LASA {CS}

So you want to build a compiler?

Independent Study 2024-25
Rachit Kakkar
4/18

Table of Contents

Introduction

1. What is a compiler?
2. Compilers vs. Interpreters

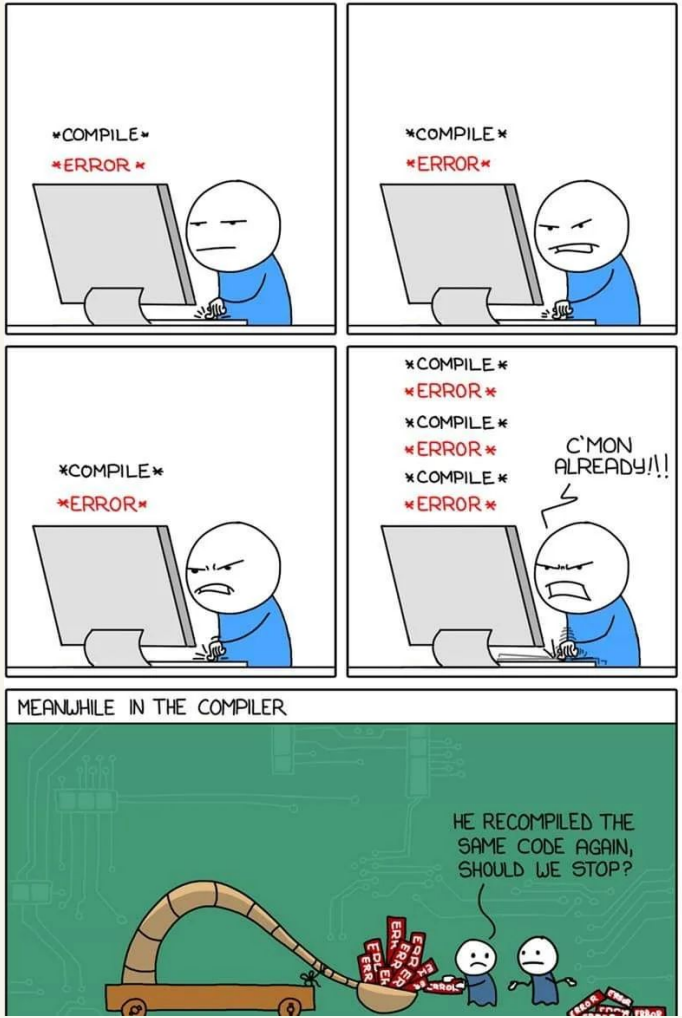
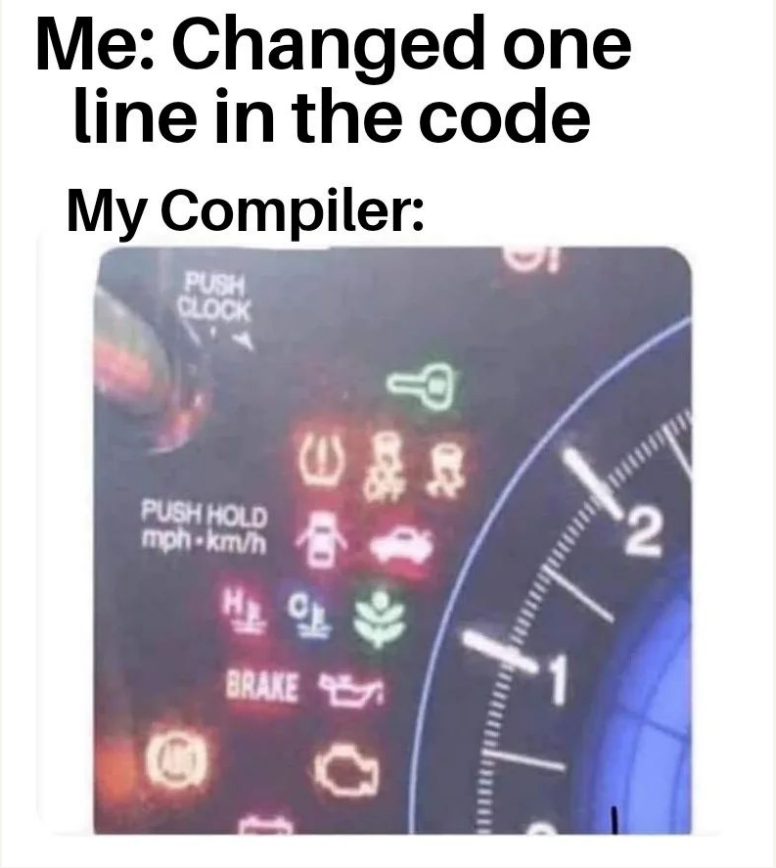


Table of Contents

Building A Compiler

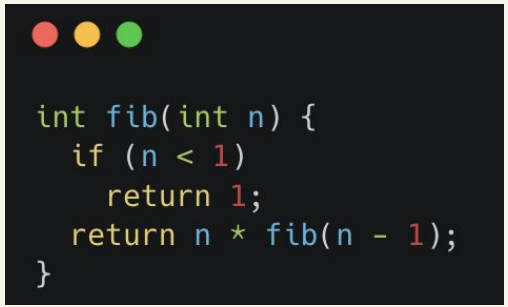
4. Language Reference
5. How a compiler works!
6. LLVM Introduction
7. Demo



What IS A Compiler?

What IS A Compiler?

Source



```
int fib(int n) {
  if (n < 1)
    return 1;
  return n * fib(n - 1);
}
```

Human readable language
(e.g. *C, C++, Java*)

What IS A Compiler?

Source

```c
int fib(int n) {
  if (n < 1)
    return 1;
  return n * fib(n - 1);
}
```

Human readable language
(e.g. *C, C++, Java*)

Compilation Target

```
fib(int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 0
        jg      .L2
        mov     eax, 1
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        sub     eax, 1
        mov     edi, eax
        call    fib(int)
        imul    eax, DWORD PTR [rbp-4]
.L3:
        leave
        ret
```

Machine code (e.g. *x86, ARM, JVM bytecode*)

What IS A Compiler?

Source

```c
int fib(int n) {
  if (n < 1)
    return 1;
  return n * fib(n - 1);
}
```

Human readable language
(e.g. *C, C++, Java*)

`JUMP 0x1000`

Assembler

Compilation Target

```
fib(int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 0
        jg      .L2
        mov     eax, 1
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        sub     eax, 1
        mov     edi, eax
        call    fib(int)
        imul    eax, DWORD PTR [rbp-4]
.L3:
        leave
        ret
```

Machine code (e.g. *x86, ARM, JVM bytecode*)

What IS A Compiler?

Source

```c
int fib(int n) {
  if (n < 1)
    return 1;
  return n * fib(n - 1);
}
```

Human readable language
(e.g. *C, C++, Java*)

Assembler

```
JUMP 0x1000

000010 00000 00000 00000 10000 000000
```

Compilation Target

```
fib(int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 0
        jg      .L2
        mov     eax, 1
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        sub     eax, 1
        mov     edi, eax
        call    fib(int)
        imul    eax, DWORD PTR [rbp-4]
.L3:
        leave
        ret
```

Machine code (e.g. *x86, ARM, JVM bytecode*)

What IS A Compiler?

Source

```c
int fib(int n) {
  if (n < 1)
    return 1;
  return n * fib(n - 1);
}
```

Human readable language
(e.g. *C, C++, Java*)

Assembler

```
JUMP 0x1000
000010 00000 00000 00000 10000 000000
```

Compilation Target

```
fib(int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 0
        jg      .L2
        mov     eax, 1
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        sub     eax, 1
        mov     edi, eax
        call    fib(int)
        imul    eax, DWORD PTR [rbp-4]
.L3:
        leave
        ret
```

Machine code (e.g. *x86, ARM, JVM bytecode*)

What IS A Compiler?

Source

```c
int fib(int n) {
  if (n < 1)
    return 1;
  return n * fib(n - 1);
}
```

Human readable language
(e.g. *C, C++, Java*)

Assembler

```
JUMP 0x1000
000010 00000 00000 00000 10000 000000
```

Compilation Target

```
fib(int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 0
        jg      .L2
        mov     eax, 1
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        sub     eax, 1
        mov     edi, eax
        call    fib(int)
        imul    eax, DWORD PTR [rbp-4]
.L3:
        leave
        ret
```

Machine code (e.g. *x86, ARM, JVM bytecode*)

What IS A Compiler?

Source

```c
int fib(int n) {
  if (n < 1)
    return 1;
  return n * fib(n - 1);
}
```

Human readable language
(e.g. *C, C++, Java*)

COMPILER

Assembler

```
JUMP 0x1000
000010 00000 00000 00000 10000 000000
```

Compilation Target

```
fib(int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 0
        jg      .L2
        mov     eax, 1
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        sub     eax, 1
        mov     edi, eax
        call    fib(int)
        imul    eax, DWORD PTR [rbp-4]
.L3:
        leave
        ret
```

Machine code (e.g. *x86, ARM, JVM bytecode*)

Compilers VS Interpreters

Compilers VS Interpreters

Compiler

A compiler translates the whole program at once*

*more or less

Compilers VS Interpreters

Compiler

A compiler translates the whole program at once*

*more or less

Interpreter

An interpreter “interprets” a program line-by-line*

*more or less





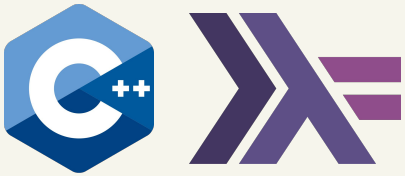


Compilers VS Interpreters

Compiler

A compiler translates the whole program at once*

*more or less



Interpreter

An interpreter “interprets” a program line-by-line*

*more or less

JIT Compilers & Bytecode

Just-in-time (JIT) compilation involves compiling code on the fly into a faster form (typically machine code)

Some languages compile to and then interpret a “bytecode” first and then execute that instead

JIT compilation is also often used with bytecode

Modern Languages are more complex....

Code

```
def fib(n):
  if (n < 1):
    return 1
  return n * fib(n-1)
```

Modern Languages are more complex....

Code

```
def fib(n):
    if (n < 1):
        return 1
    return n * fib(n-1)
```

Compiler

Bytecode

```
 0 RESUME                   0

 2 LOAD_FAST                0 (n)
 4 LOAD_CONST               1 (1)
 6 COMPARE_OP               0 (<)
12 POP_JUMP_FORWARD_IF_FALSE     2 (to 18)

14 LOAD_CONST               1 (1)
16 RETURN_VALUE

18 LOAD_FAST                0 (n)
20 LOAD_GLOBAL              1 (NULL + fib)
32 LOAD_FAST                0 (n)
34 LOAD_CONST               1 (1)
36 BINARY_OP               10 (-)
40 PRECALL                  1
44 CALL                     1
54 BINARY_OP                5 (*)
58 RETURN_VALUE
```

Modern Languages are more complex....

Code

```
def fib(n):
    if (n < 1):
        return 1
    return n * fib(n-1)
```

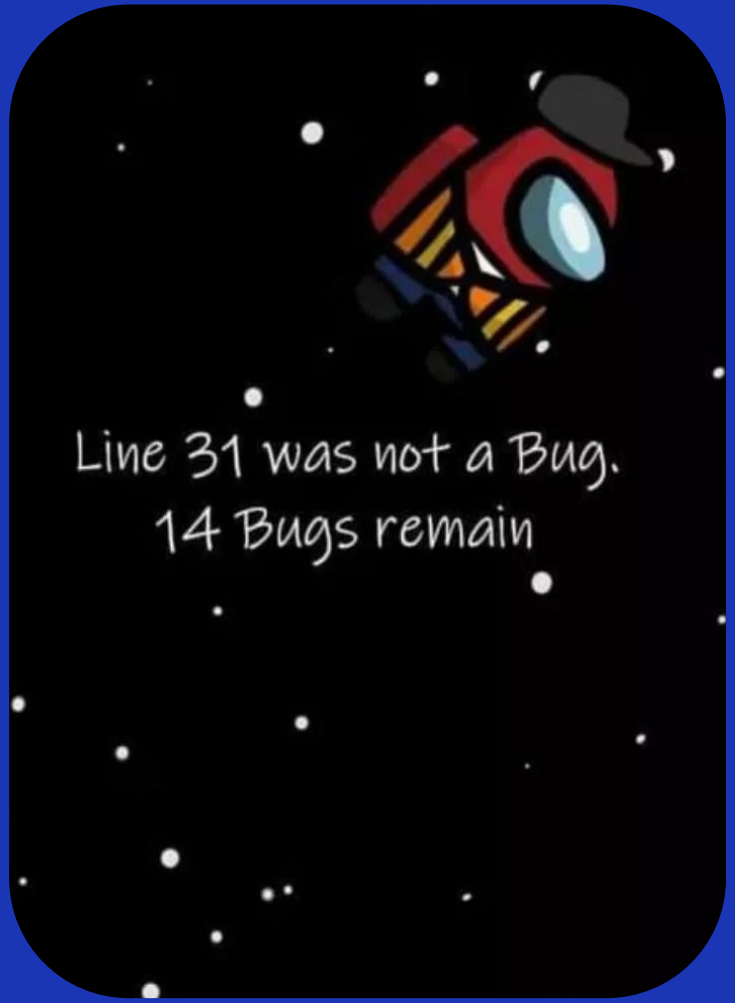
Compiler

Bytecode

```
 0 RESUME                   0

 2 LOAD_FAST                0 (n)
 4 LOAD_CONST               1 (1)
 6 COMPARE_OP               0 (<)
12 POP_JUMP_FORWARD_IF_FALSE     2 (to 18)

14 LOAD_CONST               1 (1)
16 RETURN_VALUE

18 LOAD_FAST                0 (n)
20 LOAD_GLOBAL              1 (NULL + fib)
32 LOAD_FAST                0 (n)
34 LOAD_CONST               1 (1)
36 BINARY_OP               10 (-)
40 PRECALL                  1
44 CALL                     1
54 BINARY_OP                5 (*)
58 RETURN_VALUE
```

INTERPRETER

Part 2:
Building a Compiler



Our Language

```
# Compute the x'th fibonacci number.
def fib(x) {
  if (x < 3) {
    return 1
  }
  else {
    return fib(x-1)+fib(x-2)
  }
}

# This expression will compute the 40th number.
fib(40)
```

```
def if else while return break continue int bool void true false
```

```
( { [ ] } ) , ; = + - * / % < > <= >= == != && || !
```

Our Language

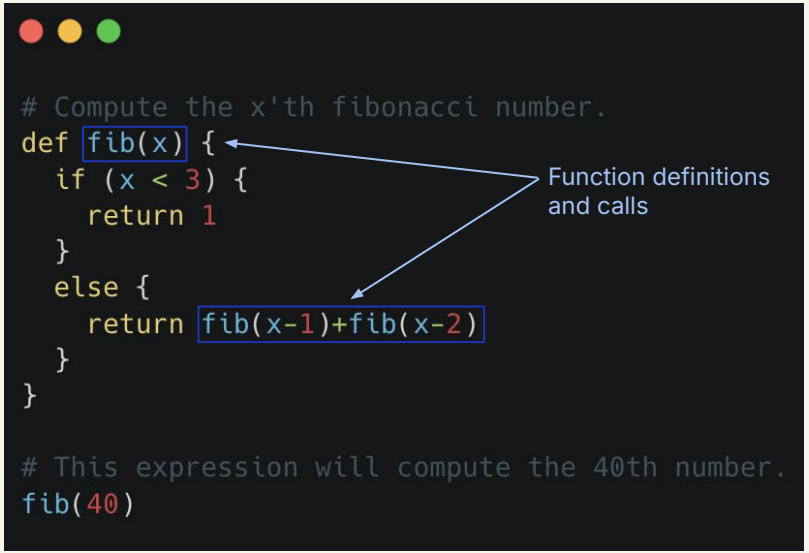
```
# Compute the x'th fibonacci number.
def fib(x) {
  if (x < 3) {
    return 1
  }
  else {
    return fib(x-1)+fib(x-2)
  }
}

# This expression will compute the 40th number.
fib(40)
```

```
def if else while return break continue int bool void true false
```

```
( { [ ] } ) , ; = + - * / % < > <= >= == != && || !
```

An imperative language similar to Java or C, but is greatly **simplified** compared to those languages.

Our Language



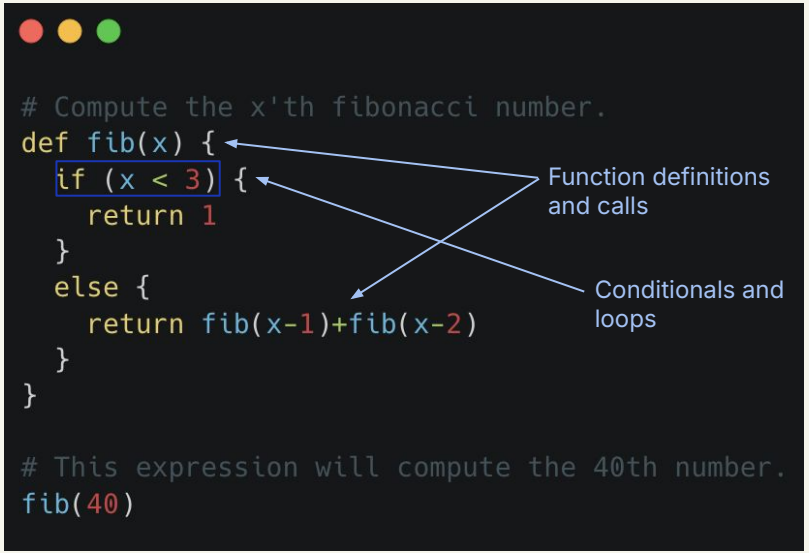

```
# Compute the x'th fibonacci number.
def fib(x) {
  if (x < 3) {
    return 1
  }
  else {
    return fib(x-1)+fib(x-2)
  }
}

# This expression will compute the 40th number.
fib(40)
```

```
def if else while return break continue int bool void true false
```

```
( { [ ] } ) , ; = + - * / % < > <= >= == != && || !
```

An imperative language similar to Java or C, but is greatly **simplified** compared to those languages.

Our Language




```
# Compute the x'th fibonacci number.
def fib(x) {
  if (x < 3) {
    return 1
  }
  else {
    return fib(x-1)+fib(x-2)
  }
}

# This expression will compute the 40th number.
fib(40)
```

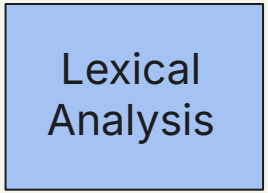
```
def if else while return break continue int bool void true false
```

```
( { [ ] } ) , ; = + - * / % < > <= >= == != && || !
```

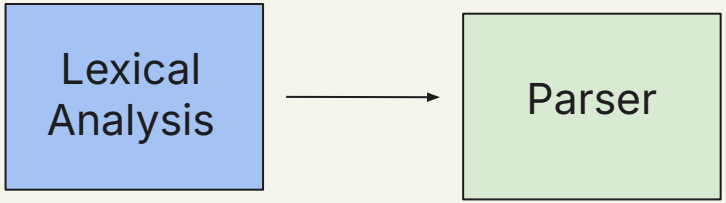
An imperative language similar to Java or C, but is greatly **simplified** compared to those languages.

How a Compiler Works

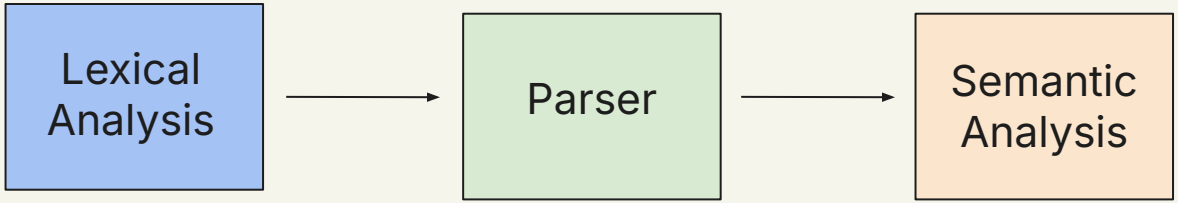
How a Compiler Works



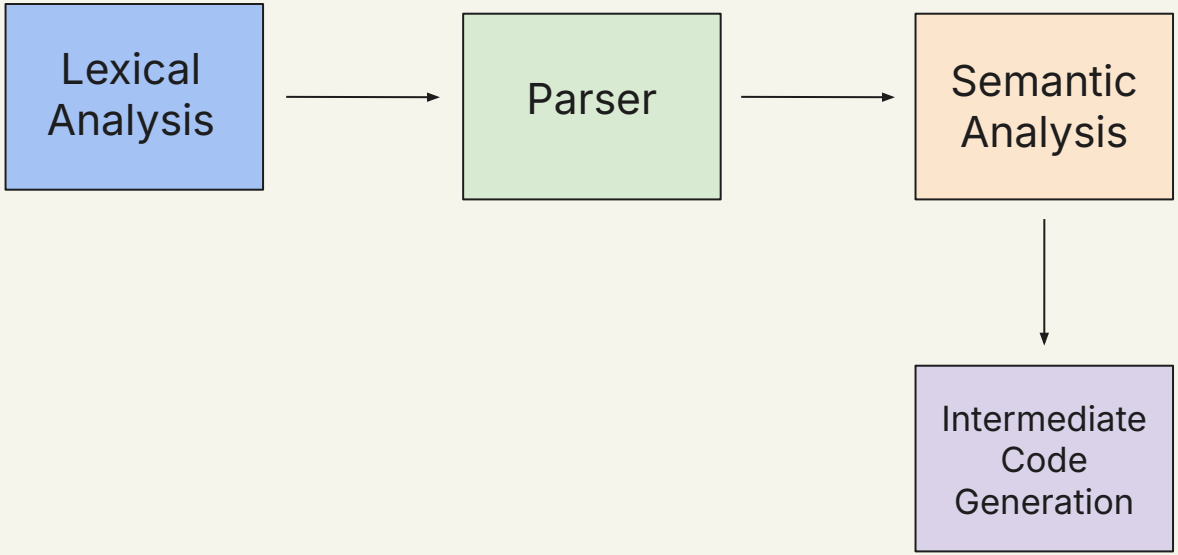
How a Compiler Works



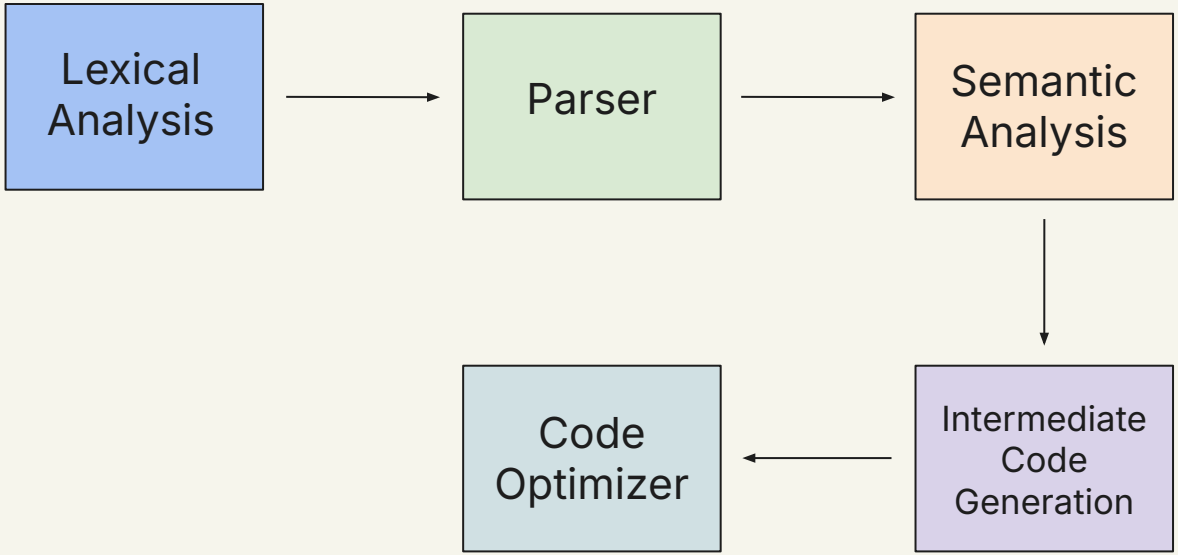
How a Compiler Works



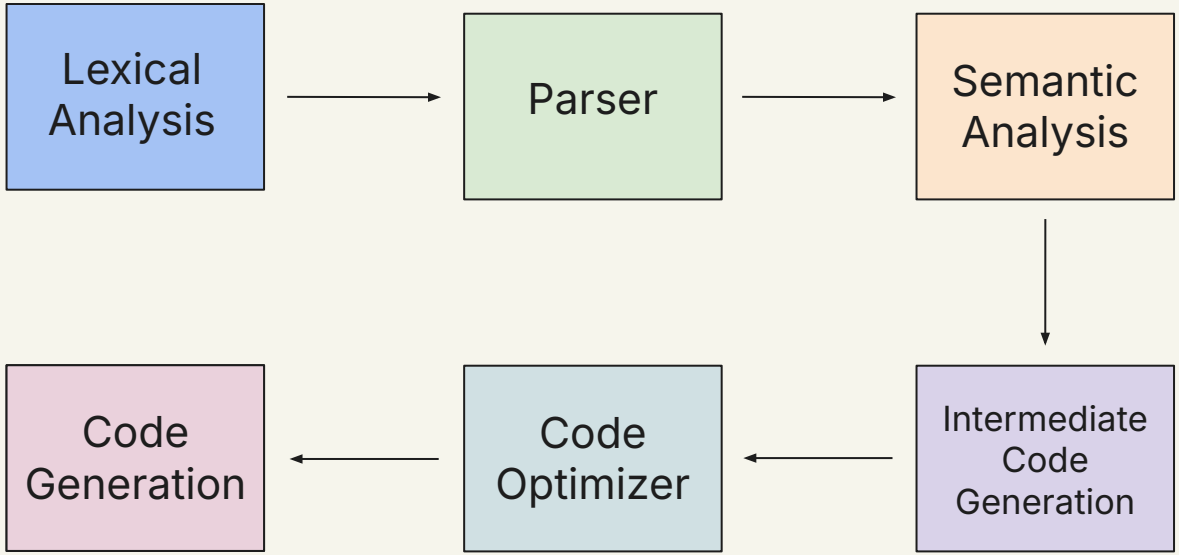
How a Compiler Works



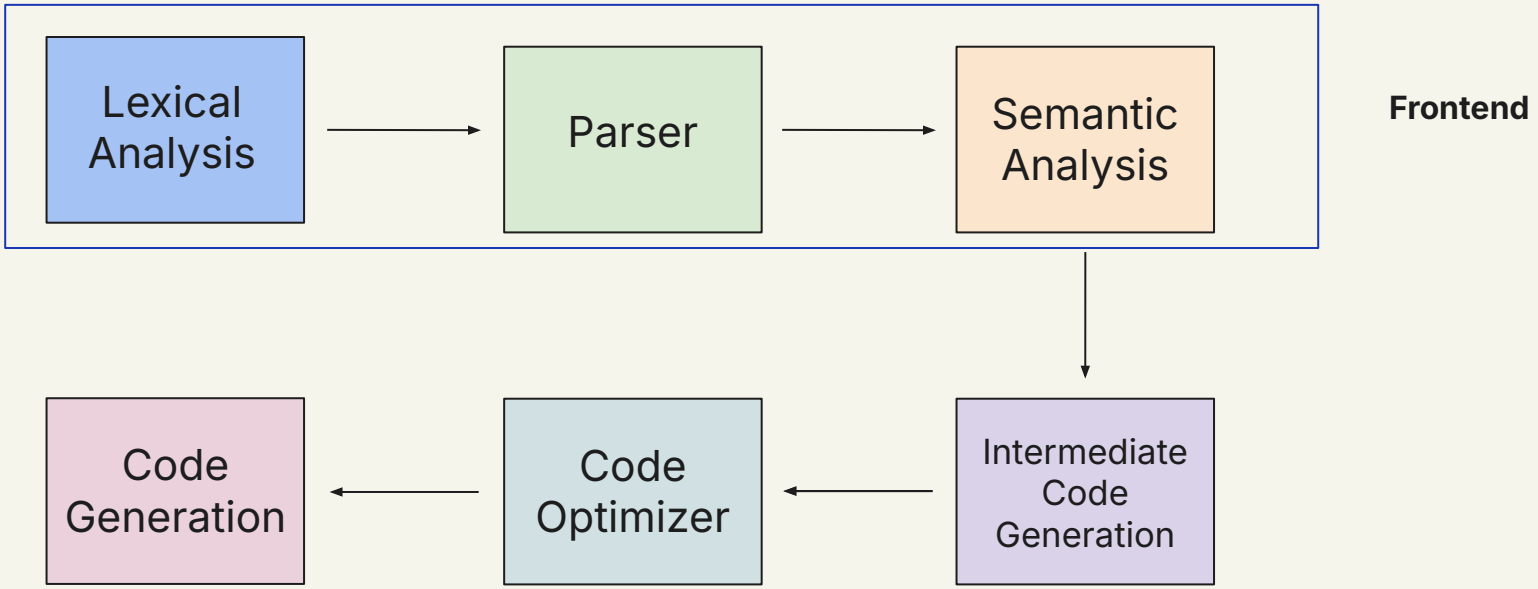
How a Compiler Works



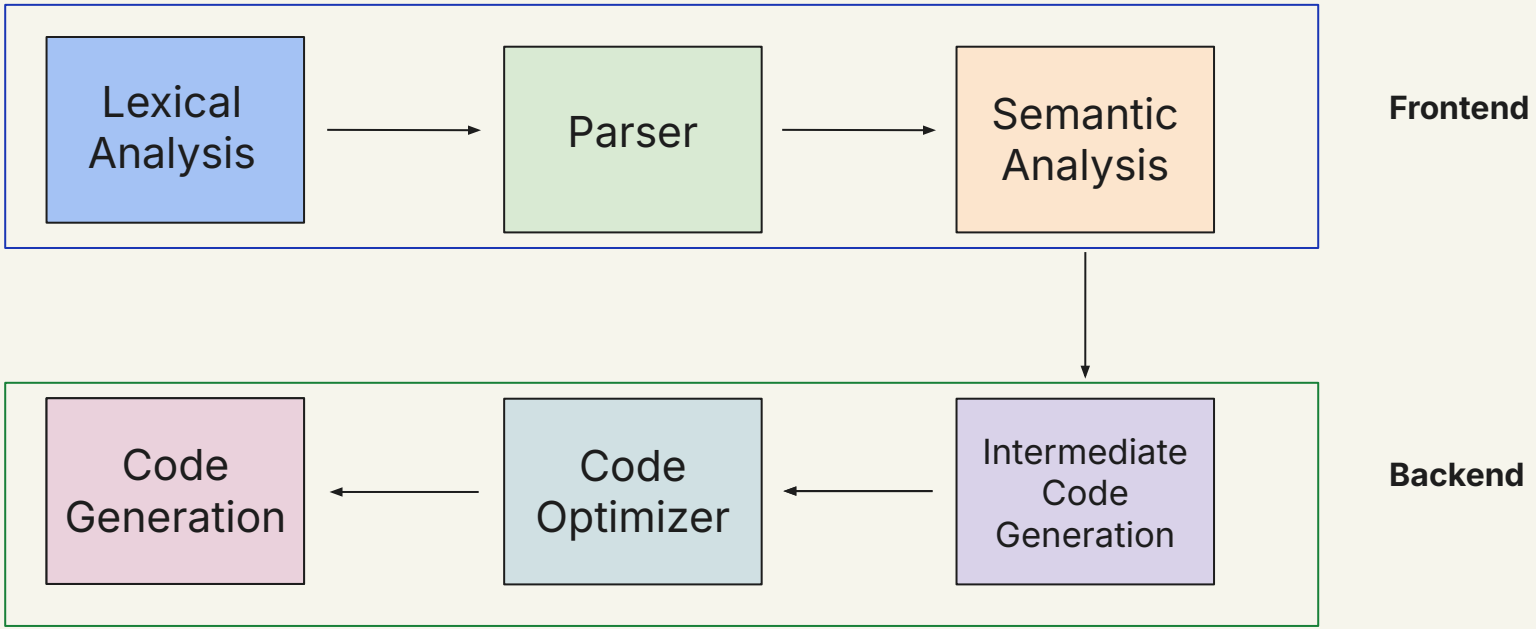
How a Compiler Works



How a Compiler Works



How a Compiler Works

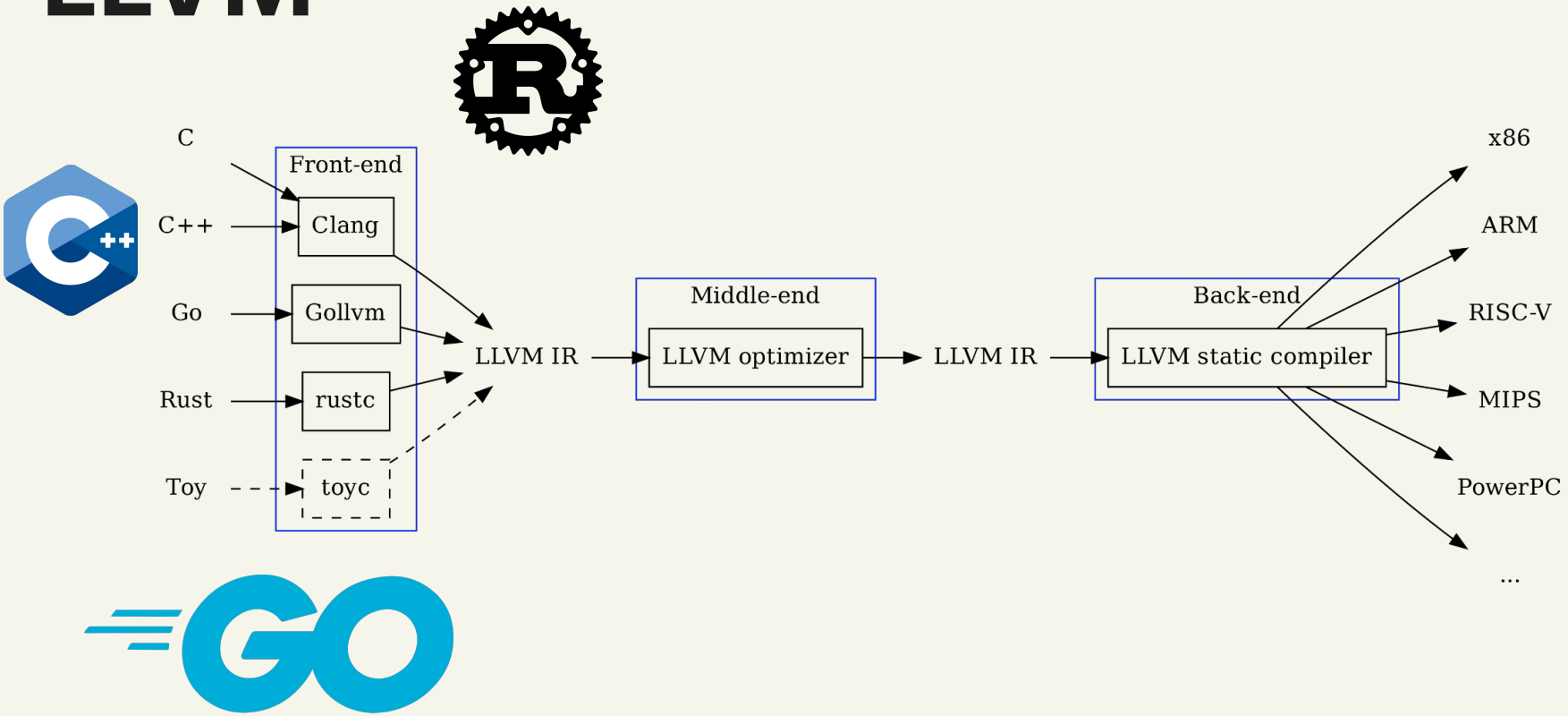


LLVM

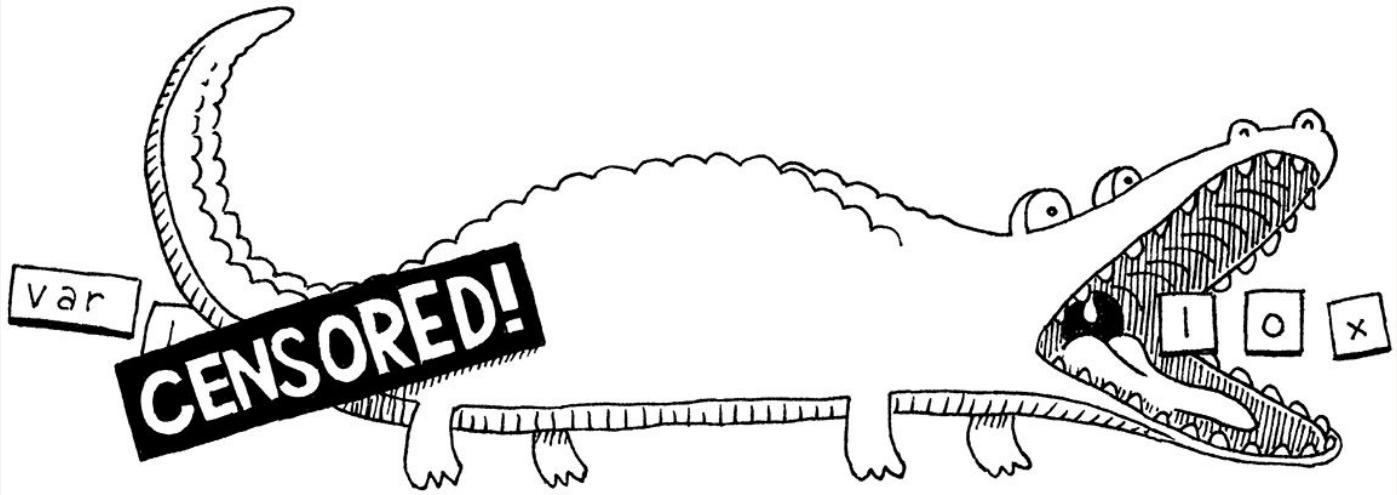
A set of reusable compiler and toolchain technologies

- Backend for almost any instruction set
  - ARM, MIPS, x86, and even PowerPC
- Lots of optimizers
- Extensible, strongly typed intermediate representation
- JIT Compilation

LLVM


C
C++
Go
Rust
Toy
Front-end
Clang
Gollvm
rustc
toyc
LLVM IR
Middle-end
LLVM optimizer
LLVM IR
Back-end
LLVM static compiler
x86
ARM
RISC-V
MIPS
PowerPC
...

Lexical Analysis/Scanning



Token Data Structure

```
struct Token {
  TokenType type;
  std::size_t position;
  std::optional<std::size_t> length;
  std::optional<std::string> value {};
};
```

Token Data Structure

```
struct Token {
  TokenType type;
  std::size_t position;
  std::optional<std::size_t> length;
  std::optional<std::string> value {};
};
```

For string and number literals (i.e. identifiers, key words, etc.)

Lexical Analysis/Scanning

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

Lexical Analysis/Scanning

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
```

Lexical Analysis/Scanning

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
```

Lexical Analysis/Scanning

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
```

Lexical Analysis/Scanning

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
```

Lexical Analysis/Scanning

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
```

Lexical Analysis/Scanning

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
```

Lexical Analysis/Scanning

```
def foo(x) {
   (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
```

Lexical Analysis/Scanning

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
Token Type: CLOSE_CURLY
```

Lexical Analysis/Scanning

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
enum class TokenType {
  DEF,
  IF,
  ELSE,
  WHILE,
  RETURN,
  BREAK,
  CONTINUE,
  INT,
  BOOL,
  VOID,
  TRUE,
  FALSE,

  OPEN_PAREN,
  CLOSE_PAREN,
  OPEN_CURLY,
  CLOSE_CURLY,
  OPEN_BRACKET,
  CLOSE_BRACKET,

  EQUAL,
  EQUAL_EQUAL,
  LESS_THAN,
  GREATER_THAN,
  LESS_THAN_EQUAL,
  GREATER_THAN_EQUAL,
  PLUS,
  MINUS,
  TIMES,
  DIVIDE,
  COMMA,
  SEMICOLON,

  NUMBER,
  IDENTIFIER
};
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
Token Type: CLOSE_CURLY
```

Parsing

Analyzing a string of symbols that conform to the rules of some formal grammar

- Build a parse tree/abstract syntax tree

Parsing

Analyzing a string of symbols that conform to the rules of some formal grammar

- Build a parse tree/abstract syntax tree

Grammar

$$pal \longrightarrow (\texttt{"a"} \mid \texttt{"b"} \mid \texttt{"a"}\ pal\ \texttt{"a"} \mid \texttt{"b"}\ pal\ \texttt{"b"})?$$

String

abbba

Parsing

Analyzing a string of symbols that conform to the rules of some formal grammar

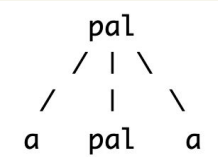
- Build a parse tree/abstract syntax tree

Grammar

$$pal \longrightarrow (\texttt{"a"} \mid \texttt{"b"} \mid \texttt{"a"}\ pal\ \texttt{"a"} \mid \texttt{"b"}\ pal\ \texttt{"b"})?$$

String

abbba ⟶ *“a” pal “a”*



Parsing

Analyzing a string of symbols that conform to the rules of some formal grammar

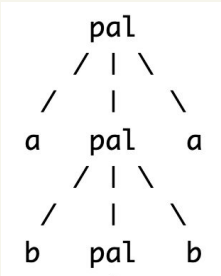
- Build a parse tree/abstract syntax tree

Grammar

$$pal \longrightarrow (\texttt{"a"} \mid \texttt{"b"} \mid \texttt{"a"}\ pal\ \texttt{"a"} \mid \texttt{"b"}\ pal\ \texttt{"b"})?$$

String

abbba ⟶ *“a” pal “a”*
bbb ⟶ *“b” pal “b”*



Parsing

Analyzing a string of symbols that conform to the rules of some formal grammar

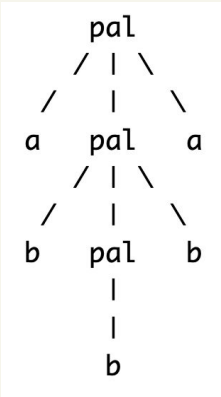
- Build a parse tree/abstract syntax tree

Grammar

$$pal \longrightarrow (\texttt{"a"} \mid \texttt{"b"} \mid \texttt{"a"}\ pal\ \texttt{"a"} \mid \texttt{"b"}\ pal\ \texttt{"b"})?$$

String

abbba ⟶ “a” *pal* “a”
bbb ⟶ “b” *pal* “b”
b ⟶ “b”



Context

Context-Free Grammar (CFG):

A grammar where each rule is of the form:

$A \rightarrow \gamma$

(Replace a single non-terminal with a string, regardless of context.)

Context-Sensitive Grammar (CSG):

A grammar where rules are of the form:

$\alpha A \beta \rightarrow \alpha \gamma \beta$

(Replace a non-terminal **only when it appears in a specific context**.)

Parser Ambiguity

```
Expr        → Term (("+" | "-") Term)*
Term        → Factor (("*" | "/") Factor)*
Factor      → "(" Expr ")" | number | id
```

Grammar

Parser Ambiguity

```
Expr        → Term (("+" | "-") Term)*
Term        → Factor (("*" | "/") Factor)*
Factor      → "(" Expr ")" | number | id
```

Grammar

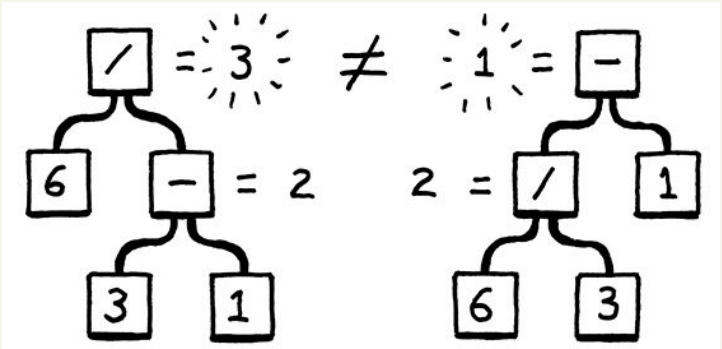
Tokens

Parser Ambiguity

```
Expr        → Term (("+" | "-") Term)*
Term        → Factor (("*" | "/") Factor)*
Factor      → "(" Expr ")" | number | id
```

Grammar

Tokens

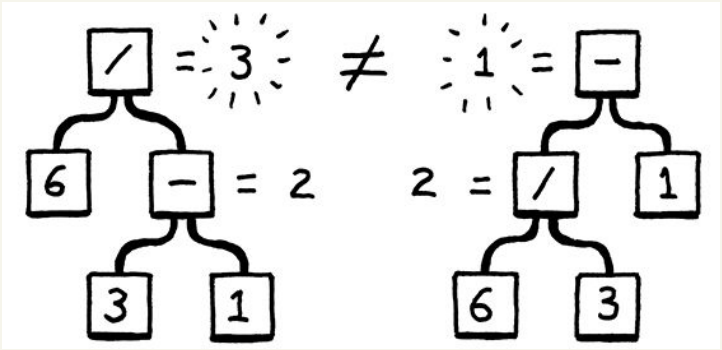


Parser Ambiguity

```
Expr        → Term (("+" | "-") Term)*
Term        → Factor (("*" | "/") Factor)*
Factor      → "(" Expr ")" | number | id
```

Grammar

Tokens



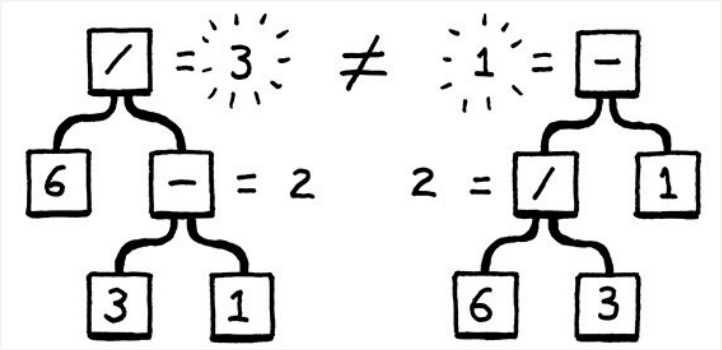
`6 / (3 - 1)`

Parser Ambiguity

```
Expr        → Term (("+" | "-") Term)*
Term        → Factor (("*" | "/") Factor)*
Factor      → "(" Expr ")" | number | id
```

Grammar

Tokens



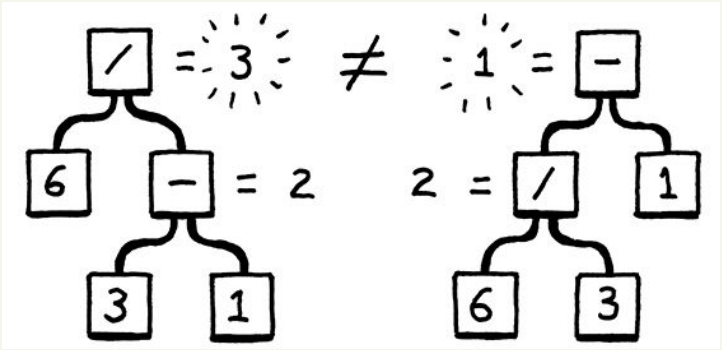
`6 / (3 - 1)` `(6 / 3) - 1`

Parser Ambiguity

```
Expr     → Term (("+" | "-") Term)*
Term     → Factor (("*" | "/") Factor)*
Factor   → "(" Expr ")" | number | id
```

Grammar

Tokens



`6 / (3 - 1)` `(6 / 3) - 1`

A grammar is said to be *ambiguous* if the grammar has more than one parse tree for a given string of tokens.

Operators

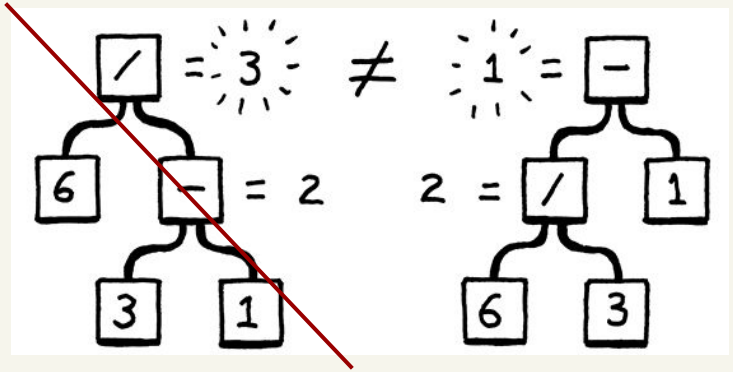
Operator Associativity

- Binary format (sometimes called “bitcode”)
- A portable, high-level assembly language
- Can be optimized with a variety of transformations
    - Multiple passes

Operator Precedence

Operators

Tokens



6 / (3 - 1)

(6 / 3) - 1

Backus–Naur Form (BNF) Grammar

$$
\begin{array}{rcl}
\textit{Program} & \rightarrow & (\textit{Var} \mid \textit{Func})^* \\
\\
\textit{Var} & \rightarrow & \textit{Type}\ \texttt{ID}\ (\text{`[' }\texttt{DEC}\text{ `]'})?\ \text{`;'} \\
\textit{Type} & \rightarrow & \texttt{int} \mid \texttt{bool} \mid \texttt{void} \\
\\
\textit{Func} & \rightarrow & \texttt{def}\ \textit{Type}\ \texttt{ID}\ \text{`('}\ \textit{Params}?\ \text{`)'}\ \textit{Block} \\
\textit{Params} & \rightarrow & \textit{Type}\ \texttt{ID}\ (\text{`,'}\ \textit{Type}\ \texttt{ID})^* \\
\\
\textit{Block} & \rightarrow & \text{`\{'}\ \textit{Var}^*\ \textit{Stmt}^*\ \text{`\}'} \\
\\
\textit{Stmt} & \rightarrow & \textit{Loc}\ \text{`='}\ \textit{Expr}\ \text{`;'} \\
 & \mid & \textit{FuncCall}\ \text{`;'} \\
 & \mid & \texttt{if}\ \text{`('}\ \textit{Expr}\ \text{`)'}\ \textit{Block}\ (\texttt{else}\ \textit{Block})? \\
 & \mid & \texttt{while}\ \text{`('}\ \textit{Expr}\ \text{`)'}\ \textit{Block} \\
 & \mid & \texttt{return}\ \textit{Expr}?\ \text{`;'} \\
 & \mid & \texttt{break}\ \text{`;'} \\
 & \mid & \texttt{continue}\ \text{`;'}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{Expr} & \rightarrow & \textit{Expr}\ \texttt{BINOP}\ \textit{Expr} \\
 & \mid & \texttt{UNOP}\ \textit{BaseExpr} \\
 & \mid & \textit{BaseExpr} \\
\\
\textit{BaseExpr} & \rightarrow & \text{`('}\ \textit{Expr}\ \text{`)'} \\
 & \mid & \textit{Loc} \\
 & \mid & \textit{FuncCall} \\
 & \mid & \textit{Lit} \\
\\
\textit{Loc} & \rightarrow & \texttt{ID}\ (\text{`['}\ \textit{Expr}\ \text{`]'})? \\
\\
\textit{FuncCall} & \rightarrow & \texttt{ID}\ \text{`('}\ \textit{Args}?\ \text{`)'} \\
\textit{Args} & \rightarrow & \textit{Expr}\ (\text{`,'}\ \textit{Expr})^* \\
\\
\textit{Lit} & \rightarrow & \texttt{DEC} \mid \texttt{HEX} \mid \texttt{STR} \mid \texttt{true} \mid \texttt{false}
\end{array}
$$

Parsing

Many Types

1. Top-Down parsing
    a. Predictive parsing/Recursive Descent Parsing
        i. The most common kind of hand-crafted parser
2. Bottom-Up parsing
3. Table-based parsing???

Parsing

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

Parsing

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
```

Parsing

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
```

```
case DecafScanning::TokenType::DEF:
  return parseFuncDefinition();
```

```
function parseFunction():
    match("def")
    matchIdentifier()
    match("(")
    matchIdentifier()
    match(")")
    match("{")
    parseExpr()
    match("}")
```

Simplified pseudo-code

Parsing

```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
```

```
case DecafScanning::TokenType::DEF:
  return parseFuncDefinition();
```

```
function parseFunction():
    match("def")
    matchIdentifier()
    match("(")
    matchIdentifier()
    match(")")
    match("{")
    parseExpr()
    match("}")
```

Simplified pseudo-code

Parsing

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
```

```cpp
std::unique_ptr<AST::Expr> Parser::parseExpr() {
  // Parse any expression (including both the primary ones and bin-ops)
  auto LHS = parsePrimaryExpr();

  if (!LHS)
    return nullptr;

  auto expr = parseBinaryExpr(0, std::move(LHS));
  return expr;
}
```

Parsing

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
```

```cpp
std::unique_ptr<AST::Expr> Parser::parseExpr() {
  // Parse any expression (including both the primary ones and bin-ops)
  auto LHS = parsePrimaryExpr();

  if (!LHS)
    return nullptr;

  auto expr = parseBinaryExpr(0, std::move(LHS));
  return expr;
}
```

```cpp
std::unique_ptr<AST::Expr> Parser::parsePrimaryExpr() {
  // Parse basic, not bin-op expressions
  switch (peek().value().type) {
    default:
      return nullptr; // Todo: Throw an error
    case DecafScanning::TokenType::IDENTIFIER:
      return identifierExpr();
    case DecafScanning::TokenType::NUMBER:
      return numberExpr();
    case DecafScanning::TokenType::OPEN_PAREN:
      return groupingExpr();
```

Parsing

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
```

```cpp
std::unique_ptr<AST::Expr> Parser::parseExpr() {
  // Parse any expression (including both the primary ones and bin-ops)
  auto LHS = parsePrimaryExpr();

  if (!LHS)
    return nullptr;

  auto expr = parseBinaryExpr(0, std::move(LHS));
  return expr;
}
```

```python
def parseExpr(min_prec=0):
    left = parsePrimary()

    while currentToken is operator and precedence(currentToken) >= min_prec:
        op = currentToken
        prec = precedence(op)
        assoc = associativity(op)

        # Left-associative: use prec + 1 for right-hand side
        next_min_prec = prec + 1 if assoc == 'left' else prec
        match(op)
        right = parseExpr(next_min_prec)

        left = BinaryExpr(op, left, right)

    return left
```

“Precedence Climbing”

Parsing

```
Debug Info: Token List
Token Type: DEF
Token Type: IDENTIFIER, Value: foo
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_CURLY
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: PLUS
Token Type: IDENTIFIER, Value: x
Token Type: CLOSE_PAREN
Token Type: OPEN_PAREN
Token Type: IDENTIFIER, Value: x
Token Type: PLUS
Token Type: OPEN_PAREN
Token Type: NUMBER, Value: 1
Token Type: PLUS
Token Type: NUMBER, Value: 2
Token Type: CLOSE_PAREN
Token Type: CLOSE_PAREN
```

```
std::unique_ptr<AST::Expr> Parser::parseExpr() {
  // Parse any expression (including both the primary ones and bin-ops)
  auto LHS = parsePrimaryExpr();

  if (!LHS)
    return nullptr;

  auto expr = parseBinaryExpr(0, std::move(LHS));
  return expr;
}
```

```
// 1 is lowest precedence.
m_binopPrecedence[DecafScanning::TokenType::TIMES] = 4;
m_binopPrecedence[DecafScanning::TokenType::DIVIDE] = 4;
m_binopPrecedence[DecafScanning::TokenType::PLUS] = 3;
m_binopPrecedence[DecafScanning::TokenType::MINUS] = 3;
m_binopPrecedence[DecafScanning::TokenType::LESS_THAN] = 2;
m_binopPrecedence[DecafScanning::TokenType::GREATER_THAN] = 2;
m_binopPrecedence[DecafScanning::TokenType::LESS_THAN_EQUAL] = 2;
m_binopPrecedence[DecafScanning::TokenType::GREATER_THAN_EQUAL] = 2;
m_binopPrecedence[DecafScanning::TokenType::EQUAL_EQUAL] = 1;
```

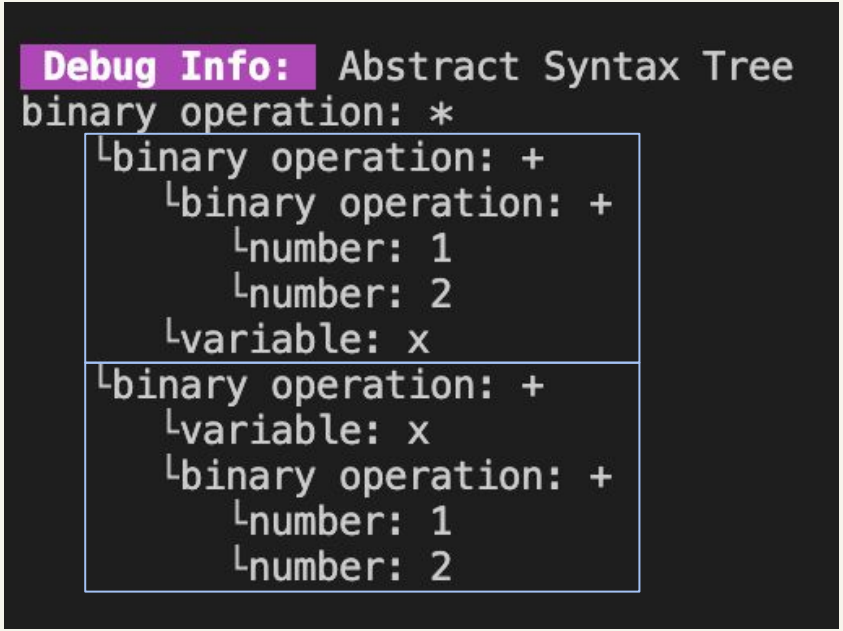
“Precedence Climbing”

Parsing

```
Debug Info: Abstract Syntax Tree
binary operation: *
    └binary operation: +
        └binary operation: +
            └number: 1
            └number: 2
        └variable: x
    └binary operation: +
        └variable: x
        └binary operation: +
            └number: 1
            └number: 2
```

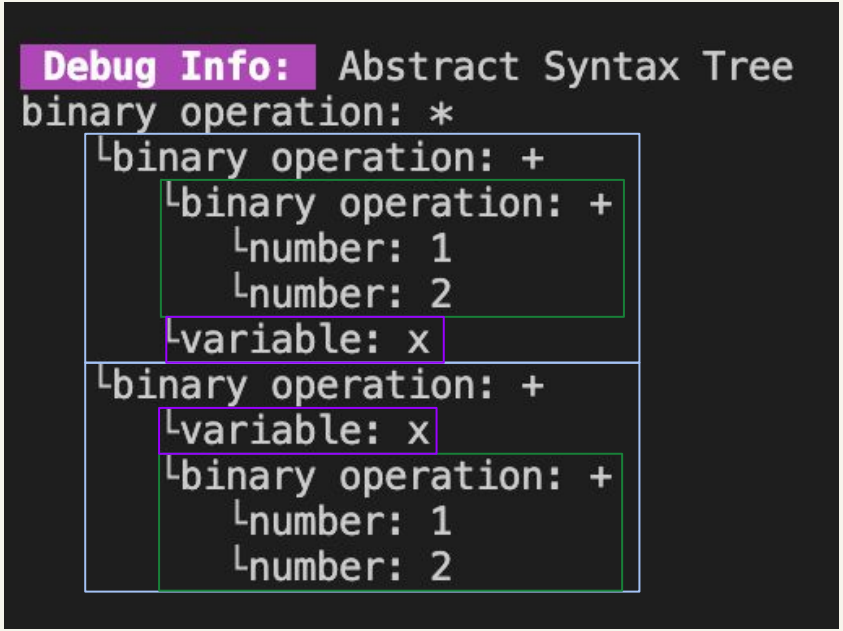
```
def foo(x) {
  (1+2+x)*(x+(1+2))
}
```

Parsing




```
def foo(x) {
   (1+2+x)*(x+(1+2))
}
```

Parsing




```
def foo(x) {
    (1+2+x)*(x+(1+2))
}
```

Intermediate Code Generation

Intermediate Representation

- Convert AST into a low-level, abstract representation of your program
  - Independent of source language and machine architecture
  - Used for optimization and target code generation

LLVM IR

Intermediate Representation

- Binary format (sometimes called “bitcode”)
- A portable, high-level assembly language
- Can be optimized with a variety of transformations
    - Multiple passes

```
@.str = internal constant [14 x i8] c"Hello, world\0A\00"

declare i32 @printf(ptr, ...)

define i32 @main(i32 %argc, ptr %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8], ptr @.str, i32 0, i32 0
    %tmp2 = call i32 (ptr, ...) @printf( ptr %tmp1 ) nounwind
    ret i32 0
}
```

Intermediate Code Generation

```
def codegen(numAST)
  emit(numAST->val)

def codegen(varAST)
  emit(namedValues(varAST->name))

def codegen(binAST)
  L = codegen(binAST->left)
  R = codegen(binAST->right)

  switch(binAST->operand.type):
    emit("addtmp" | "subtmp" | "multmp" | "cmptmp" L, R)
```

Intermediate Code Generation

```
def codegen(functionAST)
  # All values in our toy language are doubles
  # A function takes in a vector of N doubles and returns one double as an output
  setFunctionType("double(double,double,...)")

  # For increased readibility, make all variable names matched the parsed identifiers
  nameArguments(functionAST->args)

  # Read all the argument names into named values map (so we know what variables are available
  # in our assembly)
  namedValues.clear()
  namedValues = functionAST->args

  # Create an entry point for our function
  createBlockAndSetInsertionPoint()
  emit(codegen(functionAST->body))
```

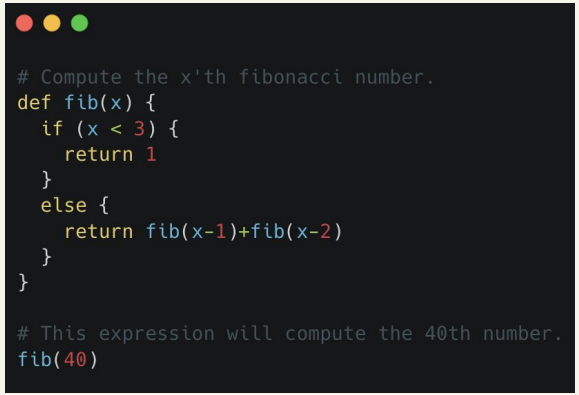
Intermediate Code Generation

```
llvm::Value *NumberExpr::codegen() {
  return llvm::ConstantFP::get(*CodeGenerator::context, llvm::APFloat(value));
}

llvm::Value *VariableExpr::codegen() {
  // Look this variable up in the function.
  llvm::Value *V = CodeGenerator::namedValues[name];
  if (!V)
    std::cout << "Unknown variable name" << std::endl; // To-do: Log error
  return V;
}
```

Intermediate Code Generation

```
llvm::Value *BinaryExpr::codegen() {
  llvm::Value *L = LHS->codegen();
  llvm::Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;

  switch (op.type) {
    case TokenType::PLUS:
      return CodeGenerator::builder->CreateFAdd(L, R, "addtmp");
    case TokenType::MINUS:
      return CodeGenerator::builder->CreateFSub(L, R, "subtmp");
    case TokenType::TIMES:
      return CodeGenerator::builder->CreateFMul(L, R, "multmp");
    case TokenType::LESS_THAN:
      L = CodeGenerator::builder->CreateFCmpULT(L, R, "cmptmp");
      // Convert bool 0/1 to double 0.0 or 1.0
      return CodeGenerator::builder->CreateUIToFP(L, llvm::Type::getDoubleTy(*CodeGenerator::context),
                                  "booltmp");
    // default:
    //   return LogErrorV("invalid binary operator");
  }

  return nullptr; // To-do fix switch default
}
```

Intermediate Code Generation



```
# Compute the x'th fibonacci number.
def fib(x) {
  if (x < 3) {
    return 1
  }
  else {
    return fib(x-1)+fib(x-2)
  }
}

# This expression will compute the 40th number.
fib(40)
```

Intermediate Code Generation

```
# Compute the x'th fibonacci number.
def fib(x) {
  if (x < 3) {
    return 1
  }
  else {
    return fib(x-1)+fib(x-2)
  }
}

# This expression will compute the 40th number.
fib(40)
```

```
Debug Info: Unoptimized function
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  br label %ifcont

else:                                             ; preds = %entry
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp1 = fsub double %x, 2.000000e+00
  %calltmp2 = call double @fib(double %subtmp1)
  %addtmp = fadd double %calltmp, %calltmp2
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

JIT Compilation

- Compile code “on-the-fly” as it is needed
  - Rather than compiling whole programs to disk ahead of time as a traditional compiler does (Ahead-of-time compilation)

`addModule`

```
auto RT = DecafJIT::JIT::JIT_->getMainJITDylib().createResourceTracker();
auto TSM = llvm::orc::ThreadSafeModule(std::move(DecafCodeGen::CodeGenerator::module_), std::move(DecafCodeGen::CodeGenerator::context));
DecafJIT::JIT::exitOnError(DecafJIT::JIT::JIT_->addModule(std::move(TSM), RT));
```

`lookup`

```
auto exprSymbol = DecafJIT::JIT::exitOnError(DecafJIT::JIT::JIT_->lookup("__anon_expr"));
```

JIT Compilation

```
// Get the symbol's address and cast it to the right type (takes no
// arguments, returns a double) so we can call it as a native function.
// double (*FP)() = exprSymbol.getAddress().toPtr<double (*)()>();
double (*FP)() = (double (*)())(intptr_t)exprSymbol.getAddress();
double result = FP();
// fprintf(stderr, "Evaluated to \e[1;37;41m%f\e[m\n\n", FP());
// Delete the anonymous expression module from the JIT.
DecafJIT::JIT::exitOnError(RT->remove());
return result;
```

Optimization

We can now produce IR! But it’s not great....

- Let’s optimize to get nice, efficient code

Optimization

```
(1 + 2 + x) * (x + (1 + 2))
```

"3-Address Code"

```
t1 = 1 + 2
t2 = t1 + x
t3 = 1 + 2
t4 = x + t3
t5 = t2 * t4
```

Optimization

```
(1 + 2 + x) * (x + (1 + 2))
```

“3-Address Code”

```
t1 = 1 + 2
t2 = t1 + x
t3 = 1 + 2
t4 = x + t3
t5 = t2 * t4
```

Trivial Constant Folding

```
- t1 = 1 + 2
+ t1 = 3
- t3 = 1 + 2
+ t3 = 3
```

Optimization

```
(1 + 2 + x) * (x + (1 + 2))
```

"3-Address Code"

```
t1 = 1 + 2
t2 = t1 + x
t3 = 1 + 2
t4 = x + t3
t5 = t2 * t4
```

Trivial Constant Folding

```
- t1 = 1 + 2
+ t1 = 3
- t3 = 1 + 2
+ t3 = 3
```

LLVM IR

```
%addtmp = fadd double 2.000000e+00, 1.000000e+00
%addtmp1 = fadd double %addtmp, %x
ret double %addtmp1
```



```
%addtmp = fadd double %x, 3.000000e+00
%multmp = fmul double %addtmp, %addtmp
ret double %multmp
```

Optimization

```
(1 + 2 + x) * (x + (1 + 2))
```

Common Subexpression Elimination (CSE)

Avoid recomputing the same thing — 1 + 2 was used twice.

```
- t3 = 3
+ t3 = t1
```

"3-Address Code"

```
t1 = 1 + 2
t2 = t1 + x
t3 = 1 + 2
t4 = x + t3
t5 = t2 * t4
```

Optimization

```
(1 + 2 + x) * (x + (1 + 2))
```

"3-Address Code"

```
t1 = 1 + 2
t2 = t1 + x
t3 = 1 + 2
t4 = x + t3
t5 = t2 * t4
```

Common Subexpression Elimination (CSE)

Avoid recomputing the same thing — `1 + 2` was used twice.

```
- t3 = 3
+ t3 = t1
```

Copy Propagation

Replace variables that are just aliases of others

```
- t3 = t1
- t4 = x + t3
+ t4 = x + t1
```

Optimization

```
(1 + 2 + x) * (x + (1 + 2))
```

"3-Address Code"

```
t1 = 1 + 2
t2 = t1 + x
t3 = 1 + 2
t4 = x + t3
t5 = t2 * t4
```

Common Subexpression Elimination (CSE)

Avoid recomputing the same thing — 1 + 2 was used twice.

```
- t3 = 3
+ t3 = t1
```

Copy Propagation

Replace variables that are just aliases of others

```
- t3 = t1
- t4 = x + t3
+ t4 = x + t1
```

LLVM IR

```
%addtmp = fadd double 3.000000e+00, %x
%addtmp1 = fadd double %x, 3.000000e+00
%multmp = fmul double %addtmp, %addtmp1
ret double %multmp
```

```
%addtmp = fadd double %x, 3.000000e+00
%multmp = fmul double %addtmp, %addtmp
ret double %multmp
```

Other Optimizations

Dead Code Elimination

Remove unused or “dead” code

Strength Reduction* / Peephole Optimization

Replace expensive operations with cheaper ones

```
t = x * 2
```

```
t = x << 1  # Bit shift instead of multiplication
```

Other Optimizations

Loop-Invariant Code Motion

Move computations out of loops if they don’t change per iteration

```
for (i = 0; i < n; i++) {
    y = (1 + 2 + x) * (x + (1 + 2));
}
```

Other Optimizations

Loop-Invariant Code Motion

Move computations out of loops if they don't change per iteration

```
for (i = 0; i < n; i++) {
    y = (1 + 2 + x) * (x + (1 + 2));
}
```

Move outside loop

```
t1 = 3
t2 = x + t1
t4 = x + t1
t5 = t2 * t4

for (i = 0; i < n; i++) {
    y = t5
}
```

Other Optimizations

Loop-Invariant Code Motion

Move computations out of loops if they don't change per iteration

```
for (i = 0; i < n; i++) {
    y = (1 + 2 + x) * (x + (1 + 2));
}
```

Move outside loop

Reduced computation from O(n) → O(1)

```
t1 = 3
t2 = x + t1
t4 = x + t1
t5 = t2 * t4

for (i = 0; i < n; i++) {
    y = t5
}
```

Other Optimizations

Loop-Invariant Code Motion

Move computations out of loops if they don't change per iteration

```
for (i = 0; i < n; i++) {
    y = (1 + 2 + x) * (x + (1 + 2));
}
```

Move outside loop

Reduced computation from O(n) → O(1)

```
t1 = 3
t2 = x + t1
t4 = x + t1
t5 = t2 * t4

for (i = 0; i < n; i++) {
    y = t5
}
```

And so much more… *seriously….*

LLVM Optimizations

```
std::unique_ptr<llvm::FunctionPassManager> CodeGenerator::FPM = std::make_unique<llvm::FunctionPassManager>();
std::unique_ptr<llvm::LoopAnalysisManager> CodeGenerator::LAM = std::make_unique<llvm::LoopAnalysisManager>();
std::unique_ptr<llvm::FunctionAnalysisManager> CodeGenerator::FAM = std::make_unique<llvm::FunctionAnalysisManager>();
std::unique_ptr<llvm::CGSCCAnalysisManager> CodeGenerator::CGAM = std::make_unique<llvm::CGSCCAnalysisManager>();
std::unique_ptr<llvm::ModuleAnalysisManager> CodeGenerator::MAM = std::make_unique<llvm::ModuleAnalysisManager>();
std::unique_ptr<llvm::PassInstrumentationCallbacks> CodeGenerator::PIC = std::make_unique<llvm::PassInstrumentationCallbacks>();
std::unique_ptr<llvm::StandardInstrumentations> CodeGenerator::SI;
llvm::PassBuilder CodeGenerator::PB;
```

LLVM Optimizations

```
std::unique_ptr<llvm::FunctionPassManager> CodeGenerator::FPM = std::make_unique<llvm::FunctionPassManager>();
std::unique_ptr<llvm::LoopAnalysisManager> CodeGenerator::LAM = std::make_unique<llvm::LoopAnalysisManager>();
std::unique_ptr<llvm::FunctionAnalysisManager> CodeGenerator::FAM = std::make_unique<llvm::FunctionAnalysisManager>();
std::unique_ptr<llvm::CGSCCAnalysisManager> CodeGenerator::CGAM = std::make_unique<llvm::CGSCCAnalysisManager>();
std::unique_ptr<llvm::ModuleAnalysisManager> CodeGenerator::MAM = std::make_unique<llvm::ModuleAnalysisManager>();
std::unique_ptr<llvm::PassInstrumentationCallbacks> CodeGenerator::PIC = std::make_unique<llvm::PassInstrumentationCallbacks>();
std::unique_ptr<llvm::StandardInstrumentations> CodeGenerator::SI;
llvm::PassBuilder CodeGenerator::PB;
```

```
Debug Info: Unoptimized function
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  br label %ifcont

else:                                             ; preds = %entry
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp1 = fsub double %x, 2.000000e+00
  %calltmp2 = call double @fib(double %subtmp1)
  %addtmp = fadd double %calltmp, %calltmp2
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```

LLVM Optimizations

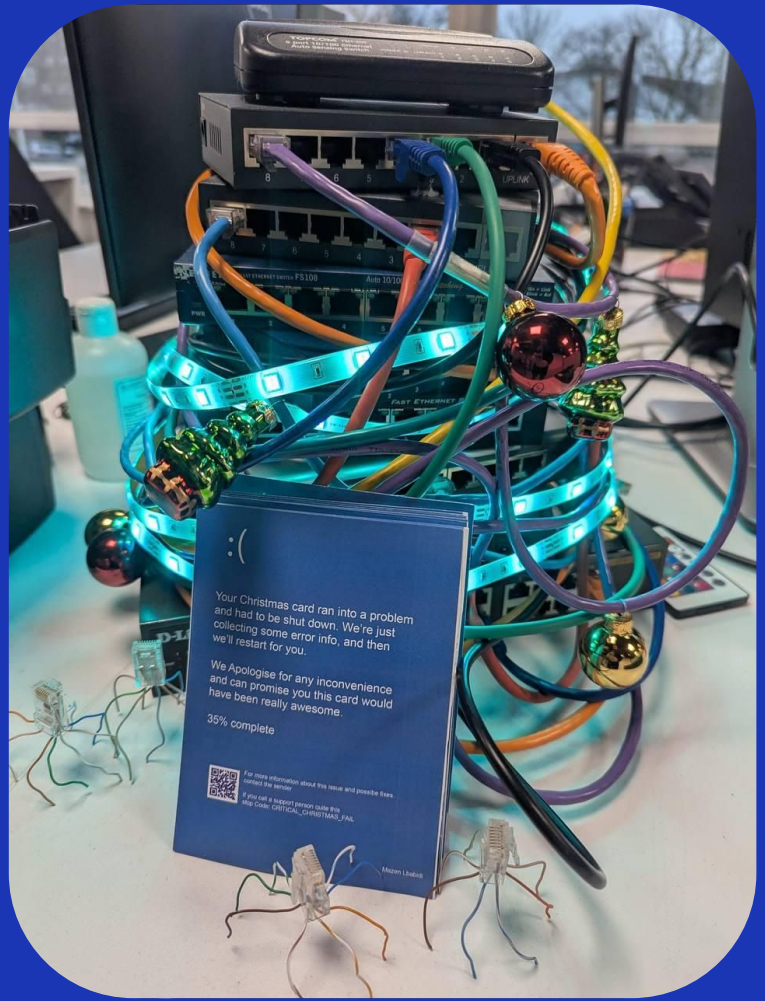
```
std::unique_ptr<llvm::FunctionPassManager> CodeGenerator::FPM = std::make_unique<llvm::FunctionPassManager>();
std::unique_ptr<llvm::LoopAnalysisManager> CodeGenerator::LAM = std::make_unique<llvm::LoopAnalysisManager>();
std::unique_ptr<llvm::FunctionAnalysisManager> CodeGenerator::FAM = std::make_unique<llvm::FunctionAnalysisManager>();
std::unique_ptr<llvm::CGSCCAnalysisManager> CodeGenerator::CGAM = std::make_unique<llvm::CGSCCAnalysisManager>();
std::unique_ptr<llvm::ModuleAnalysisManager> CodeGenerator::MAM = std::make_unique<llvm::ModuleAnalysisManager>();
std::unique_ptr<llvm::PassInstrumentationCallbacks> CodeGenerator::PIC = std::make_unique<llvm::PassInstrumentationCallbacks>();
std::unique_ptr<llvm::StandardInstrumentations> CodeGenerator::SI;
llvm::PassBuilder CodeGenerator::PB;
```

```
Debug Info: Unoptimized function
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  br label %ifcont

else:                                             ; preds = %entry
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp1 = fsub double %x, 2.000000e+00
  %calltmp2 = call double @fib(double %subtmp1)
  %addtmp = fadd double %calltmp, %calltmp2
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]
  ret double %iftmp
}
```



```
Debug Info: Optimized function
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  br i1 %cmptmp, label %ifcont, label %else

else:                                             ; preds = %entry
  %subtmp = fadd double %x, -1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp1 = fadd double %x, -2.000000e+00
  %calltmp2 = call double @fib(double %subtmp1)
  %addtmp = fadd double %calltmp, %calltmp2
  br label %ifcont

ifcont:                                           ; preds = %entry, %else
  %iftmp = phi double [ %addtmp, %else ], [ 1.000000e+00, %entry ]
  ret double %iftmp
}

Read function definition:define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  br i1 %cmptmp, label %ifcont, label %else

else:                                             ; preds = %entry
  %subtmp = fadd double %x, -1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp1 = fadd double %x, -2.000000e+00
  %calltmp2 = call double @fib(double %subtmp1)
  %addtmp = fadd double %calltmp, %calltmp2
  br label %ifcont

ifcont:                                           ; preds = %entry, %else
  %iftmp = phi double [ %addtmp, %else ], [ 1.000000e+00, %entry ]
  ret double %iftmp
}
```

Part 3:
DEMO!


:(
Your Christmas card ran into a problem and had to be shut down. We're just collecting some error info, and then we'll restart for you.
We Apologise for any inconvenience and can promise you this card would have been really awesome.
35% complete
For more information about this issue and possible fixes, contact the sender
If you call a support person quote this stop Code: CRITICAL_CHRISTMAS_FAIL
Mezen Lbakib

Unit Tests

```
# Compute the x'th fibonacci number.
def fib(x) {
  if (x < 3) {
    return 1
  }
  else {
    return fib(x-1)+fib(x-2)
  }
}

# This expression will compute the 40th number.
fib(40)
```

```
DecafScanning::Lexer lexer(content);
std::vector<DecafScanning::Token> tokens(lexer.tokenize());
DecafParsing::Parser parser(tokens);
std::unique_ptr<DecafParsing::AST::Function> AST;

DecafJIT::JIT::initJIT();
DecafCodeGen::CodeGenerator::initializeModuleAndPassManager();

double result = 0.0;
while (!parser.isAtEnd()) {
  // Parse program consiting of functions or (To-do) top level declarations
  switch (parser.peek().value().type) {
    default:
      result = DecafJIT::handleTopLevelStatement(&parser);
    case DecafScanning::TokenType::DEF:
      DecafJIT::handleFuncDefinition(&parser);
  }
}

REQUIRE( result == 102334155.0 );
```

Unit Tests

```
# Compute the x'th fibonacci number.
def fib(x) {
  if (x < 3) {
    return 1
  }
  else {
    return fib(x-1)+fib(x-2)
  }
}

# This expression will compute the 40th number.
fib(40)
```

```
DecafScanning::Lexer lexer(content);
std::vector<DecafScanning::Token> tokens(lexer.tokenize());
DecafParsing::Parser parser(tokens);
std::unique_ptr<DecafParsing::AST::Function> AST;

DecafJIT::JIT::initJIT();
DecafCodeGen::CodeGenerator::initializeModuleAndPassManager();

double result = 0.0;
while (!parser.isAtEnd()) {
  // Parse program consiting of functions or (To-do) top level declarations
  switch (parser.peek().value().type) {
    default:
      result = DecafJIT::handleTopLevelStatement(&parser);
    case DecafScanning::TokenType::DEF:
      DecafJIT::handleFuncDefinition(&parser);
  }
}

REQUIRE( result == 102334155.0 );
```

```
===============================================================================
All tests passed (1 assertion in 1 test case)
```

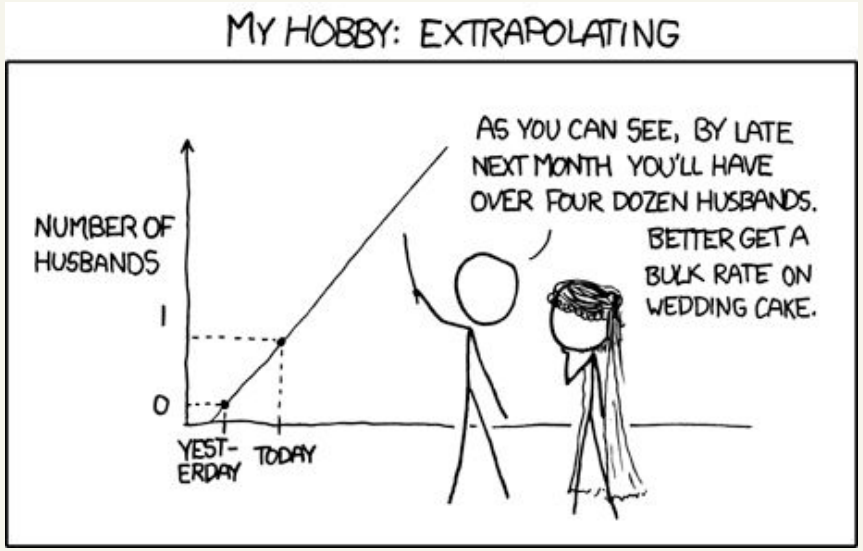
Next Time....

Remaining Topics

- JIT Compilation
- Syntax/Semantic Analysis

Additional Explorations

- Custom Code Generation
  - MIPS, Arm, etc.

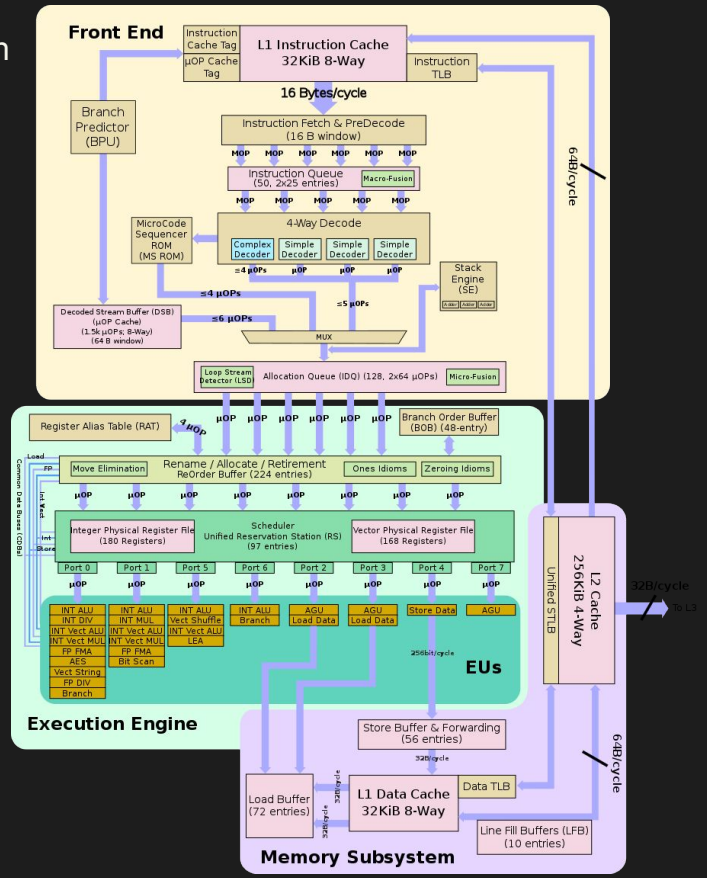


Fin.

Now go build a compiler...

Block diagram of each core

Intel Skylake chip



Context[1]

$L = \{ a^n b^n c^n \mid n \geq 1 \}$

S → aSBC | abc
CB → HB
HB → HC
HC → BC
aB → ab
bB → bb
bC → bc
cC → cc

(It's a bit technical, but the idea is that the grammar carefully keeps the counts of a, b, and c equal, which requires context.)