CS406: Compilers

Spring 2022

Week1: Overview, Structure of a compiler

Why Study Compilers?

[Job Posting: 2021 LLVM Developers' Meeting (swoogo.com)]

- Few disciplines with **deep theory + practice**

"..Theory and practice are two sides of the same coin.." - Jeff Ullman, ACM Turing Award lecture.

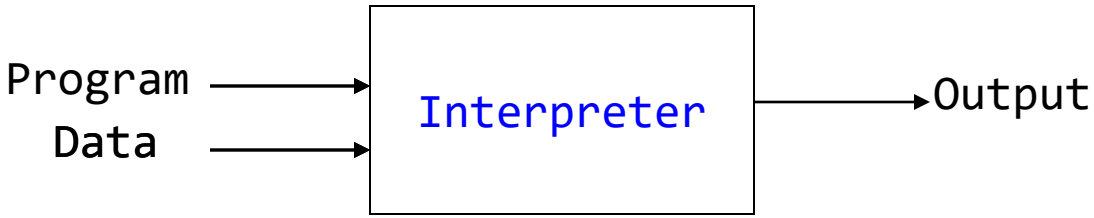
Intro to Compilers

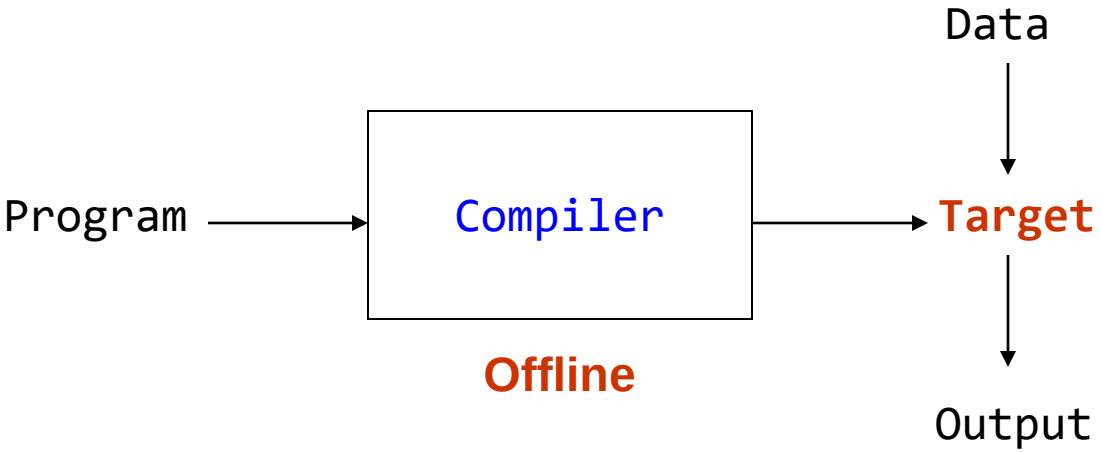
- One way to implement *programming languages*
  - Programming languages are notations for specifying computations to machines

  - *Target* can be an assembly code, executable, another source program etc.

Intro to Compilers

- Alternate way to implement programming languages



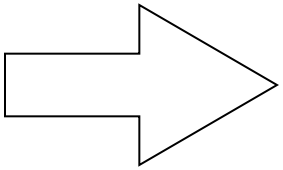


these are the two types of language processing systems

What is a Compiler?

Traditionally: Program that analyzes and **translates** from a high-level language (e.g. C++) to low-level assembly language that can be executed by the hardware

```
int a, b;
a = 3;
if (a < 4) {
    b = 2;
} else {
    b = 3;
}
```



```
    var a
    var b
    mov 3 a
    mov 4 r1
    cmpi a r1
    jge l_e
    mov 2 b
    jmp l_d
l_e:mov 3 b
l_d:;done
```


slide courtesy: Milind Kulkarni

Compilers are *translators*

- Fortran
- C
- C++
- Java
- Text processing language
- HTML/XML
- Command & Scripting Languages
- Natural Language
- Domain Specific Language

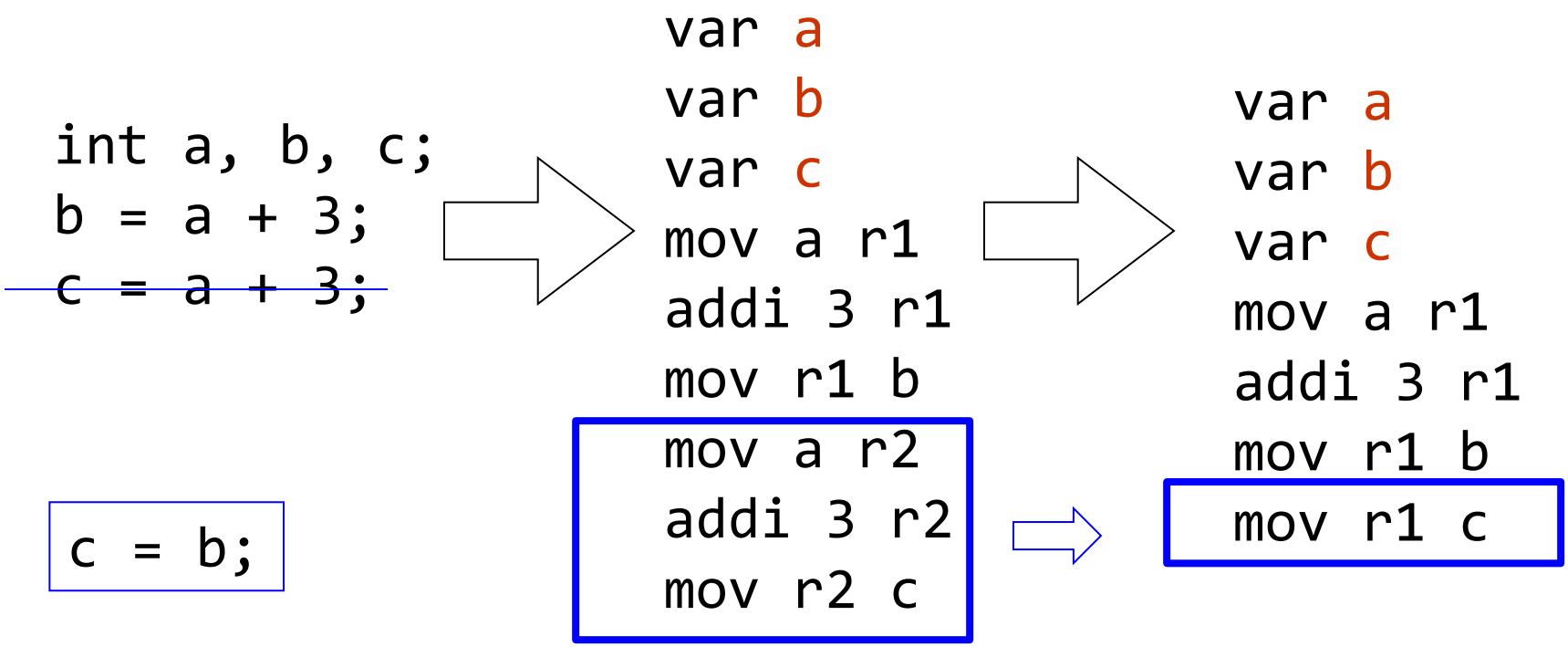
translate

- Machine code
- Virtual machine code
- Transformed source code
- Augmented source code
- Low-level commands
- Semantic components
- Another language

slide courtesy: Milind Kulkarni

Compilers are *optimizers*

- Can perform optimizations to make a program more efficient



slide courtesy: Milind Kulkarni

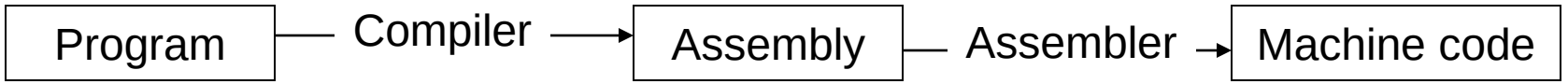
Compilers as Translators

1. High level language $\Longrightarrow$ assembly language **(e.g. gcc)**
2. High level language $\Longrightarrow$ machine independent bytecode **(e.g. javac)**
3. Bytecode $\Longrightarrow$ native machine code **(e.g. java's JIT compiler)**
4. High level language $\Longrightarrow$ High level language

   (e.g. domain-specific languages, many research languages)

How would you categorize a compiler that handles SQL queries?

HLL to Assembly



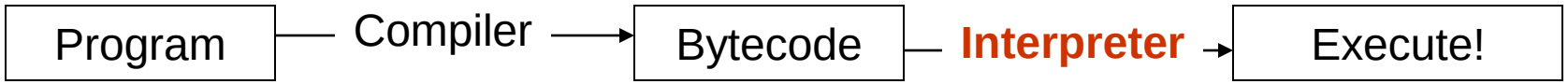
- Compiler converts program to assembly
- Assembler is machine-specific translator which converts assembly to machine code

```
add $7 $8 $9 ($7 = $8 + $9 ) => 000000 00111 01000 01001 00000 100000
```

- Conversion is usually one-to-one with some exceptions
  - Program locations
  - Variable names

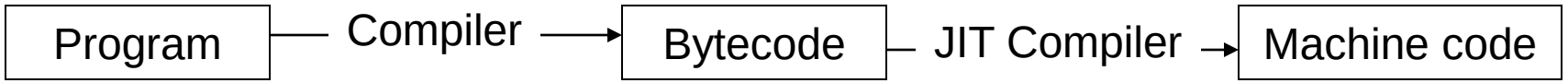
slide courtesy: Milind Kulkarni

HLL to Bytecode



- Compiler converts program into machine independent bytecode
  - e.g. javac generates Java bytecode, C# compiler generates CIL
- Interpreter then executes bytecode “on-the-fly”
- Bytecode instructions are “executed” by invoking methods of the interpreter, rather than directly executing on the machine
- Aside: what are the pros and cons of this approach?

HLL to Bytecode to Assembly



- Compiler converts program into machine independent bytecode
  - e.g. javac generates Java bytecode, C# compiler generates CIL
- Just-in-time compiler compiles code *while program executes* to produce machine code
  - Is this better or worse than a compiler which generates machine code directly from the program?

slide courtesy: Milind Kulkarni

Why do we need compilers?

- Compilers provide *portability*
- Old days: whenever a new machine was built, programs had to be rewritten to support new instruction sets
- IBM System/360 (1964): Common Instruction Set Architecture (ISA) --- programs could be run on any machine which supported ISA
  - Common ISA is a huge deal (note continued existence of x86)
- But still a problem: when new ISA is introduced (EPIC) or new extensions added (x86-64), programs would have to be rewritten
- Compilers bridge this gap: write new compiler for an ISA, and then simply recompile programs!

slide courtesy: Milind Kulkarni

Why do we need compilers?

- Compilers enable high-performance and productivity
- Old: programmers wrote in assembly language, architectures were simple (no pipelines, caches, etc.)
  - Close match between programs and machines --- easier to achieve performance
- New: programmers write in high level languages (Ruby, Python), architectures are complex (superscalar, out-of-order execution, multicore)
- Compilers are needed to bridge this *semantic gap*
  - Compilers let programmers write in high level languages and still get good performance on complex architectures

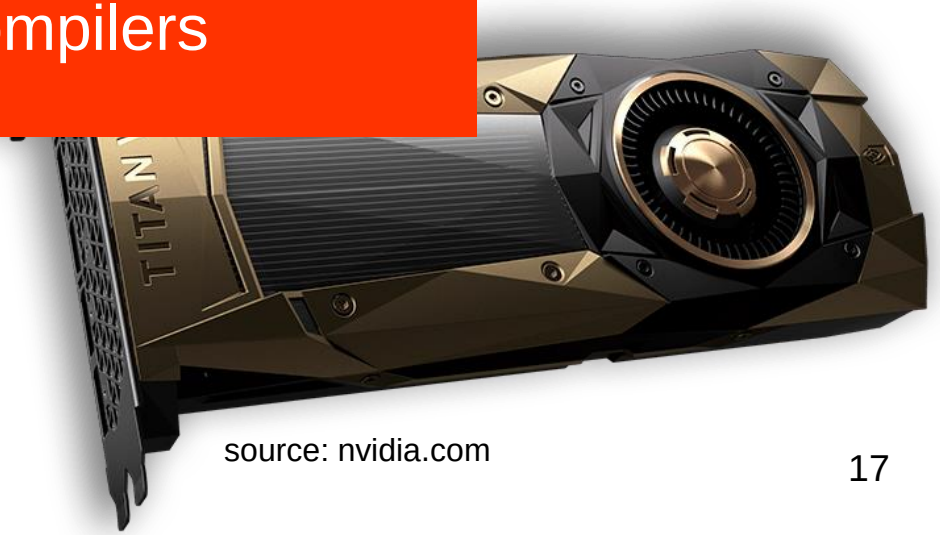
slide courtesy: Milind Kulkarni

Semantic Gap

- Python code that actually runs on GPU

```
import pycuda
import pycuda.autoinit from pycuda.tools import
make_default_context
c = make_default_context()
d = c.get_device()
……
```

Impossible without Compilers



source: nvidia.com

History

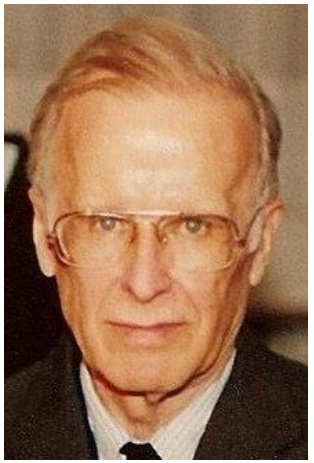
- 1954: IBM 704
  - Huge success
  - Could do complex math
  - Software cost > Hardware cost

Source: IBM Italy, https://commons.wikimedia.org/w/index.php?curid=48929471

How can we improve the efficiency of creating software?

History

- 1953: Speedcoding
  - *High-level programming language* by John Backus
  - Early form of *interpreters*
  - Greatly reduced programming effort



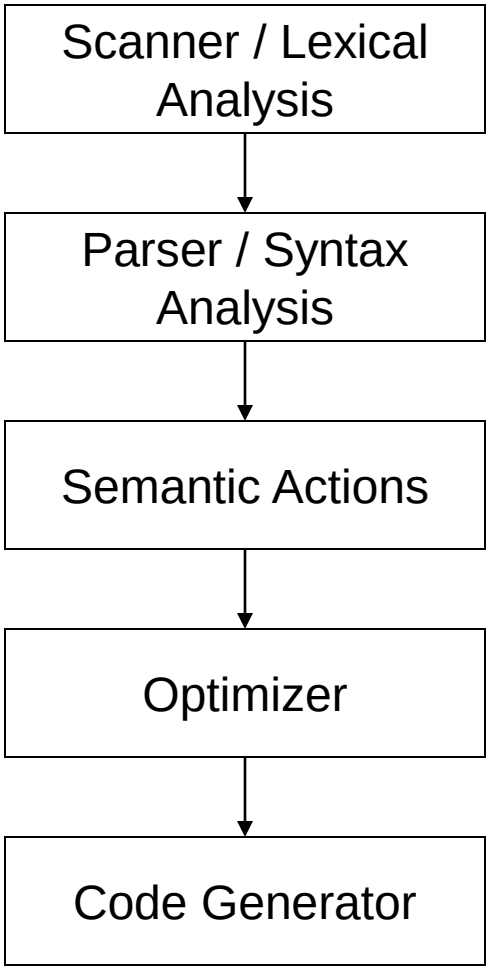
  - About 10x-20x slower
  - Consumed lot of memory (~300 bytes = about 30% RAM)

Fortran I

- 1957: Fortran released
  - Building the compiler took 3 years
  - Very successful: by 1958, 50% of all software created was written in Fortran
- Influenced the design of:
  - high-level programming languages e.g. BASIC
  - practical compilers

Today's compilers still preserve the structure of Fortran I

Structure of a Compiler



Scanner

- Analogy: Humans processing English text

```
Rama is a neighbor.

Ra mais an eigh bor.
```

You have to do some work to align the spaces and understand the sentence.

Scanner

- Consider the program text

```
if ( a < 4) {
    b = 5
}
```

  - Has tokens that are:
    1. keywords – `if`
    2. Punctuation marks – `(, ), {, }, blankspaces, tab space (\t), newlines (\n)`
    3. Identifiers – `a, b`
    4. Constants/Literals – `4, 5`
    5. Operators - `<, =`

Scanner

- A compiler starts by seeing only program text

```
if ( a < 4) {
    b = 5
}
```

- as a series of letters

```
‘i’ ‘f’ ‘ ’ ‘(’ ‘a’ ‘<’ ‘4’ ‘)’
 ‘ ’ ‘{’ ‘\n’ ‘\t’ ‘b’ ‘=’ ‘5’
          ‘\n’ ‘}’
```

Scanner

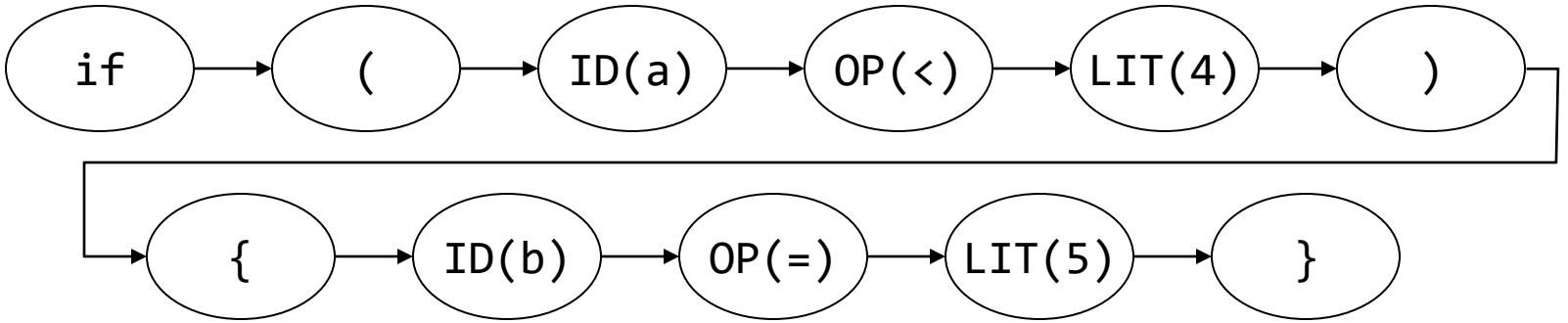
- A compiler starts by seeing only program text
- Scanner converts program text into string of *tokens*

```
‘i’ ‘f’ ‘ ’ ‘(’ ‘a’ ‘<’ ‘4’ ‘)’
 ‘ ’ ‘{’ ‘\n’ ‘\t’ ‘b’ ‘=’ ‘5’
          ‘\n’ ‘}’
```

- Analogy: Humans processing English text
  - recognize words in `Rama is a neighbor.`
    - Rama, is, a, neighbor
    - Additional details such as punctuations(.), capitalizations (R), blank spaces.

Scanner - Summary

- A compiler starts by seeing only program text
- Scanner converts program text into string of *tokens*



- But we still don't know what the *syntactic structure* of the program is

slide courtesy: Milind Kulkarni

Exercise

Convert the following program text into tokens:

```
c = a + b * 60
```

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Chapter 1 (Sections: 1.1 to 1.3, 1.5)
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 1 (Sections 1.1 to 1.3, 1.5)