26 Mar 2025 - Compilers - I - Week 01

→ introduction to compilers

→ Compilers ⇝ heart of CS

→ Compilers are becoming more and more important

→ Varieties of architecture

insanely difficult to code without compilers

→ Programs are written for humans

↓ Compile } efficiency

code

→ Probably : lab exam, final exam
mini-assignments and homeworks.

→ Slides : YNSrikant (NPTEL)

pytorch ⟶ is a compiler infrastructure

Compiler is an excellect example of theory translated into practice

Regular lang. ⟶ regex
CFLs ⟶ prog lang desc.

Parsers for HTML, javascript

ask/remind

Interpreter ~ compilers
here

Last week if possible

* Optimization — snapdragon, nvidia, amd ... → Polyhedral Compilation

↓ sir's area

loop optimization

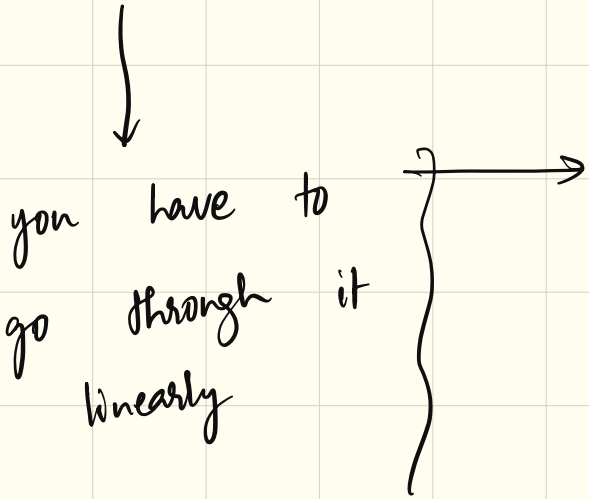
Massive data centres → Massive parallelization

Scheduler algorithm → compilers

LLMs

100 lines of ~ ⇝ 100 GB of MLIR

100GB of code



→ Compilers should be efficient

* Loop optimization →

* Security

e.g.: Deepseek IP address monitoring

Who optimizes compiler ⟶ run compiler on itself | Your optimization optimizes itself

compilers.cse.iith.ac.in

practical application of many things.

incorrect implementation heuristic ⇝ suboptimal soln

as bad ↙ as 50% loss

→ Assembler implementation

→ Grep/awk

Program Analysis Techniques

Parallel programming

→ Open MP
→ CUDA

} → Compilers

Language Processing System

lowering ↓

} Little Biased towards C/C++

pytorch

series of lowering

MLIR ← beautiful to see

LLVM

character stream

Lexical Analyzer

token stream

Syntax Analyzer

correct prog

Compiler

compile all the way to assembly

wrong prog

Verilog/Hardware

1st error

n

All errors don't stop at 1st

→ don't compile

→ understandable error message

Interpreters provide better error messages (input available)

→ very slow ∴ not used in system code

→ much more memory

Compilers → machine code

interpreter → intermediate code

↘ runs on VM

↓ runs on

M

Java

↙

both compiled and interpreted

→ generated byte code

27 Mar 2025

→ Many representations of a program

→ Slide: compiles overview

→ How do you check similarity of 2 programs?

→ what representation do you choose?

input stream { fahrenheit = cent * 1.8 + 32

f a h . . . ␣ = ␣ . . .

fahrenheit → variable name

= → assignment operator, arity = 2

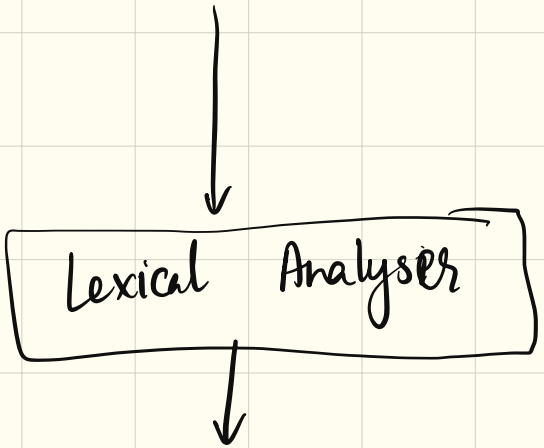
cent → id

* → multiplication, arity = 2

1.8 → fp

32 → integer

(dereference, arity = 1)

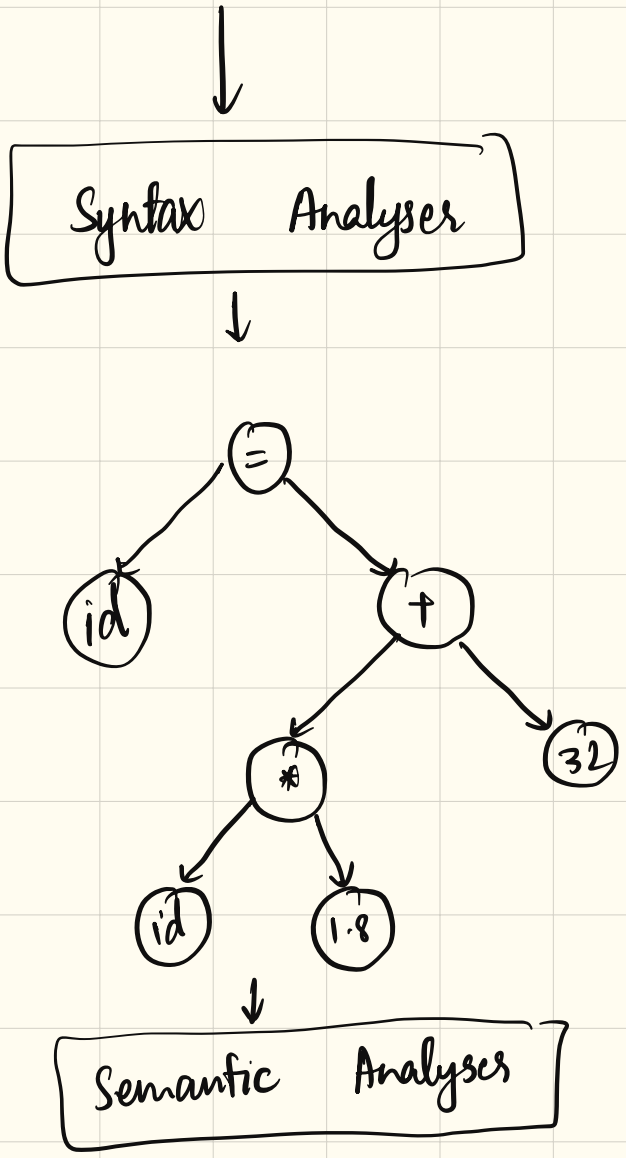


Abstraction (information of variable names is not thrown out till a stage)

<id, 1> <assign> <id, 2> <multop>

Variable names are irrelevant to some extent

<id , 1> - - -



Do you write lexical analyser and parser by hand or do you use a HLL?

PDA
DFA
??

there are tools that take grammar as input and output lex. analyser

"code Generator Generator"

e.g. YACC

syntax tree → Semantic Analyser

where compatible meaningful

→ Type coersion and casting etc.

→ Requirements of hardware and struct

multiply struct with array ⊗

Semantic Analysis

→ Bubble sort ≠ Quick sort → Semantics should not change.

→ Warnings , Errors

↓

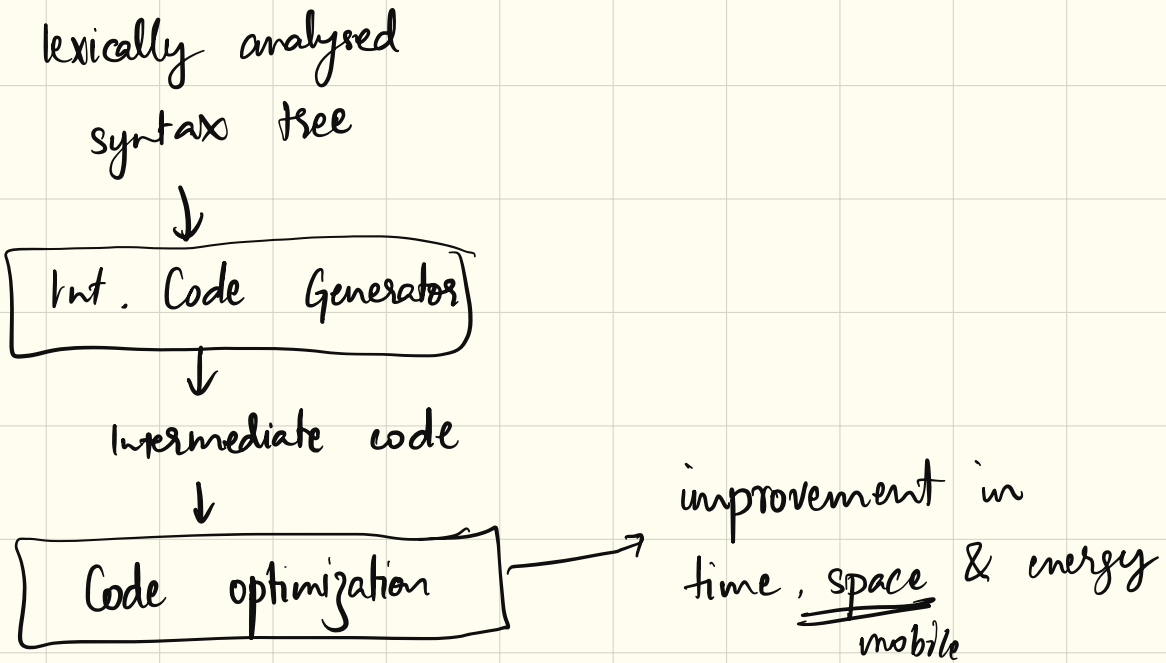
I will continue, but check this part

Type information ——stored in——→ Symbol table

→ big hash map

→ Complete checks are not possible (undecidable)

→ We do trivial checks, but it is very imp.



→ Generating machine code from source code directly

m langs

n targets

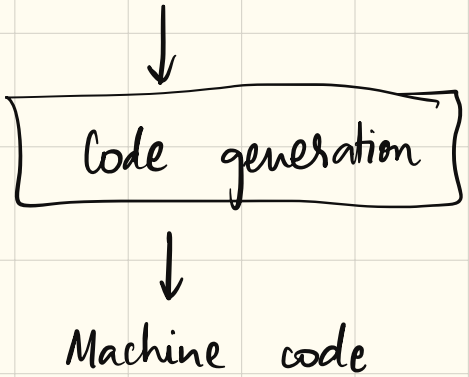
m × n compilers

intermediate:

m + n

→ Types of intermediate representation

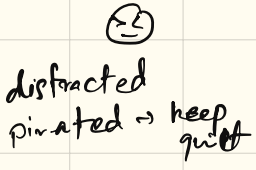
→ Optimized int. code



→ Optimization is undecidable
(not even NP)

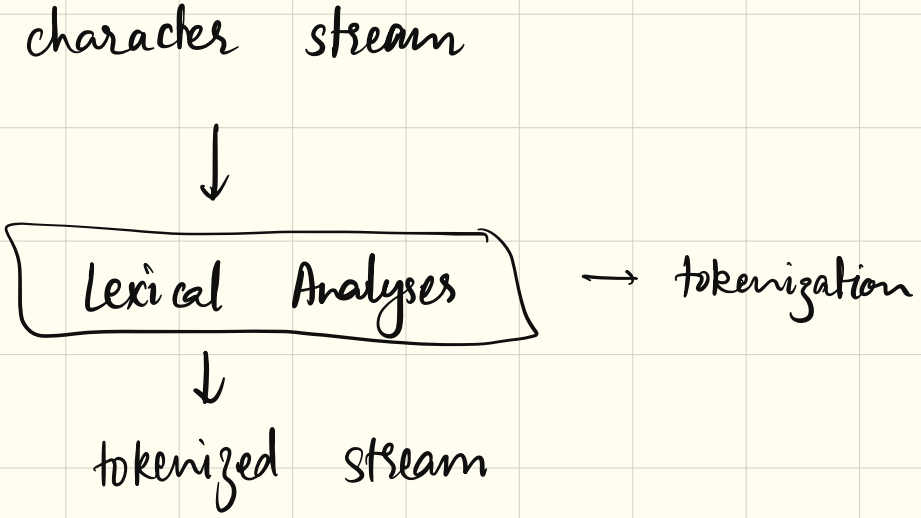
LLVM ⟶ 300 passes of optimization
↳ parallelization

→ examples of Machine - Independent optimizations
→ theorems



Lexical Analysis - Part 1

→ Outlines



Ganaka
?

What is lexical analysis?

→ slides

→ Preprocessor ⟶ ensures that the program is ready for tokenization

simple-hello.c ⟶ 25 lines

gcc -E simple-loop > simple-loop-pre.c

↓

includes

text - replacement N

preprocessed code

\ \n } → escape newline

include <stdio.h> ⟶ only declaration

Compiler does not worry about implementation } → linker handles it