



NEW YORK UNIVERSITY

G22.2130-001

Compiler Construction

Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu



Copyright © Randy Glasbergen. www.glasbergen.com

Who Am I?



- Mohamed Zahran (aka Z)
- Computer architecture/OS/Compilers Interaction
- <http://www.mzahran.com>
- Office hours: Thursdays 4:30-6:30 pm
- Room: CIWW 328
- Course web page:
<http://www.cs.nyu.edu/courses/spring10/G22.2130-001/>

Formal Goals of This Course

- What exactly is this thing called compiler?
- How does the compiler interact with the hardware and programming languages?
- Different phases of a compiler
- Develop a simple compiler

Informal Goals of This Course

- To get more than an A
- To learn compilers and enjoy it
- To use what you have learned in *MANY* different contexts

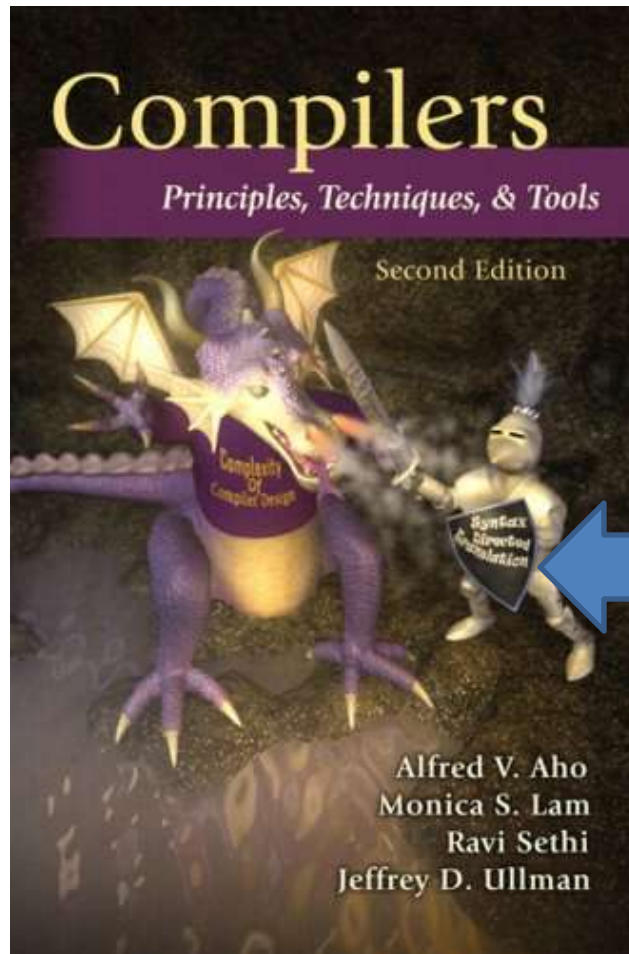
Grading

- **The project (labs):** **60%**
 - Due several lectures later
 - Several parts
 - Mostly programming and dealing with tools
 - Can do on your own machine or NYU machines
 - Must be sure that they work on NYU machines
 - Getting help: office hours, mailing list, and FAQ
- **Final exam:** **40%**

The Course Web Page

- Lecture slides
- Info about mailing list, labs,
- FAQ
- Useful links (manuals, tools, book errata, ...).
- Interesting links (geeky stuff!)

The Book



- The classic definitive compiler technology text
- It is known as the **Dragon Book**
- A knight and a dragon in battle, a metaphor for conquering complexity
- We will cover mostly chapters 1 - 8

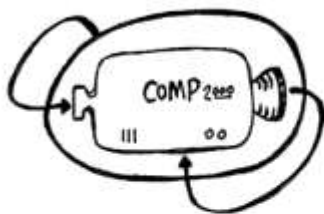
Hey Kids!!

It's

COMPILER TIME!



A **COMPILER** IS A
PROGRAM THAT,
WHEN FED ITSELF
AS INPUT, PRODUCES
ITSELF.



PEOPLE
ARE
PROGRAMS...



I'm a
compiler!

What Is A Compiler?

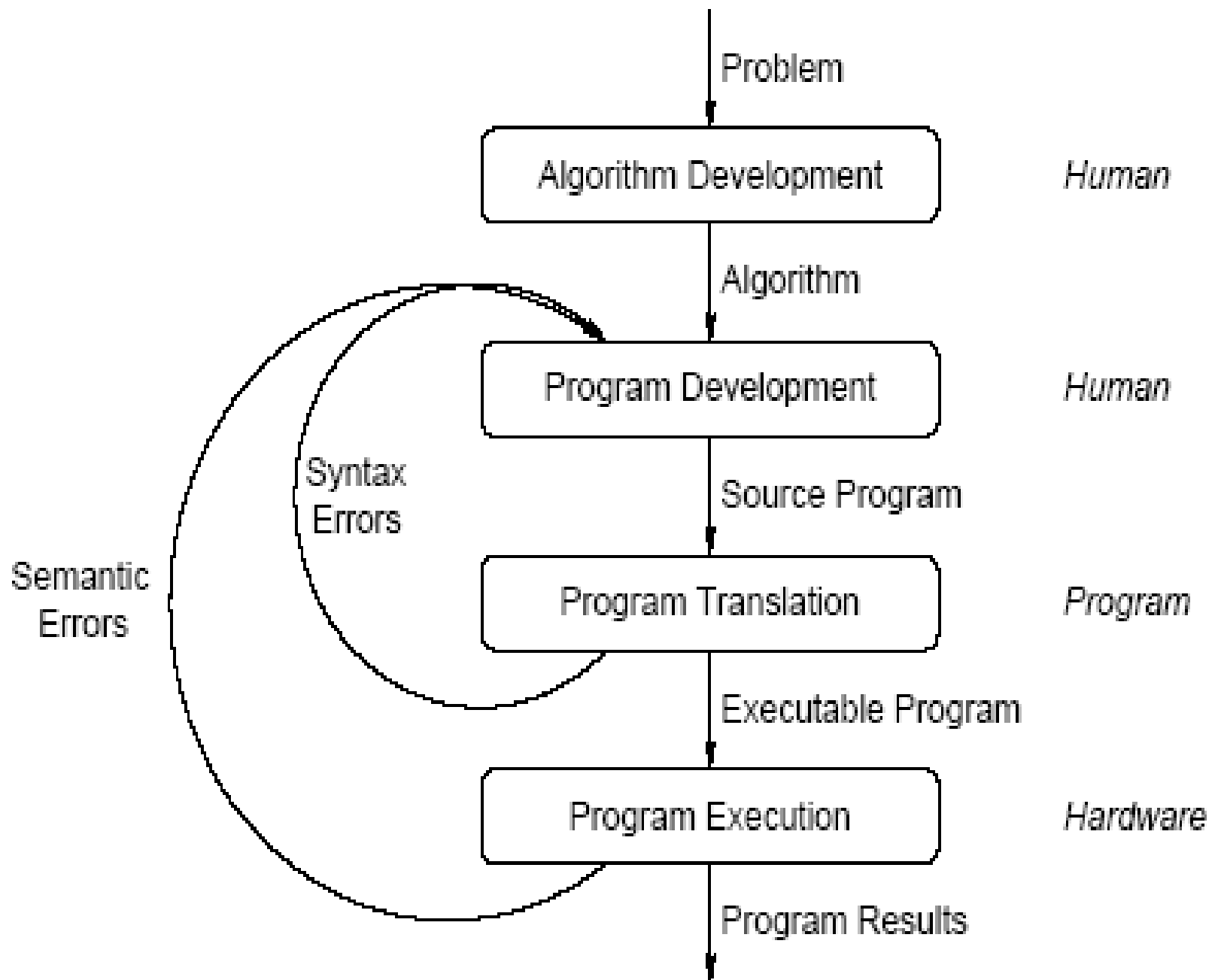
- Programming languages are **notations** for **describing computations** to **people** and to **machines**
- Machines do not understand programming languages
- So a software system is needed to do the translation
- This is the **compiler**

BEAR FACTS

by Burke



He always liked to open the Annual Programmer's Conference with a joke in binary.

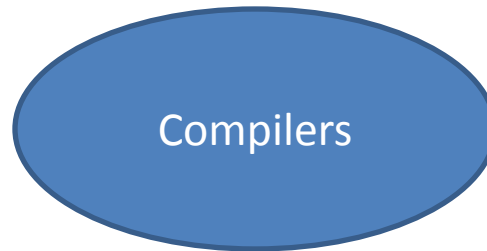


Machine Architecture

Compiler writers
must take advantage
of new hardware features

Programming Languages

Compiler writers have to track
new language features



Language Theory

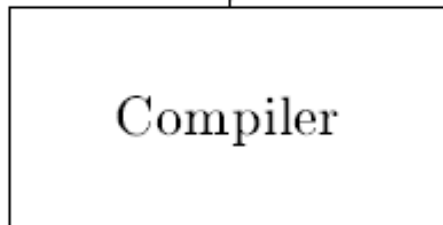
Software Engineering

- Optimizing compilers are hard to build
- Excellent software engineering case study
- Theory meets practice

Why Compilers Are So Important?

- Compiler writers have influence over all programs that their compilers compile
- Compiler study trains good developers
- We learn not only how to build compilers but the general methodology of solving complex and open-ended problems
- Compilation technology can be applied in many different areas
 - Binary translation
 - Hardware synthesis
 - DB query interpreters
 - Compiled simulation

source program



Compiler



target program

input

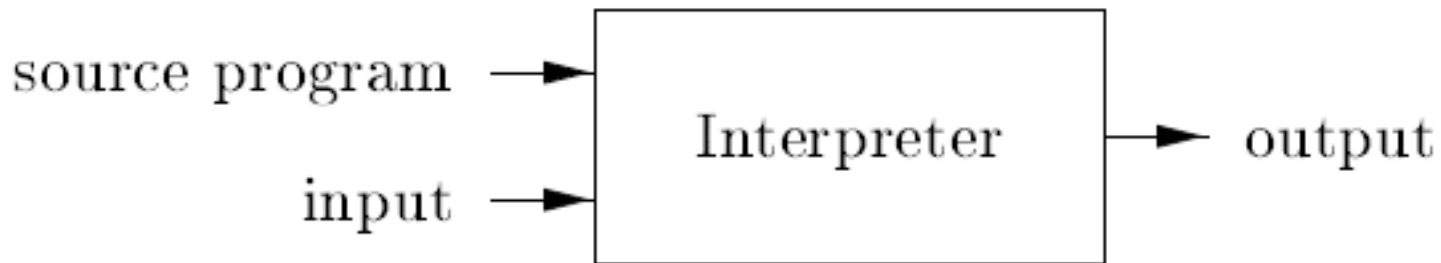


Target Program

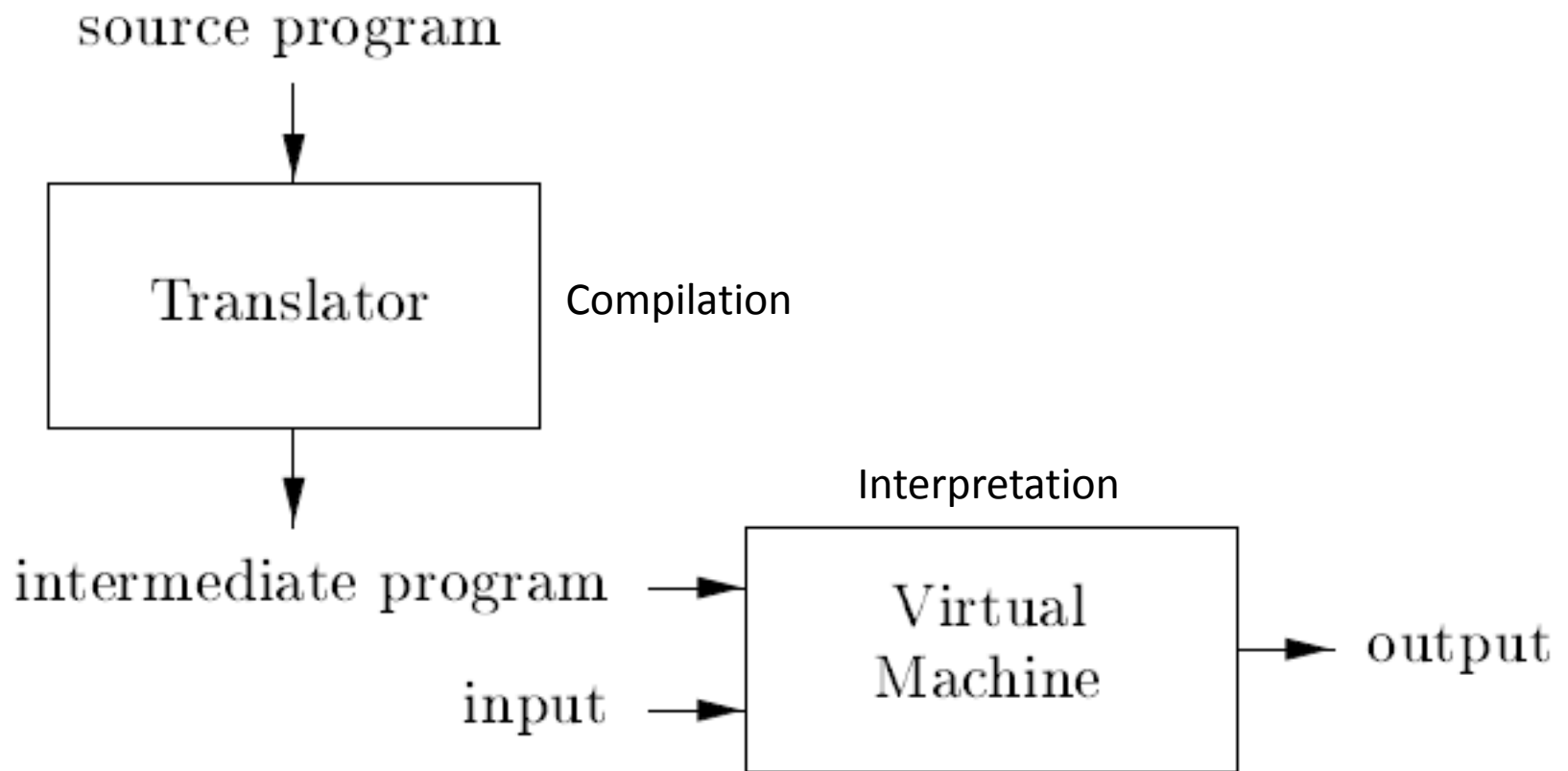


output

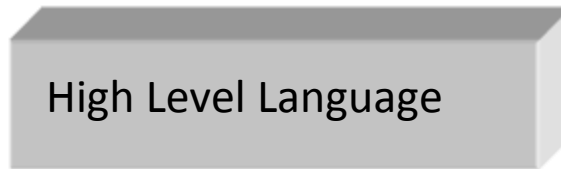
So What Is An Interpreter?



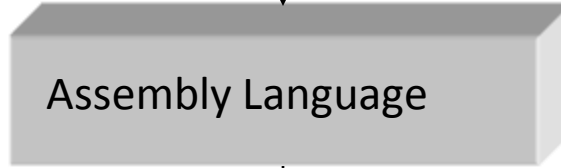
- + Better error diagnostics than compiler
- Slower than machine language code directly executed on the machine



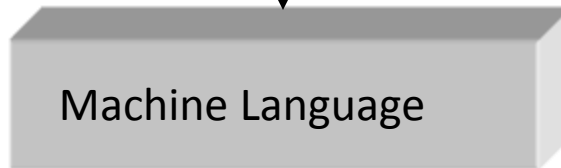
Problem → Algorithm Development → Programmer



Compiler (translator)



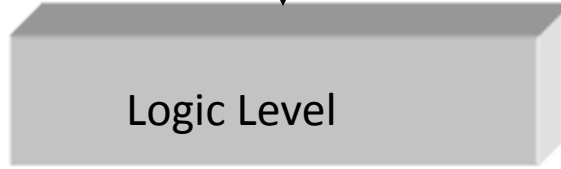
Assembler (translator)



Control Unit (Interpreter)

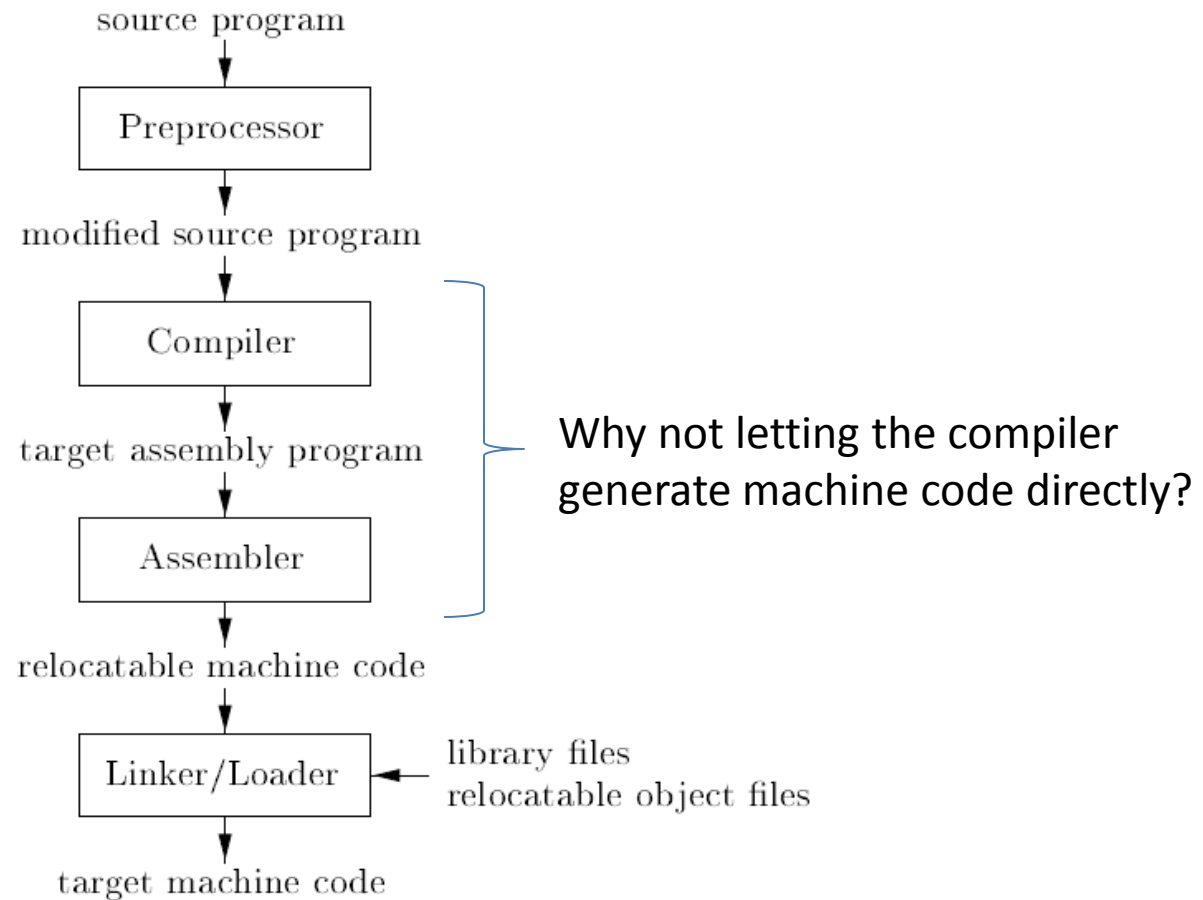


Microsequencer (Interpreter)

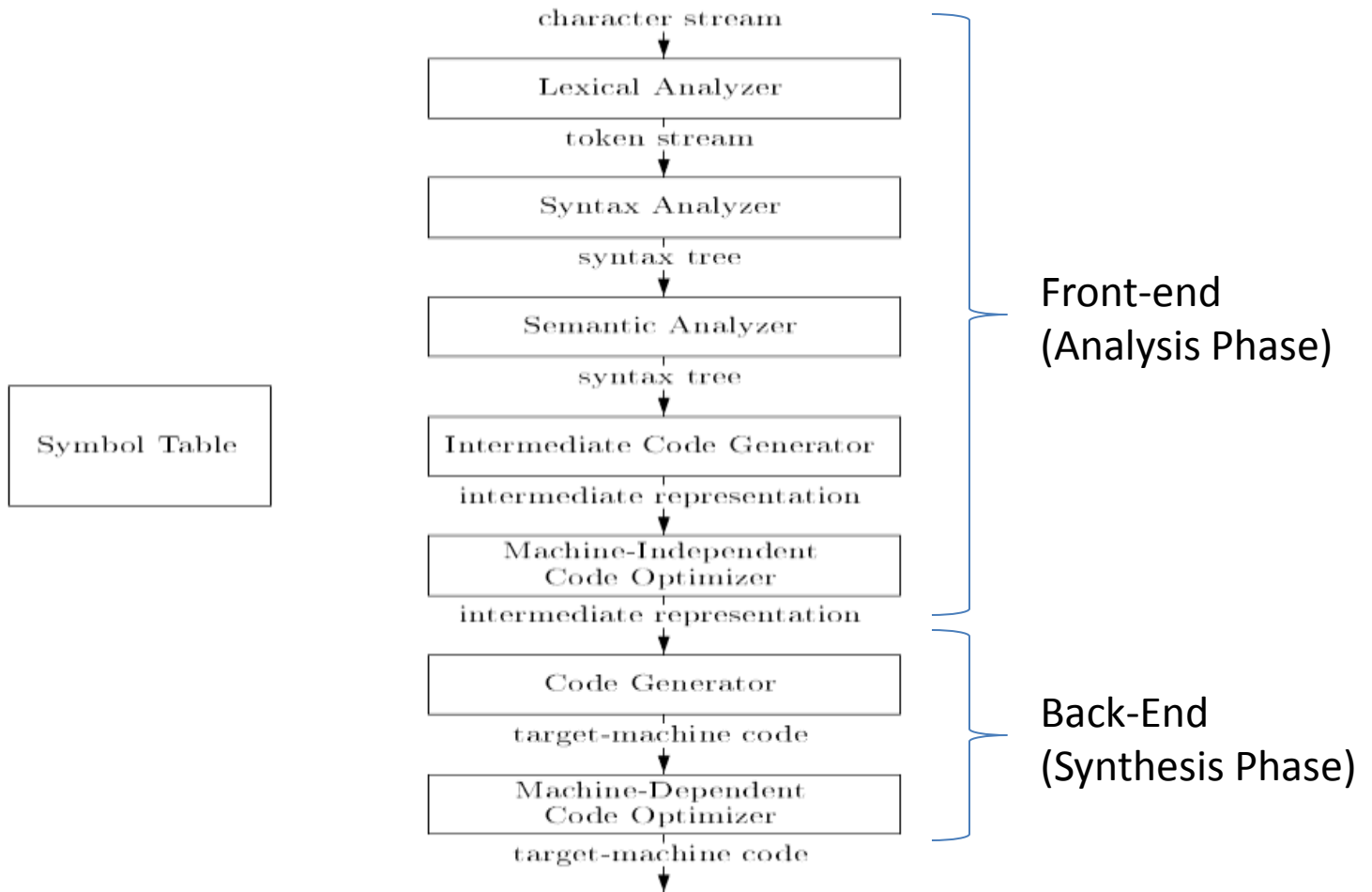


Device Level → Semiconductors → Quantum

Compiler Is Not A One-Man-Show!



Let's Have A Closer Look: Phases of A Compiler



Lexical Analysis

- Reads stream of characters: your program
- Groups the characters into meaningful sequences: **lexemes**
- For each lexeme, it produces a **token**
<token-name, attribute value>
- Blanks are just separators and are discarded
- Filters comments
- Recognizes: keywords, identifier, numbers, ...

position = initial + rate * 60

Lexical Analyzer

$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

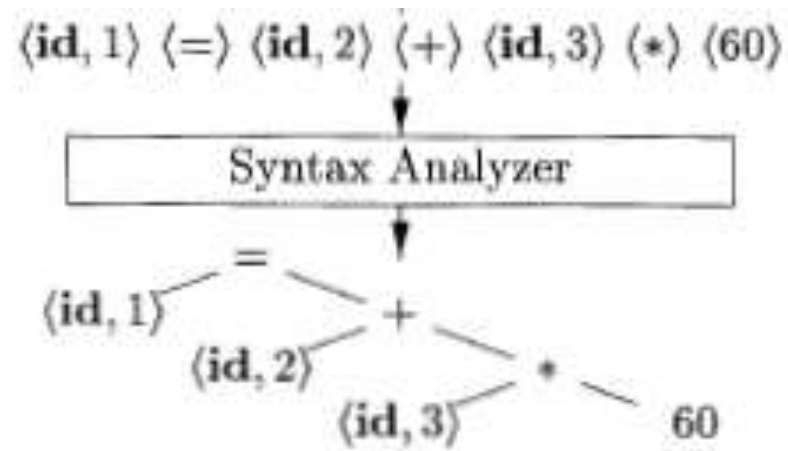
Token stream

token name

Entry into the symbol table

Syntax Analysis (Parsing)

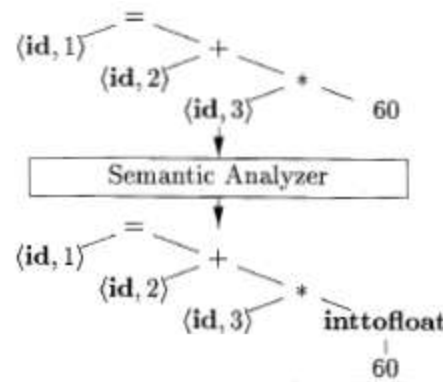
- Uses tokens to build a tree
- The tree shows the grammatical structure of the token stream
- A node is usually an operation
- Node's children are arguments



This is usually called a **syntax tree**

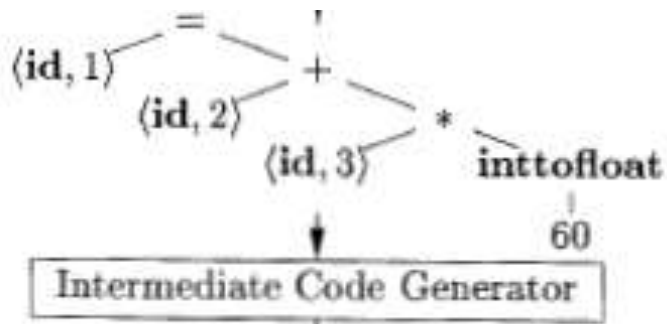
Semantic Analysis

- Uses the syntax tree and symbol tables
- Gathers type information
- Checks for semantic consistency errors



Intermediate Code Generation

- Code for an abstract machine
- Must have two properties
 - Easy to produce
 - Easy to translate to target language



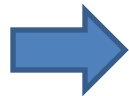
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

- Called three address code
- One operation per instruction at most
- Compiler must generate temporary names to hold values

Intermediate Code Optimization (Optional)

- Machine independent
- optimization so that better target code will result

```
t1 = inttofloat (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



```
t1 = id3 * 60.0
t2 = id2 + t1
id1 = t2
```



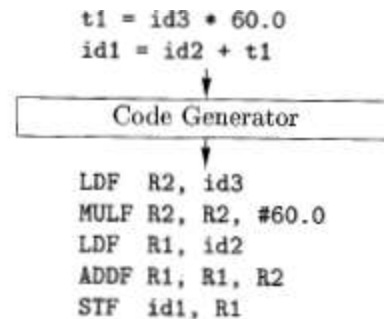
```
t1 = id3 * 60.0
id1 = id2 + t1
```

Instead of inttofloat
we can use 60.0 directly

Do we really need t2?

Code Generation

- Input: the intermediate representation
- Output: target language
- This is the backend, or synthesis phase
- Machine dependent



Qualities of a Good Compiler

- **Correct:** the meaning of sentences must be preserved
- **Robust:** wrong input is the common case
 - compilers and interpreters can't just crash on wrong input
 - they need to diagnose all kinds of errors safely and reliably
- **Efficient:** resource usage should be minimal in two ways
 - the process of compilation or interpretation itself is efficient
 - the generated code is efficient when interpreted
- **Usable:** integrate with environment, accurate feedback
 - work well with other tools (editors, linkers, debuggers, . . .)
 - descriptive error messages, relating accurately to source

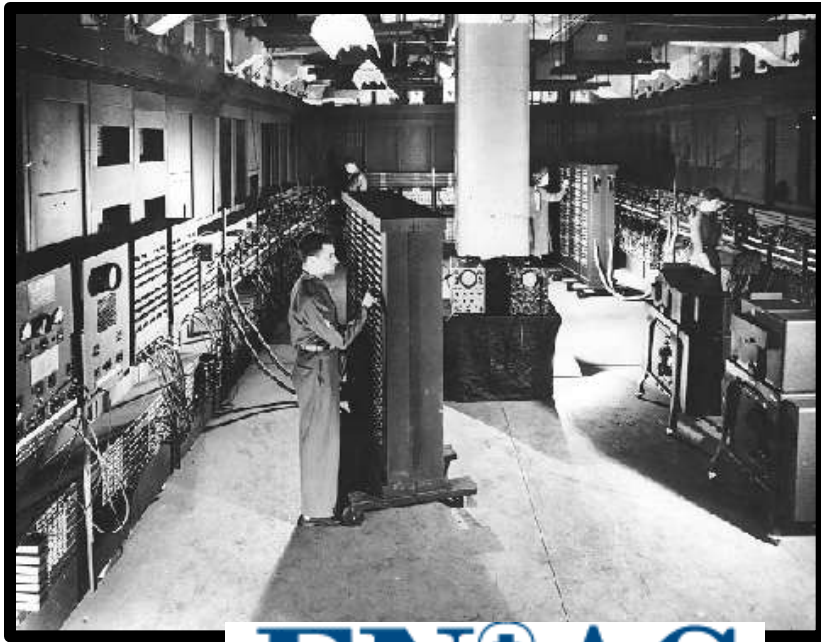
Compilers Optimize Code For:

- Performance/Speed
- Code Size
- Power Consumption
- Fast/Efficient Compilation
- Security/Reliability
- Debugging

Compiler Construction Tools

- Parser generators: automatically produce syntax analyzer
- Scanner generators: produce lexical analyzer
- Syntax-directed translation engines: collection of routines for walking a parse tree and generate intermediate code
- and many more ...

A Little Bit of History



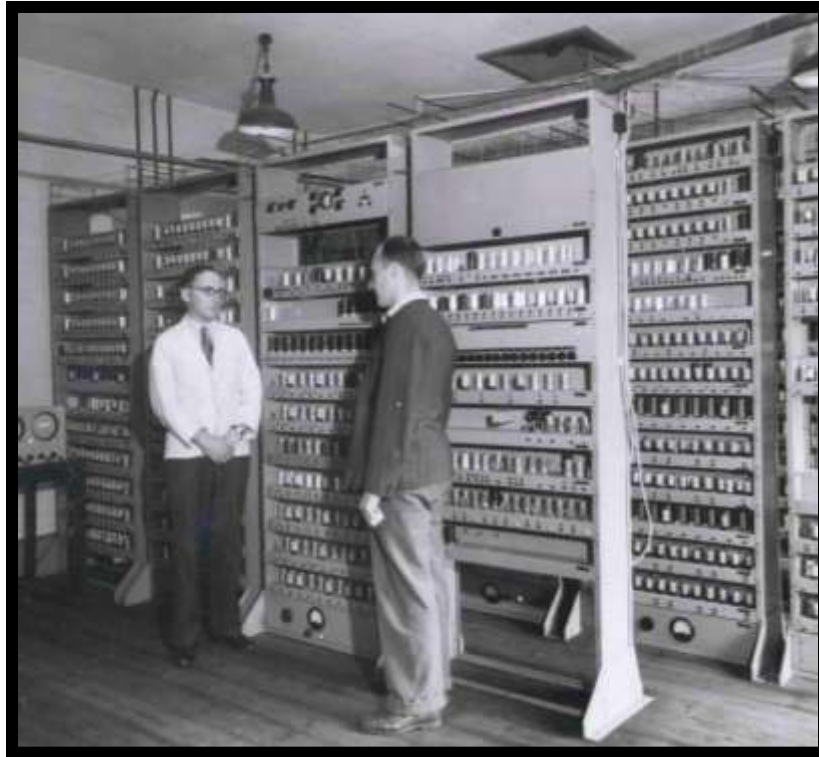
ENIAC

Eckert and Mauchly



- 1st working electronic computer (1946)
- 18,000 Vacuum tubes
- 1,800 instructions/sec
- 3,000 ft³

A Little Bit of History



EDSAC 1 (1949)

<http://www.cl.cam.ac.uk/UoCCL/misc/EDSAC99/>

- Maurice Wilkes



1st stored program
computer

650 instructions/sec

1,400 ft³

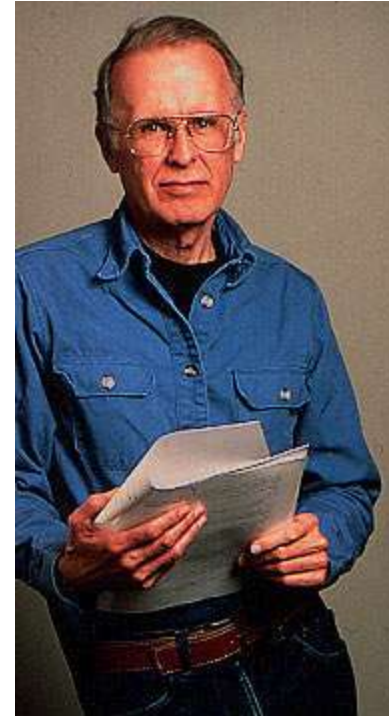
A Little Bit of History

- 1954 IBM developed 704
- All programming done in assembly
- Software costs exceed hardware costs!



A Little Bit of History

- Fortran I (project 1954-57)
- The main idea is to translate high level language to assembly
- Many thought this was impossible!
- In 1958 more than 50% of software in assembly!
- Development time halved!



John Backus
(December 3, 1924 – March 17, 2007)

A Glimpse At Programming Language Basics

- Static/Dynamic distinction
- Environments and states
 - Environment: mapping names to locations
 - States: mapping from location to values
- Procedures vs Functions
- Scope

```
main() {  
    int a = 1; B1  
    int b = 1;  
    {  
        int b = 2; B2  
        {  
            int a = 3; B3  
            cout << a << b;  
        }  
        {  
            int b = 4; B4  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```

Figure 1.10: Blocks in a C++ program

Roadmap

- Today we have mostly discussed chap 1
- Chapter 2 gives an overview of the different phases of a compiler by building the front-end of a simple compiler
- Chapters 3-8 fill the gaps