THEORY OF COMPILATION

LECTURE 01

INTRODUCTION
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 11pt higher grade on average
ENIAC

1946
“The Education of a Computer”
(Grace Hopper)

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

Fig. 4 - SOLUTION OF A PROBLEM
"The Education of a Computer"
(Grace Hopper)

Fig. 5 - Compiling Routines and Subroutines
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

target language

Source text

Executable code

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

"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."
What is a Compiler?

Compiler

Source text

Executable code

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

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

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

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

b = a*2 + 1;

Frontend (analysis)  Semantic Representation  Backend (synthesis)

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

3

EXECUTION ENGINE

7

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

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

Source text

Lexical Analysis
Syntactic Analysis
Semantic Analysis
Intermediate Representation (IR)
Code Generation

Executable code

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

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

Source text

Executable code

Compiler

- Frontend (analysis)
  - Lexical Analysis
  - Syntax Analysis
  - Parsing
- Semantic Representation
- Backend (synthesis)
  - Intermediate Representation (IR)
  - Optimizations
- Code Generation

```c
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?

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

Source → Compiler → Target

- Programming Languages
- Software Engineering
- System Architecture
- Runtime Environment
- Virtual Machines
- Garbage Collection
Course Overview

Compiler

Source text

Lexical Analysis
Syntax Analysis Parsing
Semantic Analysis
IR Optimization
Code Generation

Executable code
Journey inside a compiler

\[ x = b^2 - 4ac \]
Journey inside a compiler

\[ x = b \times b - 4 \times a \times c \]

Token Stream

\(<\text{ID,}"x"\> \ <\text{EQ}> \ <\text{ID,}"b"\> \ <\text{MULT}> \ <\text{ID,}"b"\> \ <\text{MINUS}> \ <\text{INT,}4\> \ <\text{MULT}> \ <\text{ID,}"a"\> \ <\text{MULT}> \ <\text{ID,}"c"\>\]
Journey inside a compiler

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

Syntax Tree

Lexical Analysis
Syntax Analysis
Sem. Analysis
Opt. IR
Code Gen.
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

MULT
  type: int
  loc: sp+16
  ‘b’
mult
  type: int
  loc: R1
  ‘b’
  MULT
  type: int
  loc: sp+16
  4
  MULT
  type: int
  loc: R2
  ‘c’
  MULT
  type: int
  loc: R2
  ‘a’
  type: int
  loc: sp+8

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

Intermediate Representation

\[
\begin{align*}
R2 &= 4 \times a \\
R1 &= b \times b \\
R2 &= R2 \times c \\
R1 &= R1 - R2
\end{align*}
\]

Assembly Code

\[
\begin{align*}
\text{MOV} &\ R2,(sp+8) \\
\text{SAL} &\ R2,2 \\
\text{MOV} &\ R1,(sp+16) \\
\text{MUL} &\ R1,(sp+16) \\
\text{MUL} &\ R2,(sp+24) \\
\text{SUB} &\ R1,R2
\end{align*}
\]
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
- pi = 3oranges
- pi = oranges3

- Illegal token
- Illegal token
- ⟨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

```
MULT
  
MULT
  type: int
  loc: R2
  
  type: int
  loc: sp+8

  type: string
  loc: const

  type: int
  loc: const

  type: int
  loc: const
```

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 → Syntax Analysis → Semantic Analysis → Annotated AST → Intermediate code generation → Intermediate code optimization → Code generation → Target code optimization → Machine code generation → Write executable output → Executable code
The Real Anatomy of a Compiler

Source text

Process text input

Lexical Analysis

Syntax Analysis

Semantic Analysis

Annotated AST

Intermediate code generation

Intermediate code optimization

Code generation

Target code optimization

Machine code generation

Write executable output

Executable code

Back end

Front end
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];
}

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
Next

Lexical Analysis