Who?

Shachar Itzhaky
shachari@cs.technion.ac.il
Office hours should be scheduled in advance

TAs:
• Omer Katz
• Alex Sivak
• Yaakov Sokolik
What?

- Understand:
  - What a compiler is
  - How it works
  - Proven techniques
    (most can be re-used in other settings)
How?

• What will help us:
  ▸ Textbooks
    • Modern Compiler Design
    • Compilers: Principles, Techniques & Tools
    • Modern Compiler Implementation in C
  ▸ Homework assignments
    • “Dry”: deepen understanding of theory
    • “Wet”: build a compiler yourself
  ▸ Ask questions, answer questions!
How Not?!

• “Your slides don’t have everything you say written on them” (common complaint)
  ▸ Yes, I know, this is by design
  ▸ Presentations are a teaching aid, not a substitute for coming to lectures
• If you don’t attend lectures or attend and don’t listen, you will inevitably miss some things
• If you want slides that have all the material written on them nicely, that format is indeed available and commonly known as a textbook
• See how horrible this slide is? This is why you won’t see many slides with as much text as this one for the rest of the course
Exam

• 75% of the final grade
• Look at Eran’s old exams from previous years
• Don’t worry too much...
  ▶ If you attend lectures and finish the assignments, you should do well in the exam; if you don’t attend try to keep up with the material
  ▶ Historical evidence — attending leads to higher pass rate in the final exam
"The Education of a Computer"
(Grace Hopper)

UNIVAC
"The Education of a Computer"
(Grace Hopper)

PROBLEM

FORMULAS

TABLES

CODE

PROGRAMMER

BRAIN

PROGRAM

UNIVAC

RESULTS

STANDARD

KNOWLEDGE

INPUT

DATA

Fig. 4 - SOLUTION OF A PROBLEM
As soon as the purpose is stated to make use of subroutines, two methods arise. In one, the program refers to an immediately available subroutine, uses it, and continues computation. For a limited number of subroutines, this method is feasible and useful. Such a system has been developed under the nickname of the "short-order code" by members of the staff of the Computational Analysis Laboratory.

The second method not only looks up the subroutine, but translates it, properly adjusted, into a program. Thus, the completed program may be run as a unit whenever desired, and may itself be placed in the library as a more advanced subroutine.

Each problem must be reduced to the level of the available subroutines. Suppose a simple problem, to compute \( y = \sin(x) \), using elementary subroutines. Each step of the formula falls into the operational pattern, Fig. 8; that is,

\[
\begin{align*}
U &= e^{-u} \\
V &= \sin(v) \\
y &= U + V
\end{align*}
\]
John Backus and team at IBM

The first complete compiler
John Backus and team at IBM

The first complete compiler
What is a Compiler?

- “A compiler is computer software that transforms computer code written in one programming language (the source language) into another language (the target language). …primarily to a lower level language (e.g. assembly language, object code, or machine code) to create an executable program.”

-- Wikipedia
What is a Compiler?

source language

Executable code

Source text

target language

Executable code

"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

exe
txt
What is a Compiler?

source language

C
C++
Pascal
Java
Perl
JavaScript
Python
Ruby
Prolog
Lisp
Scheme
ML
OCaml
Postscript
TeX

target language

IA32
IA64
ARM
SPARC
Java Bytecode
C
C++
Pascal
Java
PDF
Bitmap
...
What is a Compiler?

Compiler

```
int a, b;
a = 2;
b = a*2 + 1;
```
Anatomy of a Compiler

Compiler

Source text

Frontend (analysis)
Semantic Representation
Backend (synthesis)

Executable code

int a, b;
a = 2;
b = a*2 + 1;

MOV R1,2
SAL R1
INC R1
MOV R2,R1
```cpp
int a, b;
a = 2;
b = a*2 + 1;
```
Compiler vs. Interpreter

- **Source text**
  - Frontend (analysis)
  - Semantic Representation
  - Backend (synthesis)

- **Executable code**

- **Input**
  - Frontend (analysis)
  - Semantic Representation
  - Execution Engine

- **Output**
Compiler vs. Interpreter

- **Frontend (analysis)**
- **Semantic Representation**
- **Backend (synthesis)**

```
b = a*2 + 1;
```

**Execution Engine**

```
MOV R1, 2
SAL R1
INC R1
MOV R2, R1
```

Output:
3
7

```
b = a*2 + 1;
```

Output:
3
7
Just-in-time (JIT) compilation: bytecode interpreter (in the JVM) compiles program fragments during interpretation to avoid expensive re-interpretation.
Just-in-time Compiler
(Javascript example)
Just-in-time Compiler
(Javascript example)
Just-in-time Compiler
(Javascript example)

- The compiled code is optimized dynamically at runtime, based on runtime behavior
**Anatomy of a Compiler: Why?**

```
int a, b;
a = 2;
b = a*2 + 1;
```

```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```
Modularity

```plaintext
int a, b;
a = 2;
b = a*2 + 1;
```

```
SET   R1, 2
STORE #0, R1
SHIFT R1, 1
STORE #1, R1
STORE #2, R1
ADD   R1, 1
ADD   R1, 1
ADD   R1, 1
ADD   R1, 1
ADD   R1, 1
ADD   R1, 1
ADD   R1, 1
ADD   R1, 1
ADD   R1, 1
ADD   R1, 1
```

```
MOV R1, 2
SAL R1
INC R1
MOV R2, R1
```
Anatomy of a Compiler

Compiler

Source text

Preprocessing

Frontend (analysis)

Semantic Representation

Backend (synthesis)

Executable code

```
int a, b;
a = 2;
b = a*2 + 1;
```
int a, b;
a = 2;
b = a*2 + 1;
Anatomy of a Compiler

Source text

Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Representation (IR)
Optimizations
Code Generation

Compiler

Frontend (analysis)
Semantic Representation
Backend (synthesis)

Executable code

int a, b;
a = 2;
b = a*2 + 1;

MOV R1, 2
SAL R1
INC R1
MOV R2, R1
Why should you care?

• Every person in this class will build a parser some day
  – Or wish they knew how to build one...

• Better understanding of programming languages
• Understand internals of compilers
• Understand (some) details of target architectures

• Useful techniques and algorithms
  – Lexical analysis / parsing
  – Semantic representation
  – Abstraction layers
  – Modularity
Why should you care?

Source

Compiler

Target

- Useful formalisms
  - Regular expressions
  - Context-free grammars
- Data structures
- Algorithms

Programming Languages
Software Engineering

System Architecture
Runtime Environment
Virtual Machines
Garbage Collection
Course Overview

Compiler

Source text

 Lexical Analysis

 Syntax Analysis

 Semantic Analysis

 IR Optimization

 Code Generation

 Executable code
Journey inside a compiler

\[ x = b*b - 4*a*c \]
Journey inside a compiler

```
x = b*b - 4*a*c
```

Token Stream

```
<ID,”x”> <EQ> <ID,”b”> <MULT> <ID,”b”>
<MINUS> <INT,4> <MULT> <ID,”a”> <MULT> <ID,”c”>
```
Journey inside a compiler

...<ID,"b"> <MULT> <ID,"b"> <MINUS> <INT,4> <MULT> <ID,"a"> <MULT> <ID,"c">
Journey inside a compiler

...<ID,"b"> <MULT> <ID,"b"> <MINUS> <INT,4> <MULT> <ID,"a"> <MULT> <ID,"c">

Abstract Syntax Tree

Journey inside a compiler

![Annotated Abstract Syntax Tree]

- **Lexical Analysis**
- **Syntax Analysis**
- **Sem. Analysis**
- **Opt. IR**
- **Code Gen.**
Journey inside a compiler

**Intermediate Representation**

R2 = 4 * a
R1 = b * b
R2 = R2 * c
R1 = R1 – R2
Journey inside a compiler

Lexical Analysis

Syntax Analysis

Sem. Analysis

Opt. IR

Code Gen.

Intermediate Representation

\[ R_2 = 4 \times a \]
\[ R_1 = b \times b \]
\[ R_2 = R_2 \times c \]
\[ R_1 = R_1 - R_2 \]

Assembly Code

\[ \text{MOV R2,(sp+8)} \]
\[ \text{SAL R2,2} \]
\[ \text{MOV R1,(sp+16)} \]
\[ \text{MUL R1,(sp+16)} \]
\[ \text{MUL R2,(sp+24)} \]
\[ \text{SUB R1,R2} \]
Error Checking

In every stage:

- **Lexical analysis**: illegal tokens
- **Syntax analysis**: illegal syntax
- **Semantic analysis**: incompatible types, undefined variables, ...
- Even **runtime**: division by zero, array bounds, ...

- Every phase tries to recover and proceed with compilation
Lexical Errors

- pi = 3.141.562
  - Illegal token

- pi = 3oranges
  - Illegal token

- pi = oranges3
  - ⟨ID,"pi"⟩, ⟨EQ⟩, ⟨ID,"oranges3"⟩
Syntax Errors

- $x = / \text{oranges}$: Wrong number of arguments to operator "/"
- $x = \text{func(int a)}$: A declaration is not expected here
Semantic Errors: Type Checking

x = 4*a*“oranges”

Type mismatch

Type mismatch

Type mismatch
Runtime Errors

$$x = \text{det(singular\_matrix)}$$
$$y = 100 / x$$

Division by zero

$$a = \text{new int}[9] \ // a[0..8]$$
$$b = a[\text{answer}]$$

Array index out of bounds: 42 > 8
The Real Anatomy of a Compiler

Source text

Process text input → Lexical Analysis

characters → tokens

Syntax Analysis → AST

Semantic Analysis

Intermediate code generation

Intermediate code optimization

Code generation

Target code optimization

Machine code generation

Write executable output

Executable code

Executable code

Symbolic Instructions

Annotated AST
The Real Anatomy of a Compiler

Front end

Source text

1. Process text input
2. Lexical Analysis
3. Syntax Analysis
4. Semantic Analysis

Intermediate code generation

Intermediate code optimization

Code generation

Back end

Target code optimization

Machine code generation

Write executable output

Executable code

Executable code
Optimizations

• “Optimal code” is out of reach
  – many problems are undecidable or too expensive (NP-complete)
  – Use approximation and/or heuristics
  – Must preserve correctness, should (mostly) improve code

• Many optimization heuristics
  – Loop optimizations: hoisting, unrolling, ...
  – Peephole optimizations: constant folding, strength reduction, ...
  – Constant propagation
    • Leverage compile-time information to save work at runtime (pre-computation)
  – Dead code elimination

• Majority of compilation time is spent in the optimization phase
Loop Hoisting

```c
for (int i = 0; i < 100; ++i) {
    array[i] = x + y;
}
```

```c
int t = x + y;
for (int i = 0; i < 100; ++i) {
    array[i] = t;
}
```
Loop Unrolling

```cpp
for (int i = 0; i < 100; ++i) {
    delete array[i];
}
```

```cpp
for (int i = 0; i < 100; i += 5) {
    delete array[i];
    delete array[i+1];
    delete array[i+2];
    delete array[i+3];
    delete array[i+4];
}
```
Machine code generation

• Register allocation
  ‣ Optimal register assignment is NP-Complete
  ‣ In practice, known heuristics perform well

• Assigning variables to memory locations

• Instruction selection
  ‣ Convert IR to actual machine instructions

• Modern architecture challenges
  ‣ Multicores
  ‣ Memory hierarchies
  ‣ SIMD instructions
Compiler Construction Toolset

- Lexical scanner generators
  - Flex
- Parser generators
  - Bison
Summary

✓ Compiler = a program that translates code from source language (high level) to target language (low level)

✓ Compilers play a critical role
  ‣ Bridge programming languages to the machine
  ‣ Many useful techniques and algorithms
  ‣ Many useful tools (e.g., lexer/parser generators)

✓ Compiler constructed from modular phases
  ‣ Reusable
  ‣ Different front/back ends
Lexical Analysis