Who?

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

TAs:
- Hila Peleg
- Omer Katz
- Avner Elizarov

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
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 grade on average

ENIAC

“The Education of a Computer” (Grace Hopper)
What is a Compiler?

- “A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language). The most common reason for wanting to transform source code is to create an executable program.”

-- Wikipedia
What is a Compiler?

Compiler

Source text → Executable code

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;

Interpreter

Interpreter

Source text → Frontend (analysis) → Semantic Representation → Execution Engine → Output

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

Compiler vs. Interpreter

Compiler

Source text → Frontend (analysis) → Semantic Representation → Backend (synthesis) → Executable code

Interpreter

Source text → Frontend (analysis) → Semantic Representation → Execution Engine → Output
Compiler vs. Interpreter

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

Just-in-time Compiler
(Java example)

- **Input**
  - Java Source
  - Java source to Java bytecode compiler
  - Java Bytecode

- **Output**
  - Java Virtual Machine
    - JIT

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)

- The compiled code is optimized dynamically at runtime, based on runtime behavior

Anatomy of a Compiler: Why?

- **Input**
  - Source text

- **Output**
  - Executable code

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

Source Language 1

Frontend SL1

Semantic Representation

Backend TL1

Executable target 1

Source Language 2

Frontend SL2

Semantic Representation

Backend TL2

Executable target 2

Source Language 3

Frontend SL3

Semantic Representation

Backend TL3

Executable target 3

SET R1,2
STORE R1, R1
SHIFT R1,1
ADDC R1, R1
STORE R1, R1

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;

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?

Compiler

Source

Programming Languages
Software Engineering

Useful formalisms
- Regular expressions
- Context-free grammars

Data structures
Algorithms

Target

System Architecture
Runtime Environment
Virtual Machines
Garbage Collection
Journey inside a compiler

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

MINUS
MULT
MULT

4
"a"
"b"
"c"

Intermediate Representation
R2 = 4 * a 
R1 = b * b 
R2 = R2 * c 
R1 = R1 – R2

Assembly Code
MOV R2,(sp+8)
SAL R2,2
MOV R1,(sp+16)
MUL R1,(sp+16)
MUL R2,(sp+24)
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 = / oranges`
  - Wrong number of arguments to operator `/`
- `x = func(int a)`
  - A declaration is not expected here

---

**Semantic Errors: Type Checking**

- `x = 4*a*"oranges"`
  - Type mismatch
Runtime Errors

- Division by zero
- Array index out of bounds: 42 > 8

The Real Anatomy of a Compiler

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

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

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

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

```c
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