West Chester University
Dr. Richard Burns
Spring 2023

CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

Today

Course Introduction

Overview of a Compiler

Programming Assignment 0

Questions

1. What is a **compilers** course?
2. What do you expect to learn?
3. What do you expect not to learn?
4. How *difficult* is this course?

Some Answers

- Traditionally one of the toughest CS courses in the curriculum.
- Sometimes a required course for the CS degree (other colleges).
- At WCU (previously): CSC 416/496/417 satisfy the Large-Scale Complex System requirement for *undergraduates*.
- **Big Idea:** implementing a very large, substantial software project.

Other Benefits of a Compilers Course

- Touches every CS course you've taken-to-date:
  - (except networks, security?)
- Makes use of lots of things you previously learned
  - Arrays, lists, queues, stacks, trees, graphs, maps, regular expressions
  - Finite state machines, context-free grammars, recursion, software patterns

More Benefits of a Compilers Course

- Coding a large-scale software project.
- Probably larger than anything you've been assigned previously at WCU
- More like real-life
- Organization of compiler is in *stages*
    - compiler writing is a case study in software engineering

Even More Benefits

- **Compilers** and **Interpreters** are everywhere.
- You'll be able to write *better* code.
- You'll be able to write *faster* code.
- Better understanding of Java.
- Maybe you'll write your own programming language in this course...
  - ...with array indices that begin at 1 rather than 0.
  - ...or simply will learn to take advantage of parsing tools.

This Course

- Assignments: Written Homeworks + Programming Assignments ("build a compiler").
- will program our Compiler in Java and utilize some tools that generate Java code.
- will make our own language (subset of Java).
- will follow the textbook along the way.
  - (there's a pretty standard sequence of steps compilers take to actually compiler a program).

Course Homepage

https://www.cs.wcupa.edu/~rburns/Compilers

Resources

COURSE HOMEPAGE.

COURSE SCHEDULE.

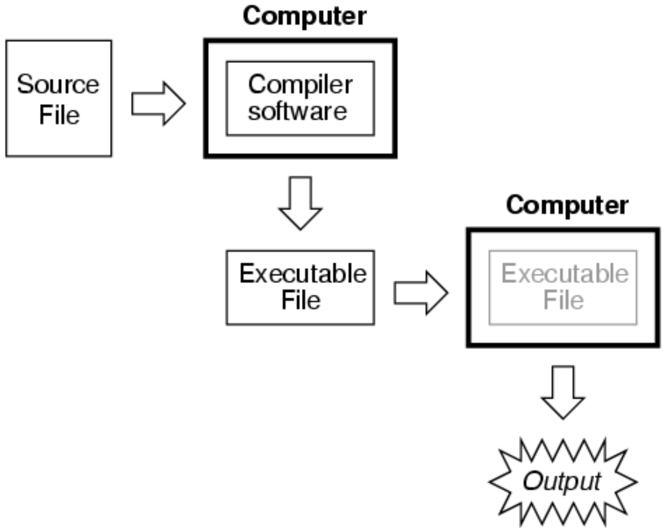
D2L FOR GRADES AND ASSIGNMENT SUBMISSION ONLY.

DISCORD SERVER.

TEXTBOOK

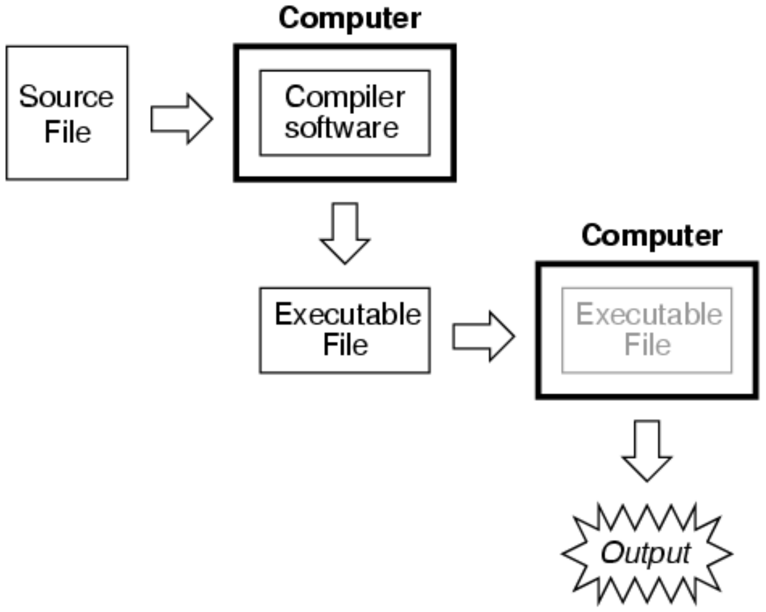
What is a Compiler?

- What are we going to build over the semester?
- Let's first contrast Interpreters and Compilers.



- runs a program by examining its high-level constructs and simulating their actions, executing it directly
  - Sometimes known as a **virtual machine**
- interpreter takes your program as input
- does not perform any preprocessing on the input (typically)
- runs the program on any input data and produces output
- "on-line" in the sense that its work is part of running the program

Interpreter



Compiler

- translates the high-level constructs (**source language**) into low-level machine instructions (**target language**) that can be executed directly by a computer
- compiler is "offline" in contrast to an interpreter - it preprocesses the data and translates it into some code form that can be executed
- when we execute the compiled code, that is when we input any data
- can run the executable on many different inputs without compiling again

Why not interpret all programs?

Performance is a major reason: native machine code programs run faster than interpreted high-level language programs.

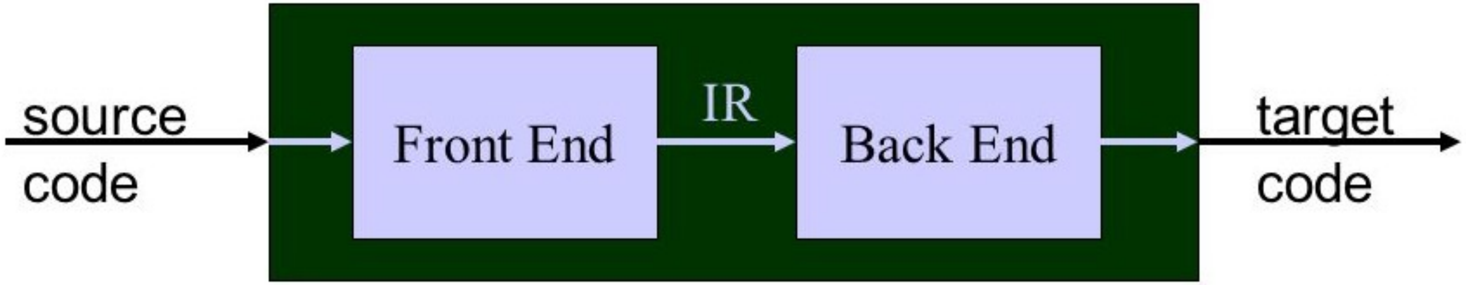
Interpreters must parse and analyze a statement to decode its meaning every time it executes that statement.

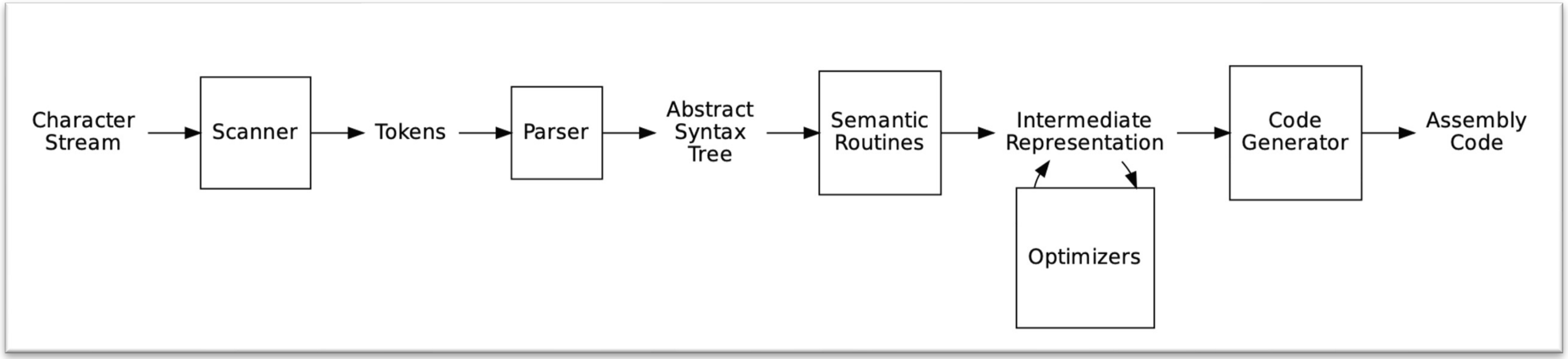
Some languages (C, C++, Java) have both interpreters (for debugging and program development) and compilers (for production work).

Some History

- Compilers have a storied past, dating back to the 1950s.
  - term coined by Grace Hopper
  - also called "automatic programming", such a big SE task there was skepticism it would ever be successful
- Hardware was very expensive,
  - ... but software was even more expensive, because of the inefficiency of the programming process.
- 1953: "Speed Coding" interpreter,
  - increased productivity of coding
  - but executed programs were ~10x slower
  - same is true today (this is why Python tries to "compile" intelligently)
- Late 1950s: FORTRAN - *FORmulas TRANslated*, (rather than interpreted)
  - the first successful compiler & machine independent language
  - construction required understanding of CS theory and SE skills
  - most modern compilers today more or less following the FORTRAN design

Structure of a Compiler

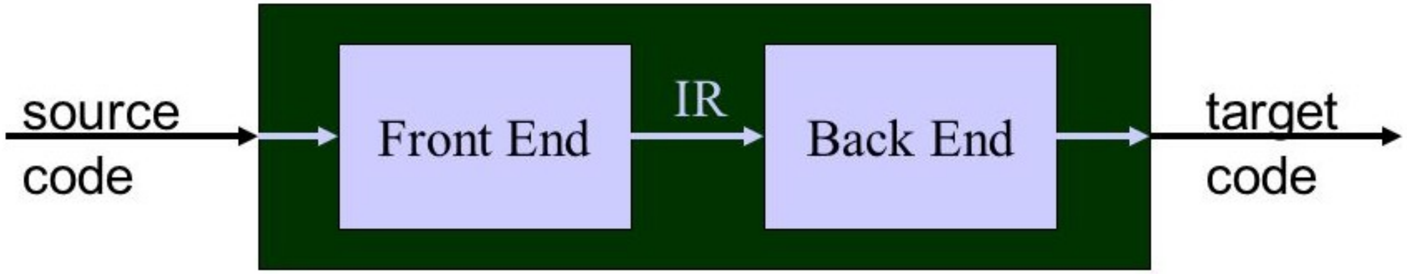




- This is the high-level structure of a compiler that we will follow:
- Easier for back-end to perform its functions on the IR, rather than the source program.

Front End

- "Translating into the IR"
- Steps of the *front end*:
  1. Lexer / Scanner
  2. Parser
  3. Semantic Analysis



Lexer / Scanner

Consumes the plain text of the source program.

Break source file into individual words or **tokens**.

Example: `This is CSC416/565.`

There are 3 tokens in this example. How can we find them?

Example: `jdcbqk   lnefg hig`

What are the **tokens** in this sentence?

<u>Code</u> Example:

```
if x == y then z = 1; else z = 2;
```

How many **tokens**?

Parser

- Consumes the tokens.
- Analyzing the phrase structure of a program.
- The scanner and parser check the syntax of the input program.
- Parser will utilize a grammar specification
  - (see CSC220? or CSC520?)

Example: Grammatical Rules of English

Rule: `Sentence -> Subject verb Object endmark`

Input: `This line is a long sentence .`

Is the input a **valid sentence** given our rule?

Of course English is complicated, and many more rules would be necessary for a full English specification.

Another Example

Example: `if x == y then z = 1; else z = 2;`

What “grammatical” rules could we write for code?

Now finished syntactic analysis performed by the front end.

1. Lexing
2. Parsing

Up next to finish front end work: semantic analysis

Semantic Analysis

Idea: check for inconsistencies in the source code

- Are variables declared?
  - only once within each scope, etc...
- Are there mismatches?
  - type-consistent use of names
- Can you think of anything else?

<u>Goals:</u>

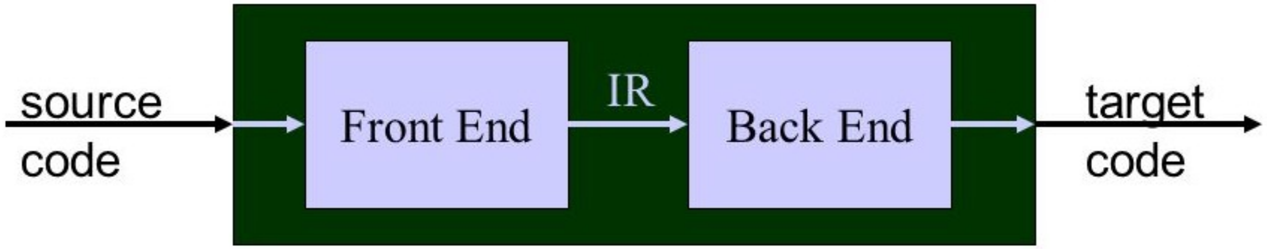
1. Catch these errors early before we try to generate target code and spend considerable time doing so.
2. If necessary, generate meaningful error messages to the programmer.

Type Checking Example: `a ← a x 2 x b x c x d`

Should this code type check correctly?

Overview of a Compiler

- Source code is any language.
- Target code is assembly, byte-code, etc.



IR: Intermediate Representation

After the compiler finishes each of the front end phases, it will transform source code into an IR (*intermediate representation*).

Various IR forms:

1. Graph - e.g. trees, ASTs
2. Sequential assembly code

Back End

The back end of a compiler typically deals with translation and optimization.

Q: Why would we want to optimize our programs?

Automatically modify (or optimize) programs so that they...

- run faster
- use less memory
- perhaps use less power?
- perhaps use less network access

Optimization

Optimization Example

One possible optimization rule for our consideration to potentially include in our compiler:

- Transform instances of `X = Y * 0` into `X = 0`

Rule seems like a real improvement, right?

But, this is not a "correct" optimization!

It is not always obvious when certain optimizations would actually break code.

Optimization Example

`X = Y * 0        =>        X = 0`

- Turns out: rule is valid for integers
  - ... but not for floats.
- In IEEE floating pt. standard: a special number is defined, `NaN` (not a number).
  - *Definition:* `NaN * 0 = NaN`
  - *So:* `NaN * 0 != 0`
- This optimization would break algorithms that rely on the proper propagation of `NaN`.
- Would break the "The NaN method of partial vector compares" that avoids conditional tests on all elements.

Another Optimization Example

Original Code

```
b ← ...
c ← ...
a ← 1
for i = 1 to n
    read d
    a ← a x 2 x b x c x d
end
```

How could this code be improved?

Improved Code

?

- What are the number of steps needed to do *multiplication*, from a register allocation perspective?
- Importance of ISA: Instruction Set Architecture

Register Allocation & Code Generation

Translate this line of code into assembly:

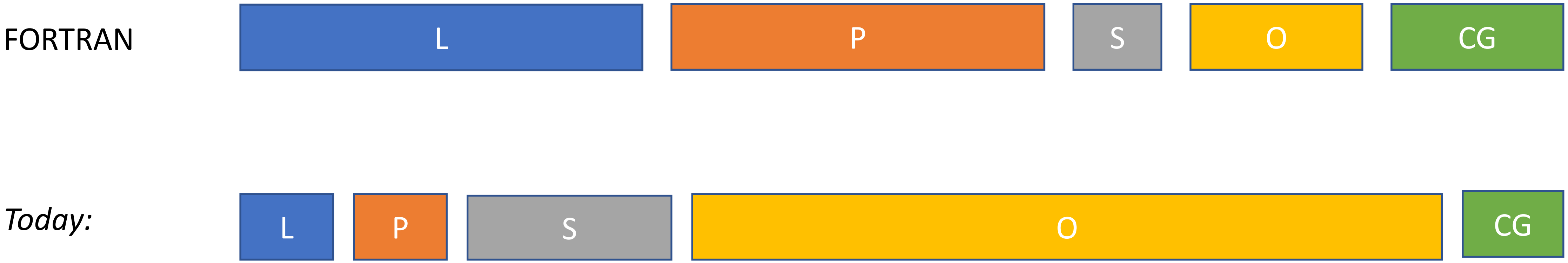
`a ← a x 2 x b x c x d`

Initially, we use temporaries instead of registers.

Some concerns:

- machines have limited number of fast registers
- will have to use slow memory if we run out of registers
- can we optimize our usage of registers?

- We are currently using <u>10 temporaries</u>.
- What is the least number of registers we would actually need for this code? It would be BAD if the machine didn't have enough registers.



- *Today:* We still do not have great tools for optimization.

Optimization is Hard!

One Big Example

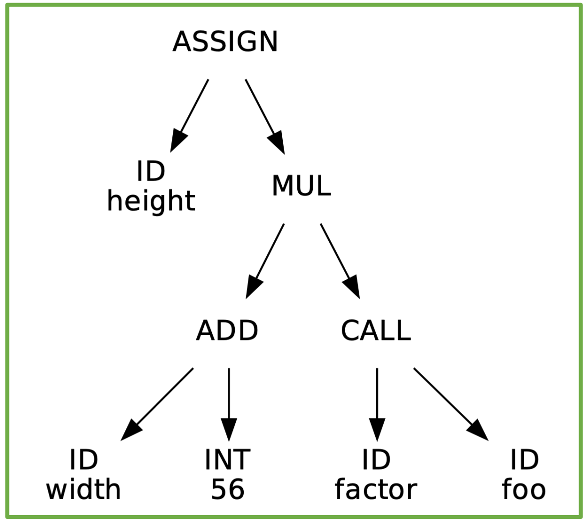
Suppose we wish to compile this fragment of code into assembly:

```
height = (width+56) * factor(foo);
```

After Scanning:

id:height	**=**	**(**	**id:width**	**+**	**int:56**	**)**	*****	**id:factor**	**(**	**id:foo**	**)**	**;**

After Parsing:



Intermediate Representation:

```
LOAD $56      -> r1
LOAD width    -> r2
IADD r1, r2   -> r3
ARG  foo
CALL factor   -> r4
IMUL r3, r4   -> r5
STOR r5      -> height
```

Assembly Code:

```
MOVQ    width, %rax      # load width into rax
ADDQ    $56, %rax        # add 56 to rax
MOVQ    %rax, -8(%rbp)   # save sum in temporary
MOVQ    foo, %edi        # load foo into arg 0 register
CALL    factor           # invoke factor, result in rax
MOVQ    -8(%rbp), %rbx   # load sum into rbx
IMULQ   %rbx             # multiply rbx by rax
MOVQ    %rax, height     # store result into height
```

More Next Class!

Deep dive into Scanning (the first phase of the front end)