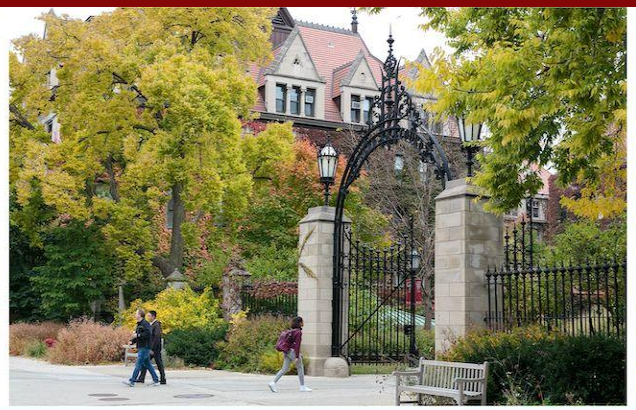
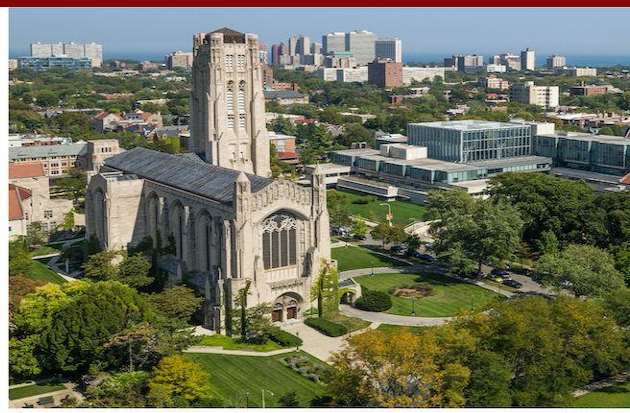
MPCS 51300 - Compilers
M1: Introduction to Compilers

Remote Students please mute your microphones, thank you.

Lamont Samuels

Agenda

- Course Logistics
- Introduction to compilers
  - What is a compiler?
  - Anatomy of a compiler
  - Why should we learn about them?
- Course Language [if time permits]

Course Logistics

- Remote Students
  - Please make sure you are muted when entering the discussion.
  - You may unmute if you would like to ask question because I may not be able to see you raise your hand or ask a question in the chat window.
  - If you are encountering issues related to Zoom, please try shutting down your machine and signing back in. These meetings are recorded so if you miss them then you'll be able to watch it at a later time.

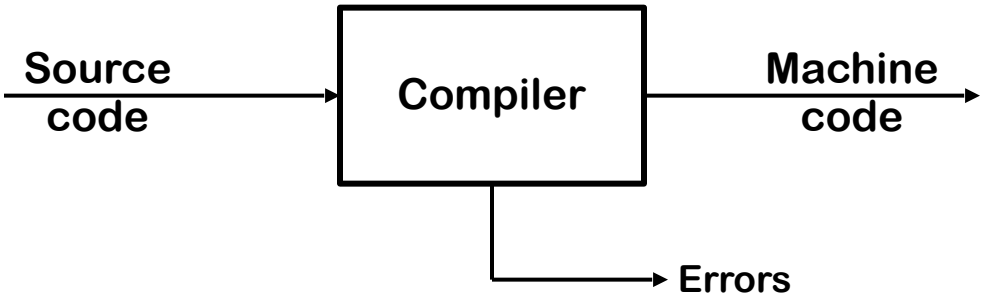
- All course information is located on the course website

  https://classes.cs.uchicago.edu/archive/2021/fall/51300-1/index.html

- First-time teaching this course so there will be some leniency in many aspects of the course to benefit you all.

What is a Compiler?

- Compilers are simply *translators*
  - Translates between representations of program code
  - Typically from a high-level source language to machine language (object code)



  - Not all compilers translate to machine code
    - Java compiler: Translate java code to interpretable JVM bytecode
    - Java JIT: bytecode to machine code.

Compiler vs. Interpreter

Are compilers necessary to execution all program code?

- No! Programs can be simulated using an *interpreter*:
  - A program that reads an executable program and produces the results of executing that program.
- Most interpreters are at least 10x slower than compiled code depending on the type of program.

clang sum.c -o sum && time ./sum 10000000

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int n, i, sum;
    n = atoi(argv[1]);
    for (sum = i = 0; i < n; ++i) {
        sum += 1;
    }
    printf("%d",sum);
    return 0;
}
```

time ./sum.py 10000000

```python
#!/usr/bin/env python3
import sys
n = int(sys.argv[1])
sum = 0
for i in range(n):
    sum += 1
print(sum)
```

C: 0.05 seconds
Python: 0.53 seconds
C runs more than 10x faster

Compiler Input: Source Code

- Optimized for human readability
  - Uses human notions of grammar to be more expressible
  - Redundant to help avoid programming errors
  - The final result may not be fully determined by the code.

```
int expr(int num)
{
    int value;
    value = (num + 1) * (num + 1) / 2;
    return value;
}
```

Compiler Output: Assembly/Machine Code

- Optimized for hardware
  - Ambiguity reduced
  - Reasoning about what the code is doing is lost

```
_expr:                                  ; @expr
; %bb.0:
    sub  sp, sp, #16                    ; =16
    .cfi_def_cfa_offset 16
    str  w0, [sp, #12]
    ldr  w8, [sp, #12]
    add  w8, w8, #1                     ; =1
    ldr  w9, [sp, #12]
    add  w9, w9, #1                     ; =1
    mul  w8, w8, w9
    mov  w9, #2
    sdivw8, w8, w9
    str  w8, [sp, #8]
    ldr  w0, [sp, #8]
    add  sp, sp, #16                    ; =16
    ret
```

Compiler Output: Optimized vs Unoptimized

Unoptimized Code

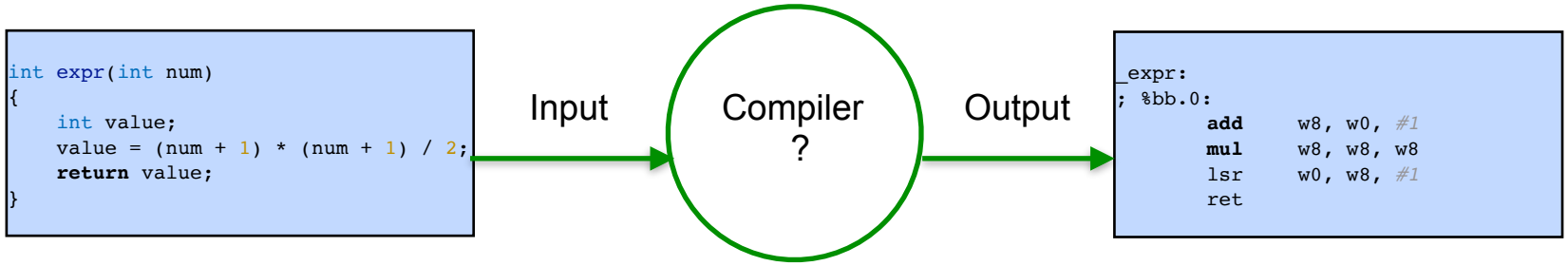
```
_expr:
; %bb.0:
    sub  sp, sp, #16
    .cfi_def_cfa_offset 16
    str  w0, [sp, #12]
    ldr  w8, [sp, #12]
    add  w8, w8, #1
    ldr  w9, [sp, #12]
    add  w9, w9, #1
    mul  w8, w8, w9
    mov  w9, #2
    sdiv w8, w8, w9
    str  w8, [sp, #8]
    ldr  w0, [sp, #8]
    add  sp, sp, #16
    ret
```

Optimized Code

```
_expr:
; %bb.0:
    add  w8, w0, #1
    mul  w8, w8, w8
    lsr  w0, w8, #1
    ret
```

Compiler Translations

- How does a compiler effectively translate high-level source code to low-level machine code?
  - Remember at the lowest level, a computer only knows about the binary encodings of 1s and 0s, which represents hardware instructions and data.



- A compiler translates the original source code into different program representations known as intermediate representations:
  - These representations are designed to support the necessary program manipulations:
    - Type checking
    - Static analysis
    - Optimization
    - Code generation

a = b x c + d

e = f + b x c + d

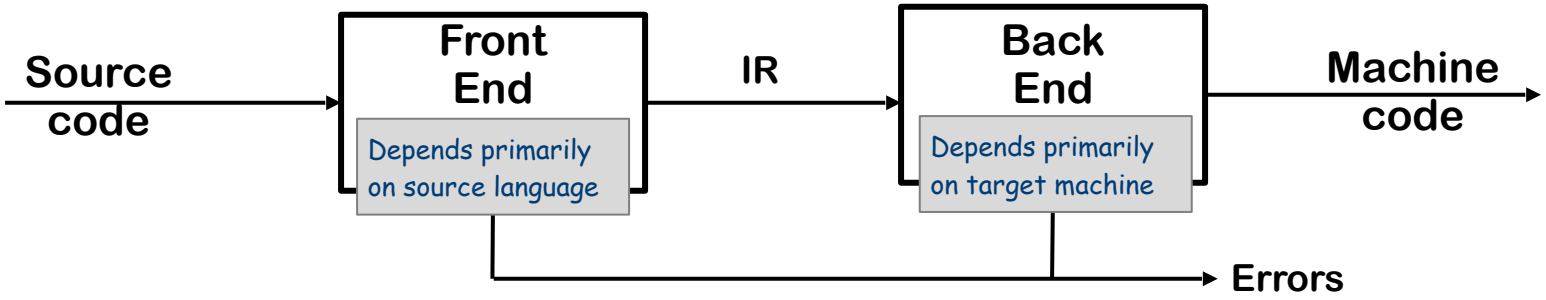
load @b ⇒ r_1
load @c ⇒ r_2
mult r_1,r_2 ⇒ r_3
load @d ⇒ r_4
add r_3,r_4 ⇒ r_5
store r_5 ⇒ @a
load @f ⇒ r_6
add r_5,r_6 ⇒ r_7
store r_7 ⇒ @e

computes b x c + d

reuses b x c + d

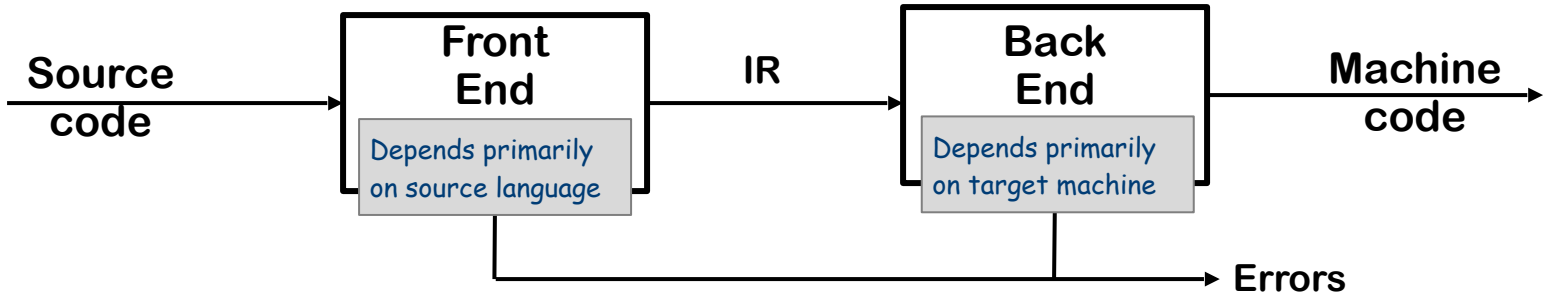
Anatomy of a Compiler

- At a high level, a compiler contains two main parts:
  - Front end: analysis
    - Analyze the source code and determine its structure and meaning to generate an intermediate representation.
  - Back end: synthesis
    - Generate low-level code for the target platform



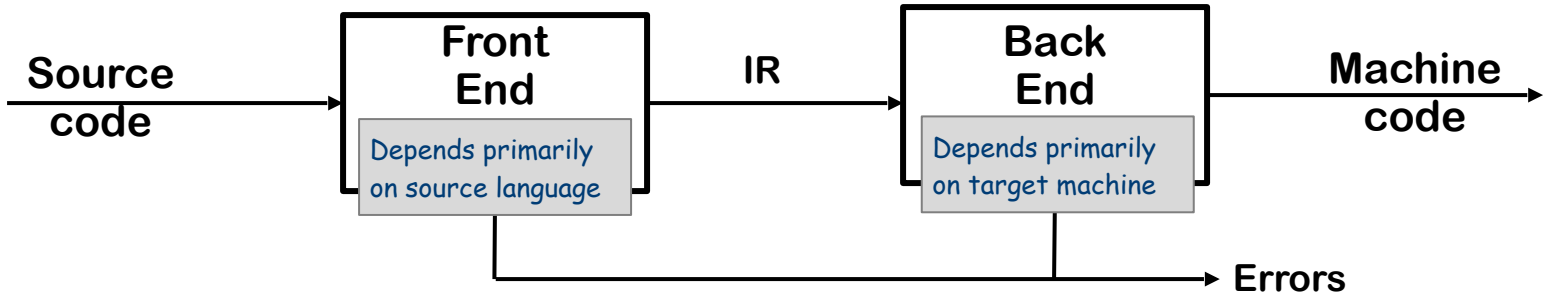
Anatomy of a Compiler: Implications

- A compiler must do the following during its translation process:
  - Must recognize legal (and complain illegal) programs
  - Must generate correct code
    - It can attempt to improve ("optimize") code, but must not change a code's behavior ("meaning")
  - Must manage storage of all variables (and code)
  - Must agree with OS & linker on format for object code



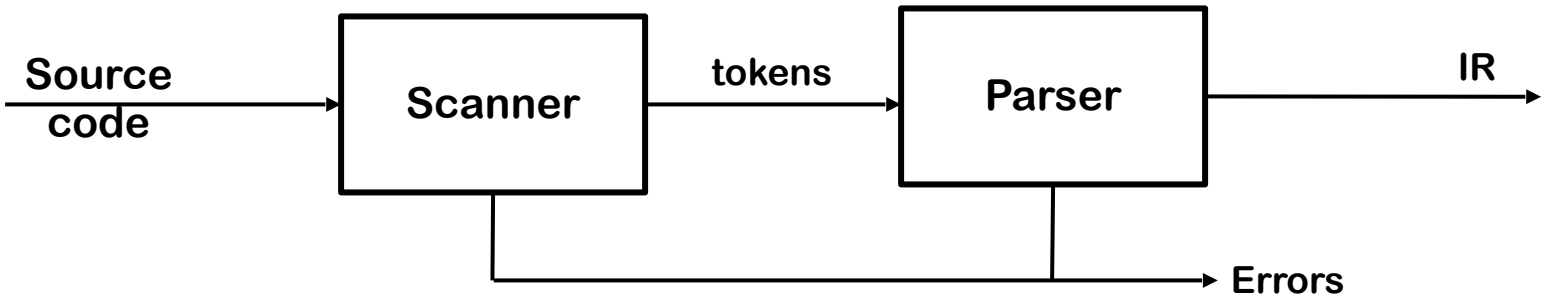
Anatomy of a Compiler: Implications

- Each phase of the compiler uses intermediate representation(s) (IR) to pass along results from its phase to another
  - Front end maps source into an IR
  - Back end maps IR to target machine code
  - Compilers do often has multiple IRs - higher at first, lower level in later phases.



Anatomy of a Compiler: Front End

- Contains two main components:
  - Scanner - Maps character stream into a token stream: keywords, operators, variables, constants, etc.
    - Also removes all white space, and comments
  - Parser: Reads in the tokens from the token stream and generates an IR
    - Also performs semantics analysis to check for type errors, etc.
- Both these components can be automatically generated
  - Define a formal grammar to specify the source language
  - An existing software tool will read the grammar and generate a scanner & parser
    (e.g., ANTLR for C/C++, Java and Go, or flex/bison for C/C++)

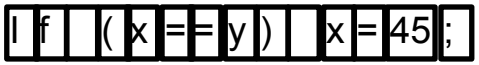


Front End: Scanner Example

- Input source code:

```
if (x==y) x=45;
```

- Character Stream:



- Token Stream:

IF	LPAREN	ID(x)	EQ	ID(y)	RPAREN	ID(x)	ASSIGN	INT(45)	SCOLON

  - Tokens are distinct objects that can carry associated data with them (e.g., numeric value, variable name, etc.)
  - Whitespace and comments are not tokens.

Front End: Parser

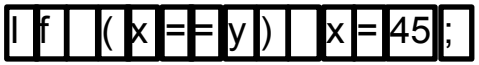
- Responsible for taking the token stream and producing an IR output that captures the meaning of the program.
- Most common output is an Abstract Syntax Tree (AST)
  - Contains meaning of program without syntactic noise
  - Internal nodes are operations, and leafs are operands
  - Known as the “natural” IR
- AST is not the only possible output
  - Parse Tree/ Syntax Tree is possible but usually contains additional information that is not needed.

Front End: Scanner/Parser Example

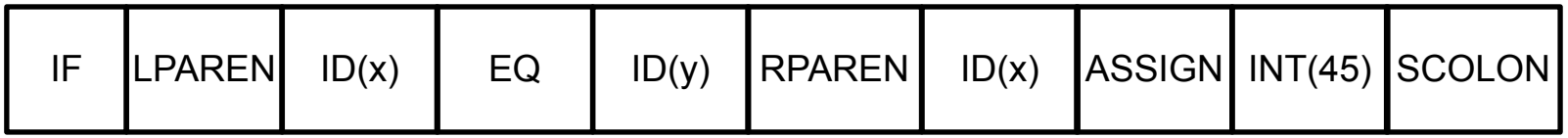
- Input source code:

```
if (x==y) x=45;
```

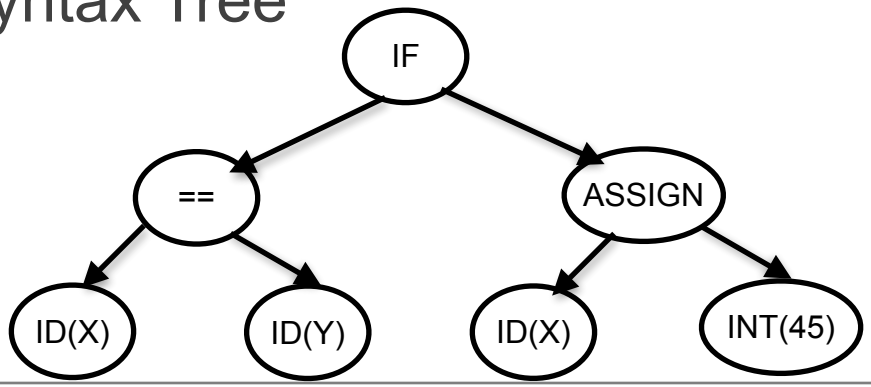
- Character Stream:



- Token Stream:

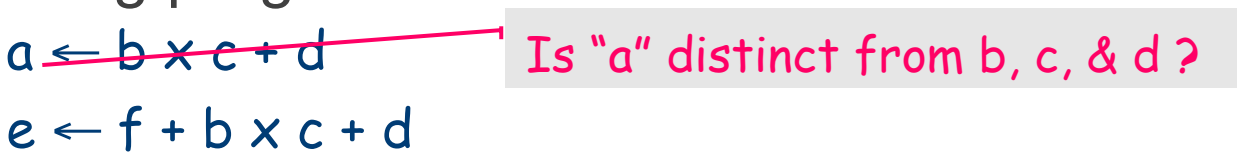


- Abstract Syntax Tree



Static Semantic Analysis

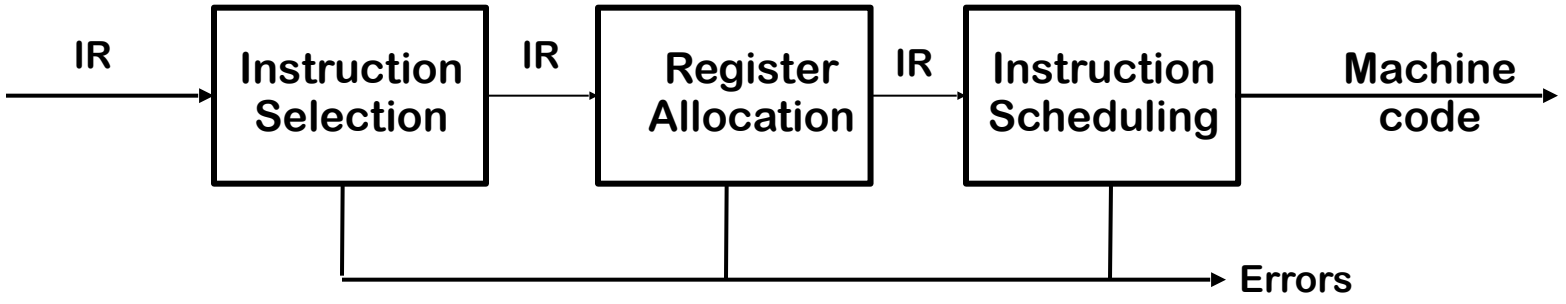
- A step in the compiler that happens during parsing or directly after to ensure a program is valid
  - Performs type checking
  - Verifies code adheres to language semantics (e.g., correct variable declarations)
  - Performs code shape determines many properties of resulting program



- Collects additional information for the back end like the Symbol Table(s)
  - One symbol table maps names to types.

Anatomy of a Compiler: Back End

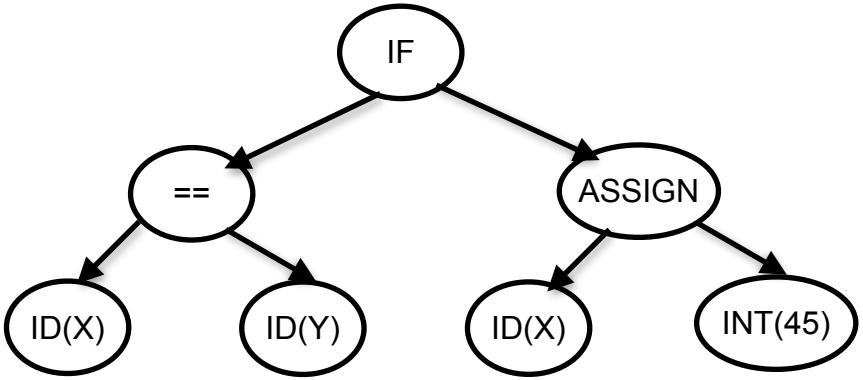
- Responsibilities
  - Translate IR (AST) into target machine code
  - Choose instructions to implement each IR operation
  - Decide which value to keep in registers
  - Ensure conformance with system interfaces
- Tries to produce the most “optimal” code
  - optimal = fast, compact, low power (can’t have them all)
- Automation has been less successful in the back end



Anatomy of a Compiler: Result

- Input

```
if (x==y) x=45;
```

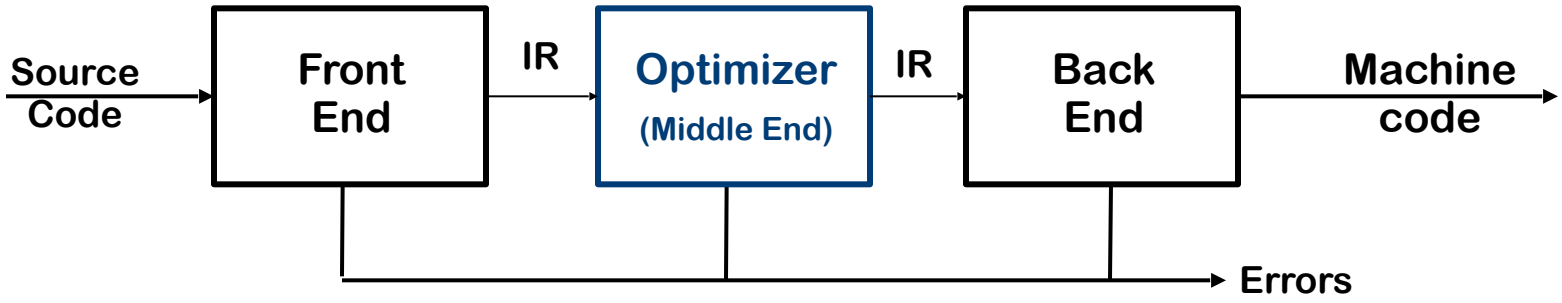


- Output

```
    sub   sp, sp, #16
    str   w0, [sp, #12]
    str   w1, [sp, #8]
    ldr   w8, [sp, #8]
    ldr   w9, [sp, #12]
    subs  w8, w8, w9
    b.ne  LBB0_2
; %bb.1:
    mov   w8, #45
    str   w8, [sp, #8]
LBB0_2:
    ldr   w0, [sp, #8]
    add   sp, sp, #16
```

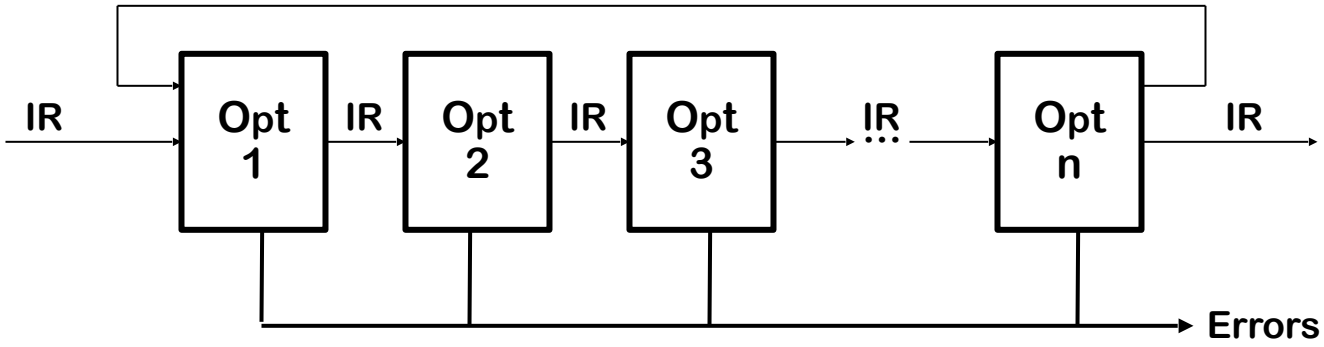
Wait! A Three-Part Compiler?

- Code Improvement (or Optimization)
  - Analyzes IR and rewrites (or transforms) IR
  - Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
  - Must preserve "meaning" of the code
  - Measured by values of named variables



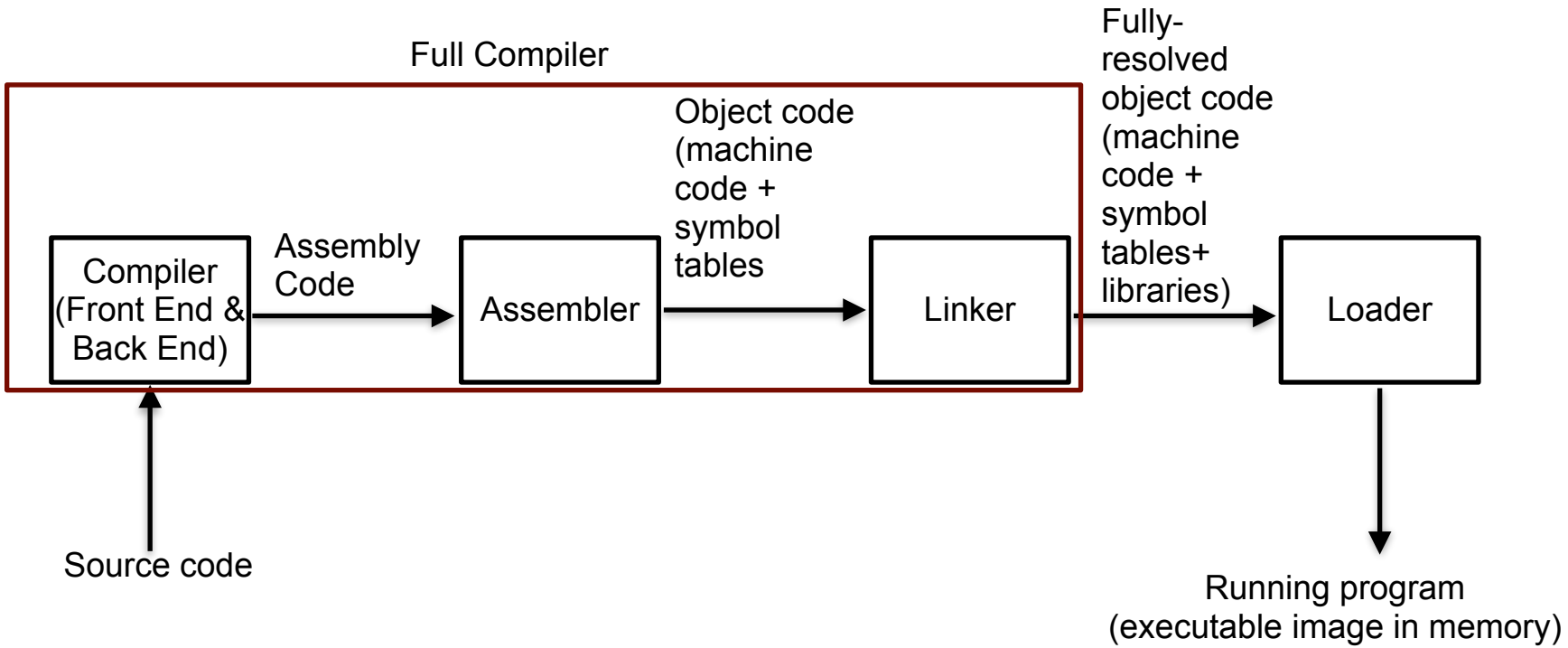
Anatomy of a Compiler: Optimizer

- Typical Transformations
  - Discover & propagate some constant value
  - Discover a redundant computation & remove it
  - Remove useless or unreachable code
- Tradeoffs in optimization
  - Ordering of optimization phases
  - What works for some programs can be bad for others



Modern optimizers are structured as a series of passes

Anatomy of a Compiler: Creating an Executable



- Note: many compilers include the assembler and linker as part of the compiler

Why Study Compilers?

- Compiler construction involves ideas from many different parts of computer science

Artificial intelligence	Greedy algorithms Heuristic search techniques
Algorithms	Graph algorithms, union-find Dynamic programming
Theory	DFAs & PDAs, pattern matching Fixed-point algorithms
Systems	Allocation & naming, Synchronization, locality
Architecture	Pipeline & hierarchy management Instruction set use

Why Study Compilers?

- Compilers are important
  - Responsible for many aspects of system performance
  - Attaining performance has become more difficult over time
    - Compiler has become a prime determiner of performance
- Compilers make you into a better programmer
  - Provides insight into interaction between languages, compilers, and hardware
  - Allows you to understand how code maps to hardware
  - Provides better intuition about what your code does
  - Understanding how compilers optimize code helps you write code that is easier to optimize
    - Helps with not writing pointless code that “optimizes” the performance when a compiler can do it better.

Why Go for this Course?

- Go was designed by Google (specifically Rob Pike, Kenneth Thompson in 2007) to solve problems that Google faces.
- Goals
  - Eliminate slowness
  - Inefficiencies
  - Maintain and improve scale

- Why is course?
  - Relatively small language that has enough features to implement a entire compiler efficiently
  - Good module system that will allow use to easily test/debug the various components of the compiler
  - Everyone is on the same level and will be easier for class interactions with each other and to get help when needed

How are we going to learn Go?

- We are actually going to learn Go by going through examples shown on: https://gobyexample.com

- Topics to look at
  - Values
  - Variables
  - Constants
  - Importing
  - For
  - If/Else
  - Switch
  - Maps
  - Range
  - Functions
  - Multiple Return Values
  - Variadic Functions
  - Pointers
  - Structs
  - String Formatting and Functions