

Compiler Design

Ammar Daşkın, Fall 2025

This week content

❖ Administrivia

❖ Introduction

- Why do we need to learn compiler design?
- What's a compiler?
- The structure of a typical compiler
- LLVM and modular compiler
- Other compiler related topics

❖ Compiler implementation approaches for project

Prerequisites

I will assume you have taken courses:

C programming,

Object oriented programming (any language)

Data Structures and Algorithms

Algorithm Design & Analysis

Computer Architecture

Operating Systems

Textbook and course material

★ No required textbook.

Weekly lecture slides are posted on classroom.

Slides are mostly based on

- <https://web.stanford.edu/class/cs143/>

other resources used in the slides:

- Introduction to Compilers and Language Design, Douglas Thain, 2nd edition, 2020. Free online textbook: <https://www3.nd.edu/~dthain/compilerbook/>
- <https://web.stanford.edu/class/cs143/>
- **Dragon Book: Compilers: Principles, Techniques, & Tools, Aho, Lam, Sethi & Ullman, A-W.**
- <https://courses.cs.washington.edu/courses/cse401/22au/>
- Engineering a Compiler, Cooper & Torczon, 3rd edition.
- AI tools DeepSeek, Copilot etc.

What is this course about?

Course has theoretical and practical aspects

- First part is more on automata theory, formal languages, regular expressions
- Second part compiler design

Some similarities to programming language and automata theory courses

Grading policy

- 20% midterm
 - mostly based on written assignments
- 40% final exam
- 10% written assignments
- 30% programming projects
 -

Hw and projects

Written assignments + exams = theory

- 2 or 3 theoretical assignment
 - a. **Everybody is alone!**
 - b. you can use AI

Programming assignments = practice

- Assigned via github and classroom
- ~3 or 4 programming projects
 - a. Can work in group of 2s or 3s
-
- Submissions through `classroom.google.com` & `github.com`
- No late submission

Programming assignments

One project in 3 or 4 parts

- A compiler project in 3 or 4 steps
 - You will design syntax rules for a programming language
 - Then write a compiler
 - You can use any language for coding
 - C/C++, Python, Java, Rust
 - Haskell, OCaml, Scala etc.
- You can use any AI tools

E.g. A mini compiler in C

Where you write a compiler in C to compile a programming language (e.g. Java).

Academic Integrity

Don't use work from uncited sources

- If you benefit from some work of others, list them as references (online references or books)

Any kind of plagiarism and cheating are prohibited (Please, refer to the university cheating policy).

Discussing the assignments or projects with your friends is allowed; but, all the submitted work should be yours alone. List your collaborators (if you discuss your homework with your friends) in your assignments.

Introduction

- Importance of learning compilers and course goals
- Typical compiler toolchain
- Modern compiler design tools

How do we execute something like this?

```
#include <stdio.h>
#define X 10

int main(){
    int a = X;

    printf("hello world!\n a = %d", a);

    return 0;
}
```

How to tell a computer to carry out a computation written as text in a file?

Course goal

Open the lid of compilers and see inside

- Understand what they do
- Understand how they work
- Understand how to build them

Correctness over performance

- Correctness is essential in compilers
- They must produce correct code

Why do we need to learn?

You can understand/compare high level languages better!

→ You can write more efficient programs

You can design similar translating tools

You learn concepts parsing, regular expressions, abstract syntax tree that are also used **in different areas.**

Compiler

a software that

- translates a program in a source language to a program in a target language (generally a low level language).
 - also improves them in this translation.

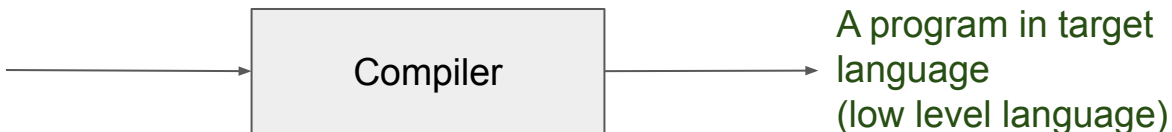
→ **Cross Compiler**

- ◆ runs on a machine 'A' and produces a code for another machine 'B'.

→ **Source-to-source Compiler**

- ◆ or transcompiler or transpiler translates source code into the source code of another programming language.

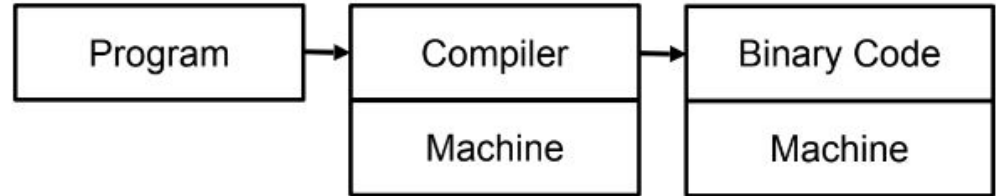
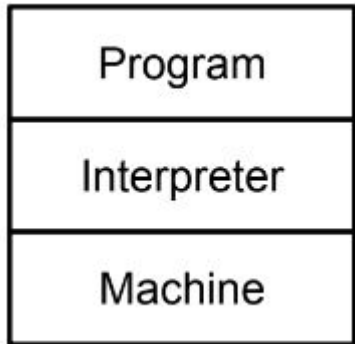
A program in
source
language



How are language are implemented?

Two major strategies:

- Interpreters run your program
 - read in a program and then executes it directly, without a translation file.
 - Python, Ruby
 - Sometimes Virtual machine
- Compilers translate your program



Language implementations and a little history

- Compilers dominate low-level languages
 - C, C++, Go, Rust
- Interpreters dominate high-level languages
 - Python, Ruby
- Some language implementations provide both
 - Java, Javascript, WebAssembly
 - Interpreter + Just in Time (JIT) compiler

```
>>> import dis # "dis" - Disassembler of Python bytecode into mnemonics.
>>> dis.dis('print("Hello, World!")')
1          0 LOAD_NAME               0 (print)
           2 LOAD CONST              0 ('Hello, World!')
           4 CALL FUNCTION            1
           6 RETURN_VALUE
```


A Typical Compiler Toolchain

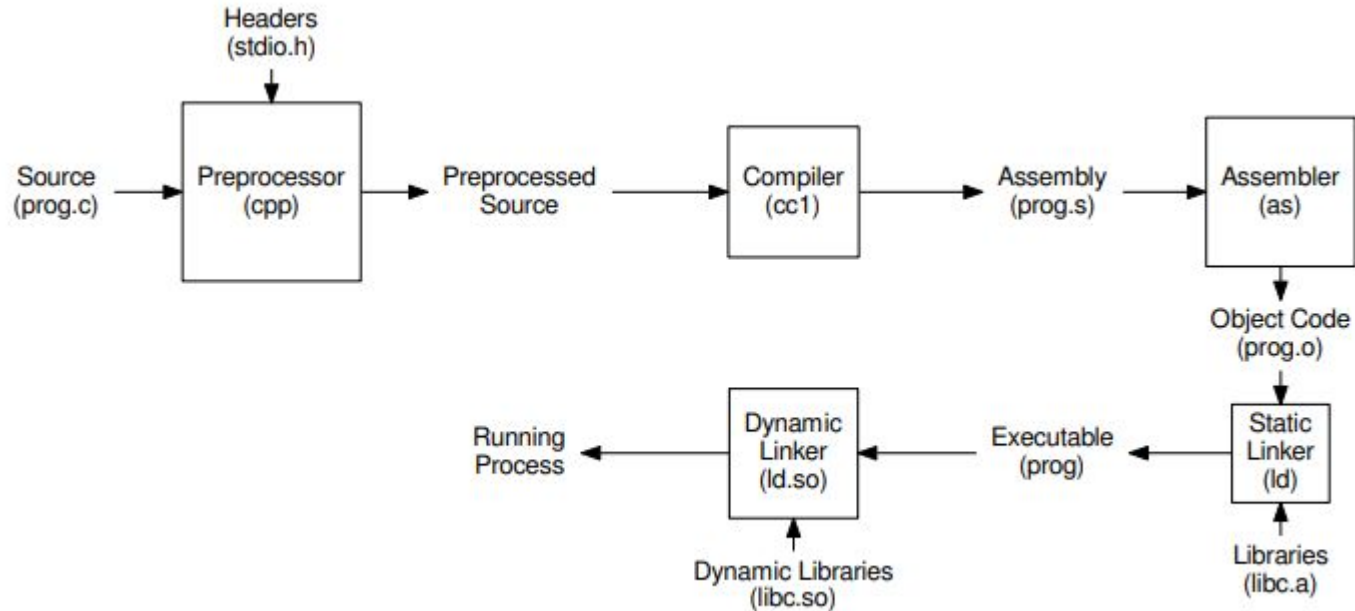


Fig2.1 from the book <https://www3.nd.edu/~dthain/compilerbook/chapter2.pdf>

The Stages of a Unix Compiler

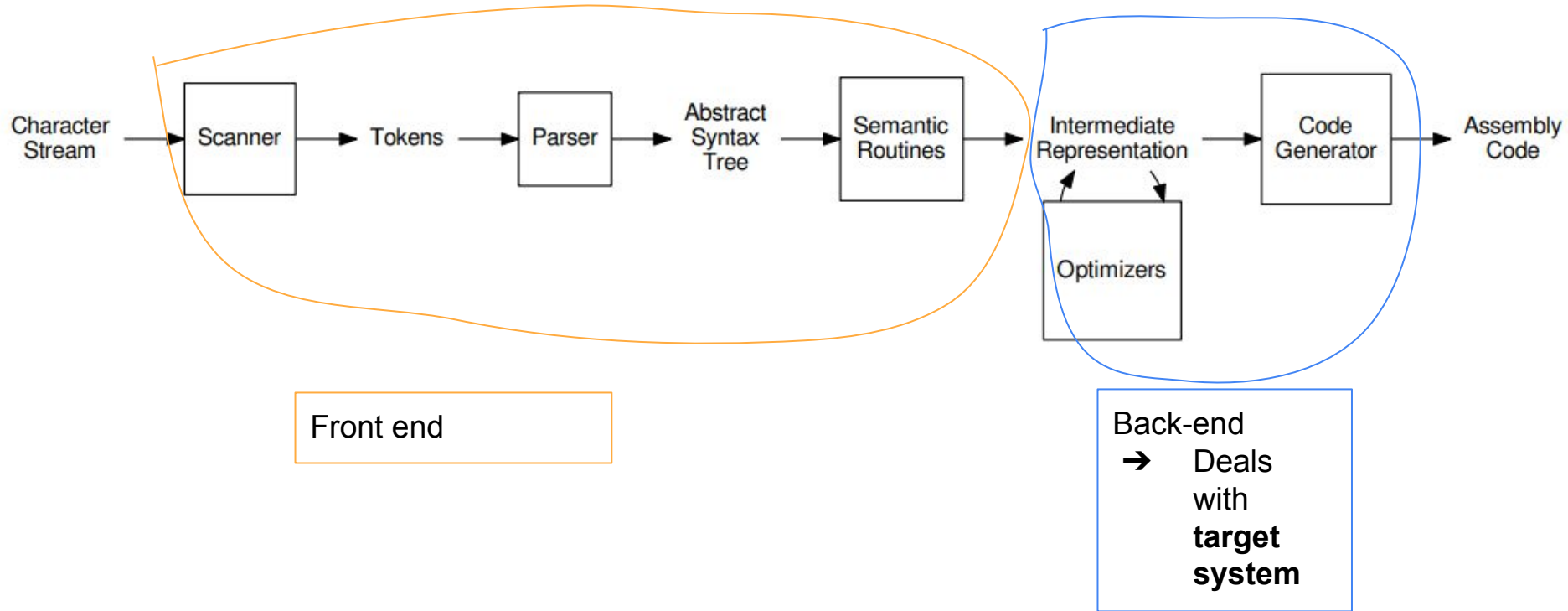


Fig2.2 with some modifications from the book <https://www3.nd.edu/~dthain/compilerbook/chapter2.pdf>

by analogy, similar to how humans comprehend English.

1. Lexical Analysis

→ identify words

2. Parsing

→ identify sentences

3. Semantic Analysis

→ analyse sentences

4. Optimization

→ editing

5. Code Generation

→ translation

Lexical analysis

First step: recognize words.

- Smallest unit above letters

This is a sentence.

Lexical analysis is not trivial

Consider:

ist his ase nte nce.

Lexical analyzer divides program text into “**words**” or “**tokens**”

→ the individual characters are grouped together to form complete tokens

If x == y then z = 1; else z = 2;

- Units:

Parser

Groups tokens into complete statements and expressions,

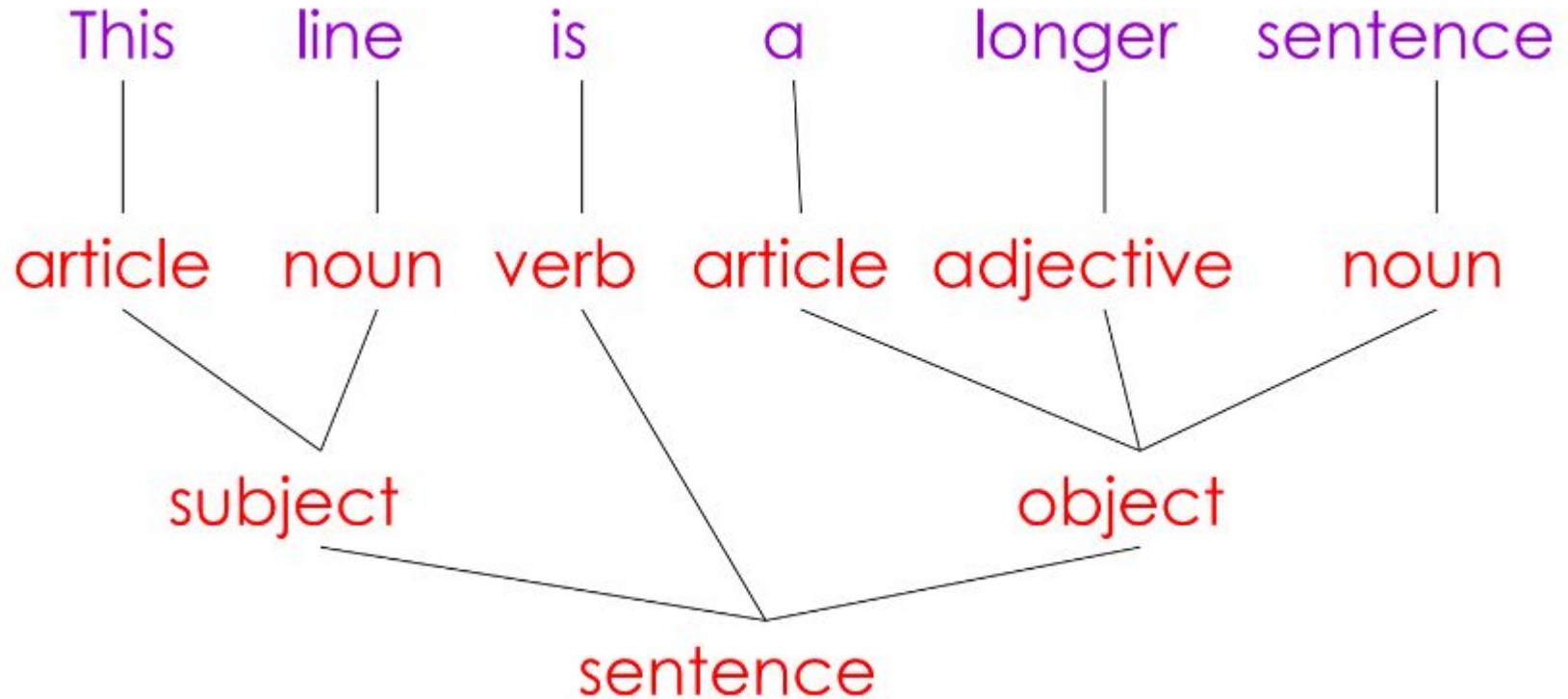
→ much like words are grouped into sentences in a natural language

Parsing = Diagramming Sentences

– The diagram is a tree

An **abstract syntax tree (AST)** captures the grammatical structures of the program.

Diagramming a Sentence

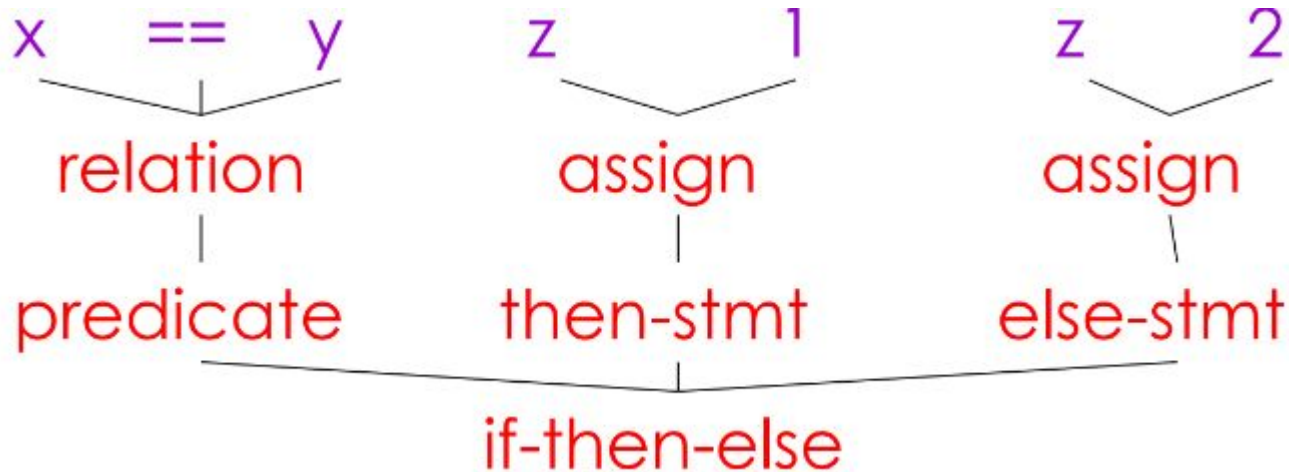


Parsing Programs

If x == y then z = 1; else z = 2;

Parsing program expression is the same

Diagrammed



Semantic Analysis

Once sentence structure is understood, we can try to understand “meaning”

– **But meaning is too hard for compilers**

Compilers perform limited semantic analysis to catch inconsistencies

Semantic Analysis in English

Example:

Jack said Jerry left his assignment at home.

What does “his” refer to? Jack or Jerry?

Even worse:

Jack said Jack left his assignment at home?


How many Jacks are there?

Which one left the assignment?

Semantic Analysis in Programming

Programming languages define strict rules **to avoid such ambiguities**

```
{  
    int Jack = 3;  
    {  
        int Jack = 4;  
        printf("%d\n", Jack);  
    }  
}
```



Prints 4, the inner
definition is used

More semantic analysis

Compilers perform many semantic checks besides variable bindings

- Example:

Jack left her homework at home.

- If Jack is male
 - Possible type mismatch between her and Jack

Optimization

- Akin to editing
 - Minimize reading time
 - Minimize items the reader must keep in short-term memory
- Automatically modify programs so that they
 - Run faster
 - Use less memory
 - In general, to use or conserve some resource
- The project has no optimization component

Optimization Example

$X = Y * 0$ is the same as $X = 0$

(the $*$ operator is annihilated by zero)

Is this optimization legal?

Code generation

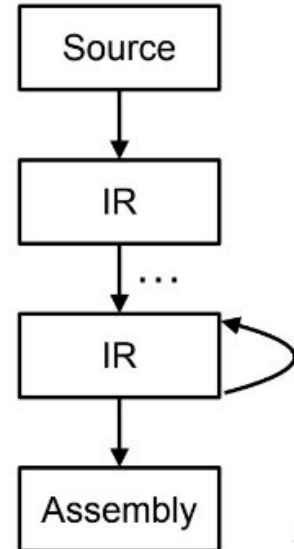
- Typically produces assembly code
- Generally a translation into another language
 - Analogous to human translation

Intermediate representations (IRs)

- Many compilers perform translations between successive intermediate languages
- All but first and last are intermediate representations (IR) internal to the compiler

IRs are generally ordered in descending level of abstraction

- Highest is source
- Lowest is assembly



IRs are useful because lower levels expose features hidden by higher levels

- registers
- memory layout
- raw pointers
- etc.

• But lower levels obscure high-level meaning

- Classes
- Higher-order functions
- Even loops...

Issues

Compiling is almost this simple, but there are many pitfalls

- Example: How to handle erroneous programs?
- Language design has big impact on compiler
 - Determines what is easy and hard to compile
 - Course theme: many trade-offs in language design

Compilers today

The overall structure of almost every compiler adheres to our outline

- The proportions have changed since FORTRAN
 - Early: lexing and parsing most complex/expensive
 - Today: optimization dominates all other phases, lexing and parsing are well understood and cheap
- Compilers are now also found inside libraries

An example steps of a compiler

```
height = (width+56) * factor(foo);
```

Lexical analyzer(the scanner) generates tokens

id:height	=	(id:width	+	int:56)	*	id:factor	(id:foo)	;
------------------	----------	----------	-----------------	----------	---------------	----------	----------	------------------	----------	---------------	----------	----------

An example steps of a compiler

id:height	=	(id:width	+	int:56)	*	id:factor	(id:foo)	;
-----------	---	---	----------	---	--------	---	---	-----------	---	--------	---	---

The next step is to determine whether this sequence of tokens forms a valid program. The parser does this by looking for patterns that match the grammar of a language.

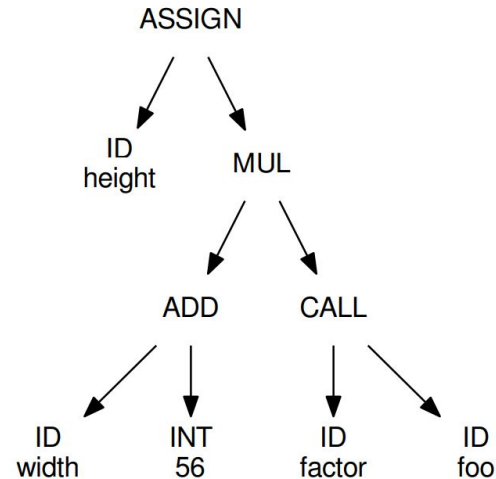
Grammar G_1

- | |
|--|
| <ol style="list-style-type: none">1. $\text{expr} \rightarrow \text{expr} + \text{expr}$2. $\text{expr} \rightarrow \text{expr} * \text{expr}$3. $\text{expr} \rightarrow \text{expr} = \text{expr}$4. $\text{expr} \rightarrow \text{id} (\text{expr})$5. $\text{expr} \rightarrow (\text{expr})$6. $\text{expr} \rightarrow \text{id}$7. $\text{expr} \rightarrow \text{int}$ |
|--|

id:height = (id:width + int:56) * id:factor (id:foo) ;

Grammar G_1

1. $\text{expr} \rightarrow \text{expr} + \text{expr}$
2. $\text{expr} \rightarrow \text{expr} * \text{expr}$
3. $\text{expr} \rightarrow \text{expr} = \text{expr}$
4. $\text{expr} \rightarrow \text{id} (\text{expr})$
5. $\text{expr} \rightarrow (\text{expr})$
6. $\text{expr} \rightarrow \text{id}$
7. $\text{expr} \rightarrow \text{int}$



The parser looks for sequences of tokens that can be replaced by the left side of a rule in our grammar.

Each time a rule is applied, the parser creates a node in a tree, and connects the sub-expressions into the **abstract syntax tree (AST)**

id:height

=

(

id:width

+

int:56

)

*

id:factor

(

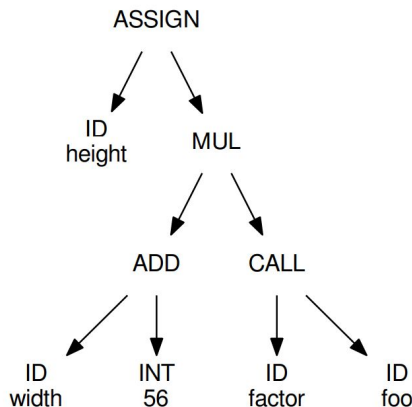
id:foo

)

;

Grammar G_1

1. $\text{expr} \rightarrow \text{expr} + \text{expr}$
2. $\text{expr} \rightarrow \text{expr} * \text{expr}$
3. $\text{expr} \rightarrow \text{expr} = \text{expr}$
4. $\text{expr} \rightarrow \text{id} (\text{expr})$
5. $\text{expr} \rightarrow (\text{expr})$
6. $\text{expr} \rightarrow \text{id}$
7. $\text{expr} \rightarrow \text{int}$



```
LOAD $56      -> r1
LOAD width    -> r2
IADD r1, r2   -> r3
ARG foo
CALL factor   -> r4
IMUL r3, r4   -> r5
STOR r5       -> height
```

The semantic routines traverse the AST and derive additional meaning by relating parts of the program to each other, and to the definition of the programming language.

a post-order traversal of the AST generates an IR instruction for each node in the tree.
A typical IR looks like an abstract assembly language.

X86 assembly code that is one possible translation of the IR.

```
MOVQ    width, %rax      # load width into rax
ADDQ    $56, %rax        # add 56 to rax
MOVQ    %rax, -8(%rbp)    # save sum in temporary
MOVQ    foo, %edi         # load foo into arg 0 register
CALL    factor            # invoke factor, result in rax
MOVQ    -8(%rbp), %rbx    # load sum into rbx
IMULQ   %rbx              # multiply rbx by rax
MOVQ    %rax, height     # store result into height
```

- The intermediate representation is where most forms of optimization occur.
- Dead code is removed, common operations are combined, and code is generally simplified to consume fewer resources and run more quickly.
- Finally, the intermediate code must be converted to the desired assembly code.
- Note that the assembly instructions do not necessarily correspond one-to-one with IR instructions.

Internals of GNU compilers

<https://gcc.gnu.org/onlinedocs/gccint/index.html#Top>

-fdump-tree-all

<https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html#Option-Summary>

<https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html#Developer-Options>

Modern Compiler Architecture: The LLVM Project

LLVM: The Low Level Virtual Machine

- **What is LLVM?** An open-source compiler infrastructure project, designed as a reusable and modular set of compiler and toolchain technologies.
- **Core Idea:** A language-agnostic **Intermediate Representation (IR)** that sits at the heart of the compilation process.
- **Key Benefit:** Separates the **front-end** (language-specific parsing, AST, semantic analysis) from the **back-end** (machine-specific code generation and optimization).

The LLVM Compiler Pipeline

1. **Frontend (e.g., Clang for C/C++)**: Translates source code into LLVM IR.
2. **Middle-end (LLVM Optimizer)**: Performs target-independent optimizations on the LLVM IR. This is where most optimizations happen (`-O1`, `-O2`, `-O3`).
3. **Backend (LLVM Code Generator)**: Converts the optimized LLVM IR into native machine code (x86, ARM, etc.).

Why is LLVM a Big Deal?

- **Reusability:** You can create a new programming language by *just* writing a frontend that generates LLVM IR. LLVM handles the difficult task of optimization and code generation for multiple architectures.
- **Performance:** LLVM's optimizations are highly effective and continuously improved by a large community.
- **Industry Standard:** Used by Apple (for Swift & Clang), Google (Android NDK, ML compilers), Rust, and many others.

Functional Programming Concepts in Compiler Design

Many modern compiler techniques are inspired by, or naturally implemented in, functional programming languages.

Key FP Concepts in Compiler Phases:

- **Immutability & Pure Functions:**
 - Compiler passes (like optimizations) are often *pure functions* that take an AST/IR and return a new, transformed AST/IR.
 - This makes passes easier to reason about, test, and debug, as they have no hidden side effects.
- **Algebraic Data Types (ADTs) & Pattern Matching:**
 - **Perfect for defining the AST.** An `Expression` node can be a `BinaryExpr(operator, left, right)`, a `Number(value)`, or a `Variable(name)`.
 - Pattern matching allows for elegant and concise traversal and transformation of these complex tree structures.
- **Higher-Order Functions:**
 - Used extensively for traversing data structures (e.g., `map`, `fold` over a list of statements in a function body).

Languages Used in Modern Compiler Construction

- **OCaml & Haskell:** Traditionally used for research and production compilers (e.g., the original F# compiler, the Glasgow Haskell Compiler).
- **Rust:** Its powerful pattern matching and focus on safety make it an excellent modern choice for writing compilers.
- **Scala:** Blends OOP and FP, used in the Dotty/Scala 3 compiler

Beyond the “Typical” Compiler

The “typical” compiler structure is common, but not the only way to build a language implementation.

- **Single-Pass Compilers:**

- Generate code immediately during parsing, without building a full AST.
- **Use Case:** Simpler languages or environments with severe memory constraints (e.g., early Pascal compilers).

- **Just-in-Time (JIT) Compilation:**

- Compilation happens at **runtime**, right before the code is executed.
- **Advantage:** Can perform optimizations based on actual runtime data (e.g., profile-guided optimization).
- **Examples:** Java’s HotSpot JVM, JavaScript’s V8 engine, .NET’s CLR.

- **Interpreters & Virtual Machines (VMs):**

- **Tree-Walk Interpreters:** Directly execute the AST. Simple to implement but slow.
- **Bytecode Interpreters:** Compile source to a compact bytecode, which is then executed by a virtual machine. Offers a good balance of performance and portability (e.g., Python, Java JVM).

- **Transpilers (Source-to-Source Compilers):**

- Translate from a high-level language to another high-level language (e.g., TypeScript to JavaScript, Kotlin to Java Bytecode).
- Allows developers to use new language features while targeting a stable, well-supported platform.

Compiler Implementation Approaches for Project

- **Path A: Traditional C with Flex/Bison**
- **Path B: Python with PLY**
- **Path C: Modern OCaml with LLVM**

Compiler Implementation Approaches

Choose Your Own Adventure: Project Paths

In this course, you can implement your MiniLang compiler using different technologies:

- **Path A: Traditional C with Flex/Bison**
 - The classic approach - understand everything from scratch
- **Path B: Python with PLY**
 - Rapid prototyping - focus on concepts over implementation details
- **Path C: Modern OCaml with LLVM**
 - Industry-inspired - leverage modern tools and functional programming

All paths implement the same language, but with different tools and trade-offs.

Path A: Traditional C with Flex/Bison

The Classic Compiler Stack

Source Code → [Flex] → Tokens → [Bison] → AST → [Your C Code] → Assembly

What You'll Work With:

- **Flex:** Generates lexical analyzer from regex rules
- **Bison:** Generates parser from grammar rules
- **Manual C Code:** Your semantic analysis, code generation
- **Output:** x86 Assembly or similar

Example Code Snippet

```
// Manual AST node definition

struct ASTNode {

    enum { NUMBER, BINOP, VARIABLE } type;

    union {

        int value;

        struct { char op; struct ASTNode *left, *right;
        } binop;

        char* varname;

    };

};
```

```
// Manual code generation

void generate_asm(struct ASTNode* node) {

    if (node->type == NUMBER) {

        printf("    mov rax, %d\n", node->value);

    }

    // ... more manual assembly

}
```

Path B: Python with PLY

Rapid Prototyping Approach

Source Code → [PLY Lex] → Tokens → [PLY Yacc] → AST → [Your Python] → Bytecode/IR

What You'll Work With:

- **PLY (Python Lex-Yacc):** Pure Python lexing/parsing
- Python Classes: For AST, symbol tables, semantic analysis
- Python Eval: Or simple stack-based bytecode generation
- Output: Python bytecode, custom VM, or simple assembly

Example Code Snippet

```
# Elegant AST definition with classes
```

```
class BinOp:
```

```
    def __init__(self, op, left, right):
```

```
        self.op = op
```

```
        self.left = left
```

```
        self.right = right
```

```
# Simple code generation
```

```
def generate_code(node):
```

```
    if isinstance(node, BinOp):
```

```
        generate_code(node.left)
```

```
        generate_code(node.right)
```

```
        print(f" {op_to_asm[node.op]}") # e.g., "add"
```

```
# Built-in data structures make symbol tables easy
```

```
symbol_table = {}
```

Best For: Quick implementation, focus on algorithms over systems details.

Path C: Modern OCaml with LLVM

Industry-Strength Functional Approach

Source Code → [OCaml Lex] → Tokens → [OCaml Parse] → AST → [LLVM IR Gen] → [LLVM]
→ Native Code

What You'll Work With:

- OCaml Lex/Yacc: Or parser combinators (Menhir)
- Algebraic Data Types: Perfect for AST representation
- LLVM Bindings: Professional optimization and code generation
- Pattern Matching: Elegant AST traversal and transformation

Example Code Snippet

```
(* Clean AST definition *)
```

```
type expr =
```

```
| Number of int
```

```
| BinOp of string * expr * expr
```

```
| Variable of string
```

```
(* Pattern matching for code generation *)
```

```
let rec codegen expr builder =
```

```
  match expr with
```

```
  | Number n -> Llvm.const_int i32_type n
```

```
  | BinOp ("+", lhs, rhs) ->
```

```
      let l = codegen lhs builder in
```

```
      let r = codegen rhs builder in
```

```
      Llvm.build_add l r "addtmp" builder
```

Best For: Experience with modern compiler architecture and functional programming.

Project Expectations by Path

Same Language, Different Implementation Effort

Common Requirements (All Paths):

- Implement MiniLang with variables, functions, arithmetic
- Handle lexical errors, syntax errors, type errors
- Produce executable output
- Write tests and documentation

How to Choose Your Path

Questions to Help You Decide:

1. What's your programming background?
 - Strong C experience? → Consider C path
 - Python comfort? → Python path
 - Want functional programming challenge? → OCaml path
2. What are your learning goals?
 - Understand computer architecture? → C path
 - Focus on compiler theory? → Python path
 - Learn industry tools? → OCaml path
3. What's your risk tolerance?
 - Safe choice: Python (easier debugging)
 - Ambitious: OCaml (modern approach)
 - Traditional: C (proven method)

All paths will teach you compiler design!

The concepts are the same; only the implementation differs.

You can switch paths early if needed!

Getting Started with Each Path

Resources for Each Approach:

C/Flex/Bison:

- GNU Flex Manual: `info flex`
- GNU Bison Manual: `info bison`
- Example: "A Compact Guide to Lex & Yacc"
- Starter code: Simple calculator example

Python/PLY:

- PLY Documentation:
 - a. [PLY \(Python Lex-Yacc\)](#)
 - b. <https://ply.readthedocs.io/en/latest/>
- Example: PLY calculator example
- Starter code: Pre-built AST classes

OCaml/LLVM:

- OCaml LLVM Bindings:
 - a. <https://llvm.org/>
 - b. <https://llvm.moe/>
- "OCaml Programming" guide
- Example: Kaleidoscope compiler tutorial
 - a. [1. Kaleidoscope: Tutorial Introduction and the Lexer – LLVM 12 documentation](#)
- Starter code: LLVM setup instructions

Next Steps:

1. Review the example code for each path
2. Setup development environment
3. Start with lexical analysis

Next

Review of formal grammars

Lexical analysis – scanning & regular expressions