quickly build compilers that support **multiple architectures**

- After finishing the course project, you will learn how to use LLVM to implement a fast compiler for a new programming language.

UNIVERSITY OF TORONTO

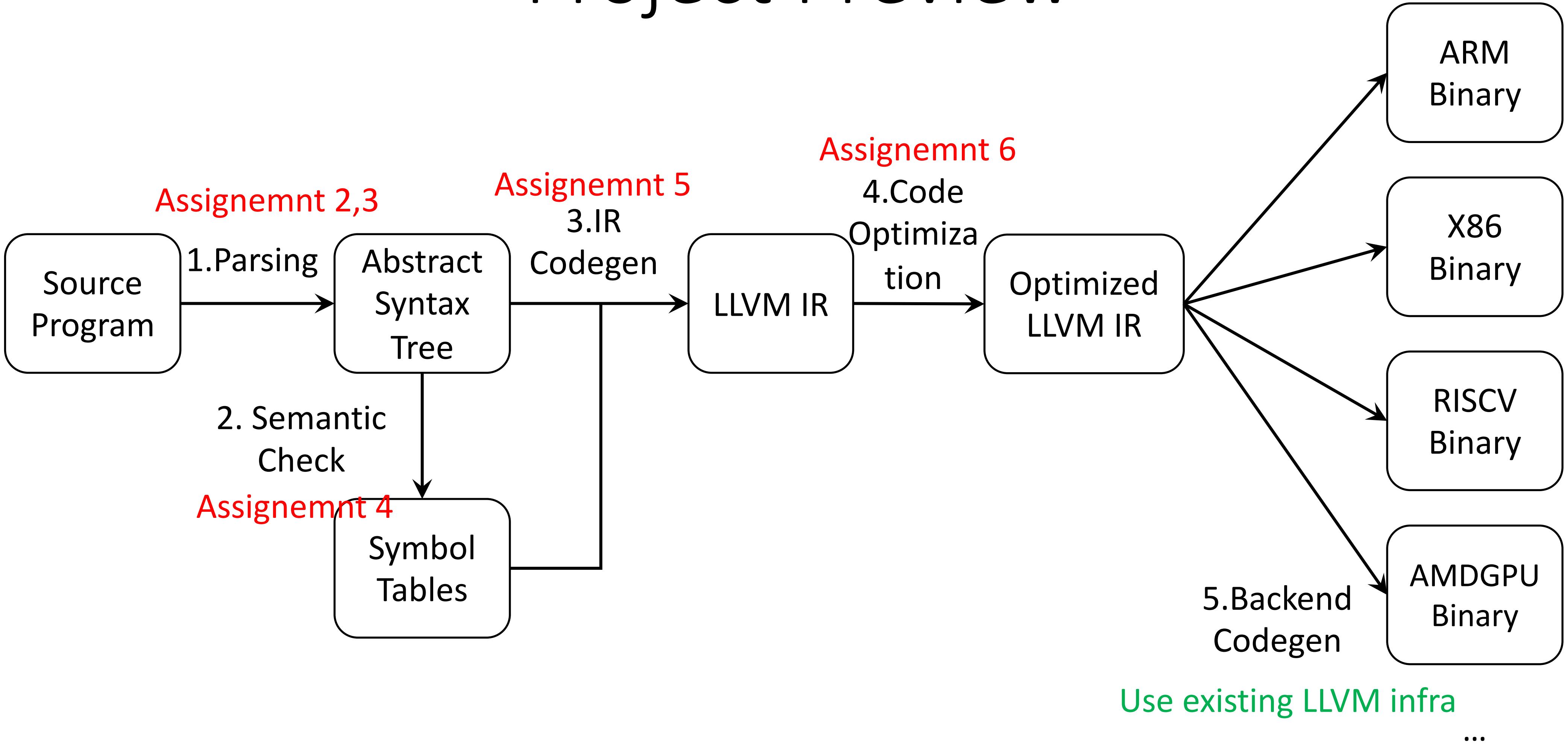
Interpretive Systems

- Compiler generates a pseudo machine code to encode the program.
- The pseudo machine code is executed by another program (an *interpreter*)
- Interpreters are used for
  - As a way to port programs between environments.
  - Implementing ugly language features.
  - Languages that allow dynamic program modification.
  - Typeless languages that can’t be semantically analyzed statically.
- Interpreters lose on
  - Execution speed, usually significantly slower than machine code.
  - May require recompilation for each run.

Example of Interpreters

- Java Virtual Machine
  - Java programs are compiled to a *byte-code* for *Java Virtual Machine* (JVM).
  - JVM designed to make Java portable to many platforms.
  - JVM slow execution speed has lead to the development of *Just In Time* (JIT) native code compilers for Java.
- Python
  - Python official implementation is an interpreter for the Python source code (no compilation).
- LLVM IR
  - LLVM IR can be directly executed with its JIT interpreter lli

Project Preview



Q/A?