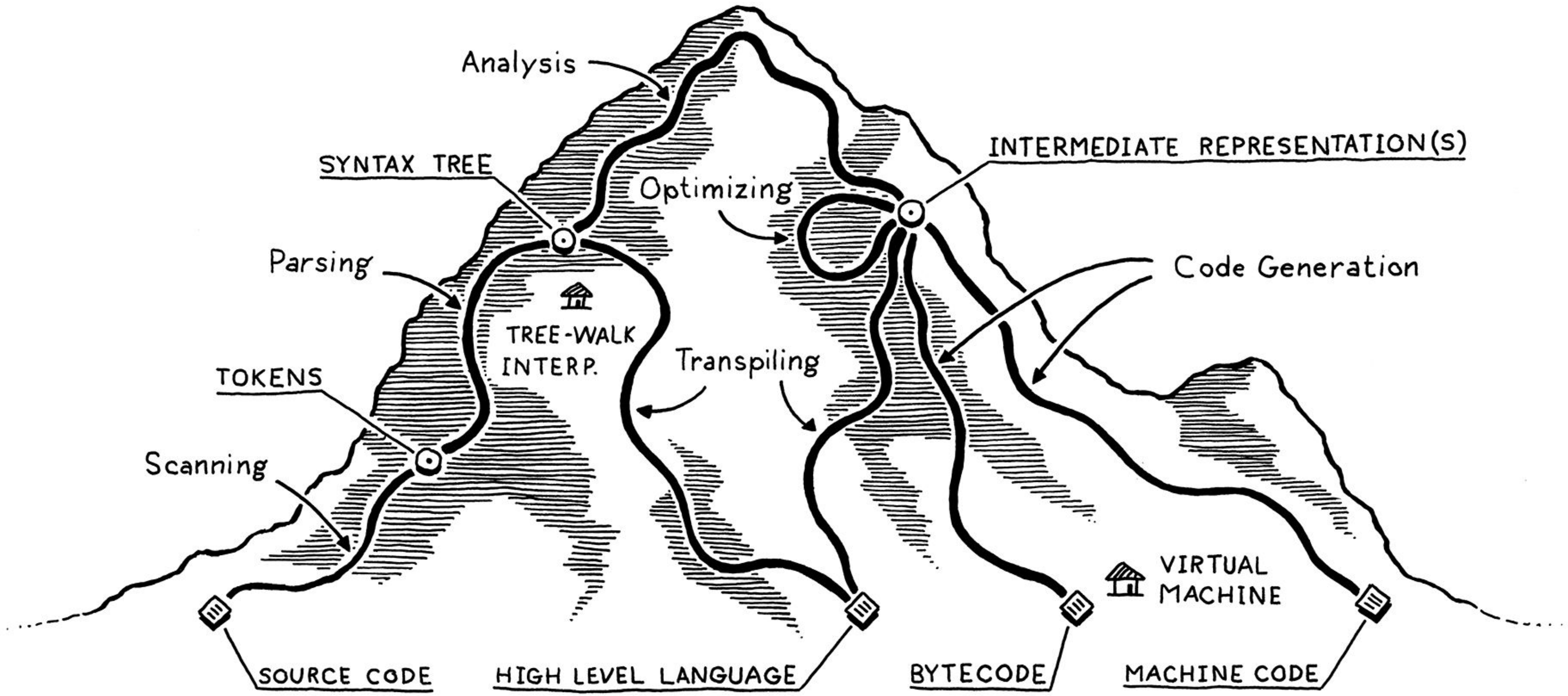
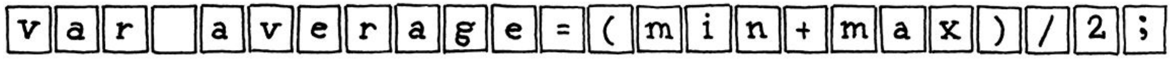
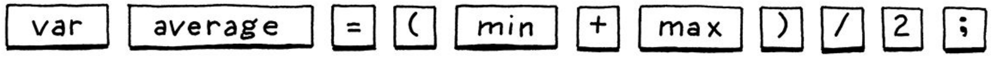
Lexer / Scanner

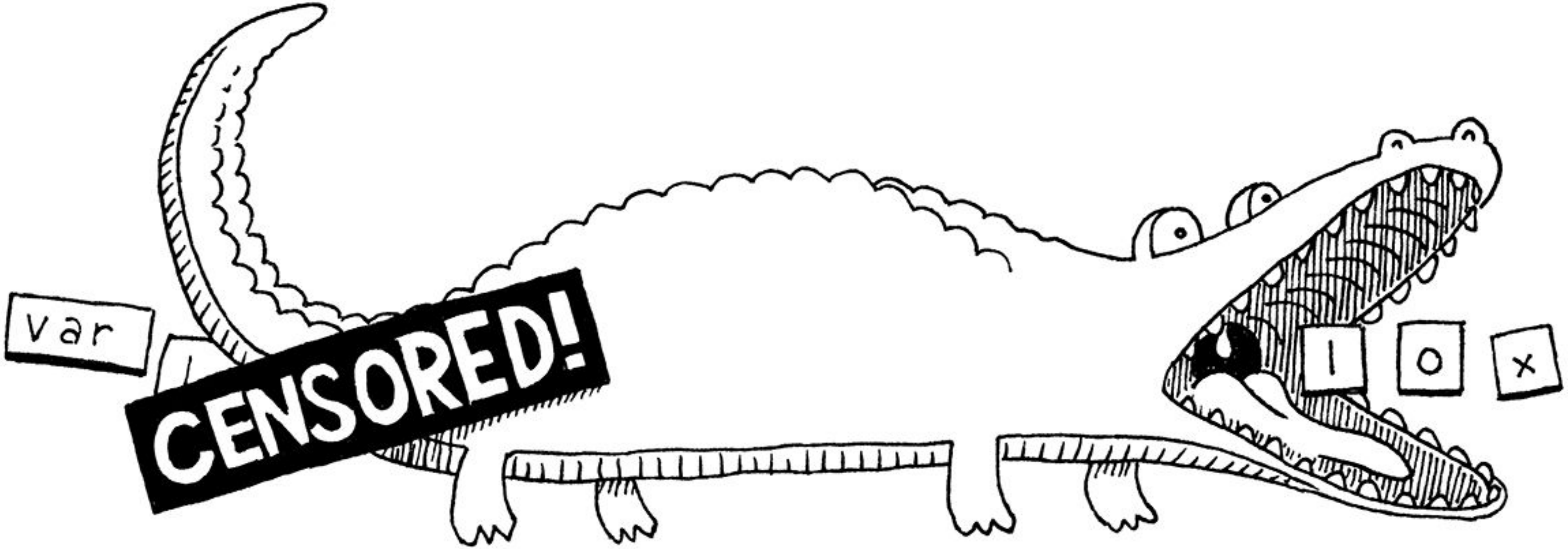
divide et impera


Analysis
SYNTAX TREE
INTERMEDIATE REPRESENTATION(S)
Optimizing
Parsing
Code Generation
TREE-WALK INTERP.
Transpiling
TOKENS
Scanning
VIRTUAL MACHINE
SOURCE CODE
HIGH LEVEL LANGUAGE
BYTECODE
MACHINE CODE

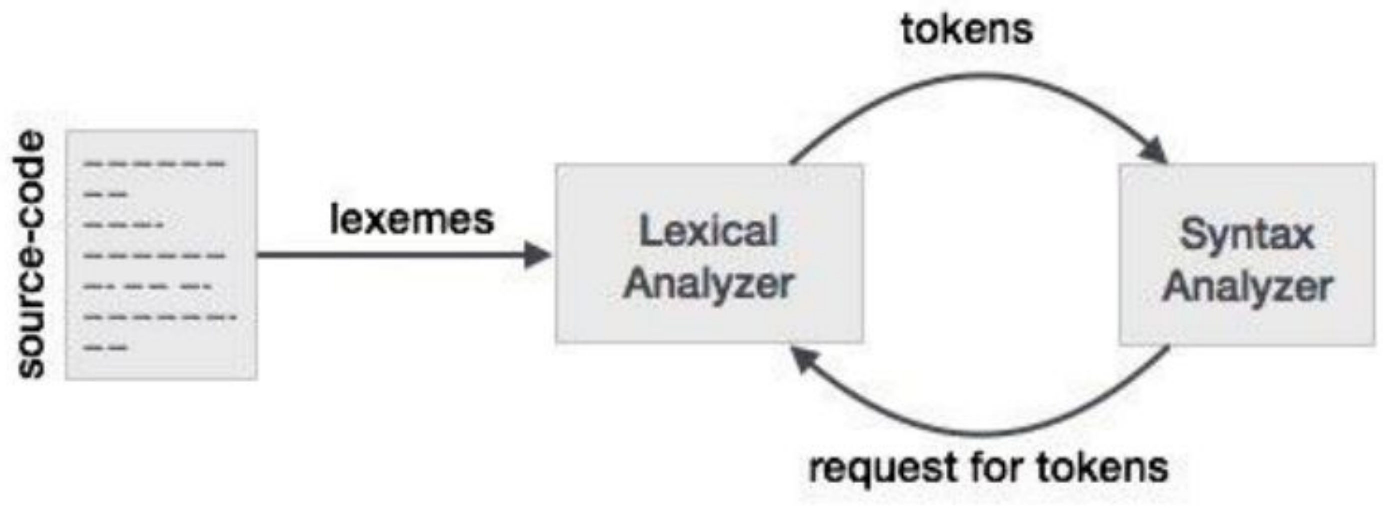


A **scanner** (or **“lexer”**) takes in the linear stream of characters and chunks them together into a series of something more akin to “words”. In programming languages, each of these words is called a **token**. Some tokens are single characters, like `(` and `,`. Others may be several characters long, like numbers (`123`), string literals (`"hi!"`), and identifiers (`min`).




var
CENSORED!
l
o
x

LEXICAL ANALYSIS



Syntax Examples

Value to name biding

```
let age = 1;
let name = "String";
let result = 10 * (20 / 2);
```

Arrays and Maps

```
let myArray = [0, 1, 2, 3, 4, 5];

let map = {"name": "First_Name", "age": 28};

myArray[0] // => 0
map["name"] // => "First_Name"
```

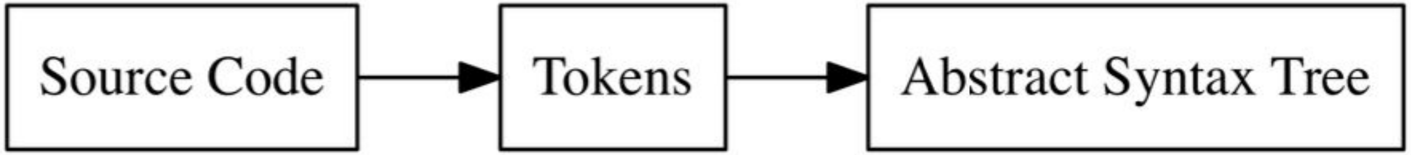
Function declaration and call

```
fn add(first Int, second Int) Int {
    return first + second
}

let sum = add(first: 2, second: 4);
```

Recursive function call

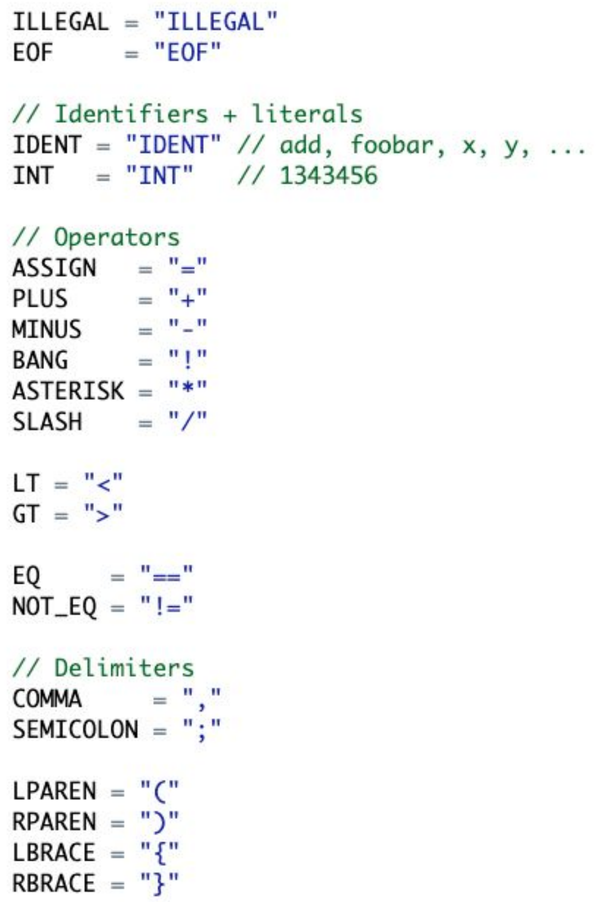
```
fn fibonacci(number Int) {
    if x == 0 {
        return 0;
    } else {
        if x == 1 {
            return 1;
        } else {
            fibonacci(number: x - 1) + fibonacci(number: x - 2);
        }
    }
}
```

Lexer

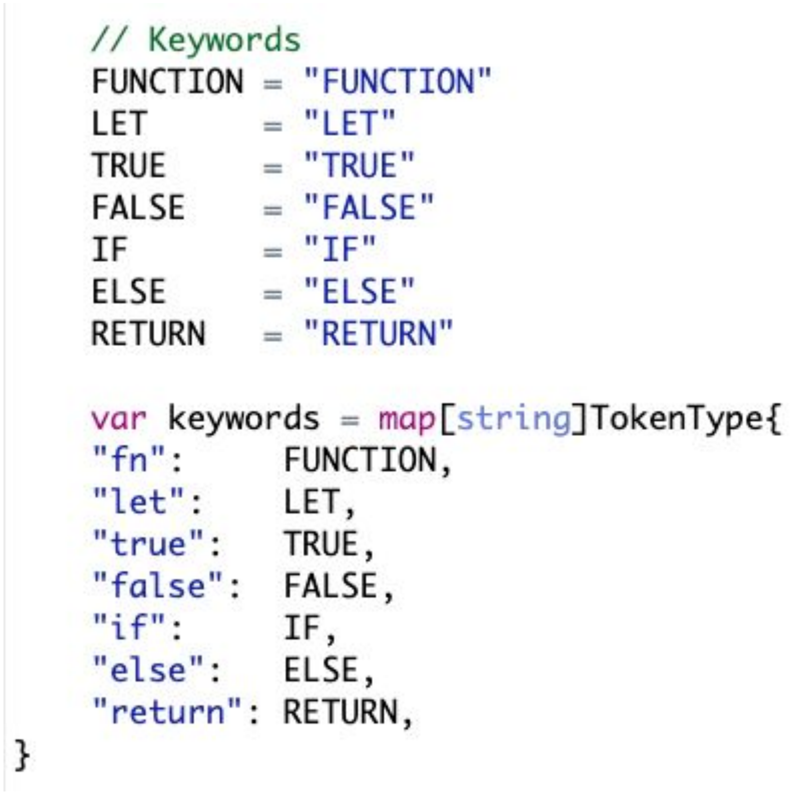


```
"let x = 5 + 5;"  ──────────→  [
                                  LET,
                                  IDENTIFIER("x"),
                                  EQUAL_SIGN,
                                  INTEGER(5),
                                  PLUS_SIGN,
                                  INTEGER(5),
                                  SEMICOLON
                                ]
```

Tokens and Keywords



```
ILLEGAL = "ILLEGAL"
EOF     = "EOF"

// Identifiers + literals
IDENT = "IDENT" // add, foobar, x, y, ...
INT   = "INT"   // 1343456

// Operators
ASSIGN   = "="
PLUS     = "+"
MINUS    = "-"
BANG     = "!"
ASTERISK = "*"
SLASH    = "/"

LT = "<"
GT = ">"

EQ     = "=="
NOT_EQ = "!="

// Delimiters
COMMA     = ","
SEMICOLON = ";"

LPAREN = "("
RPAREN = ")"
LBRACE = "{"
RBRACE = "}"
```

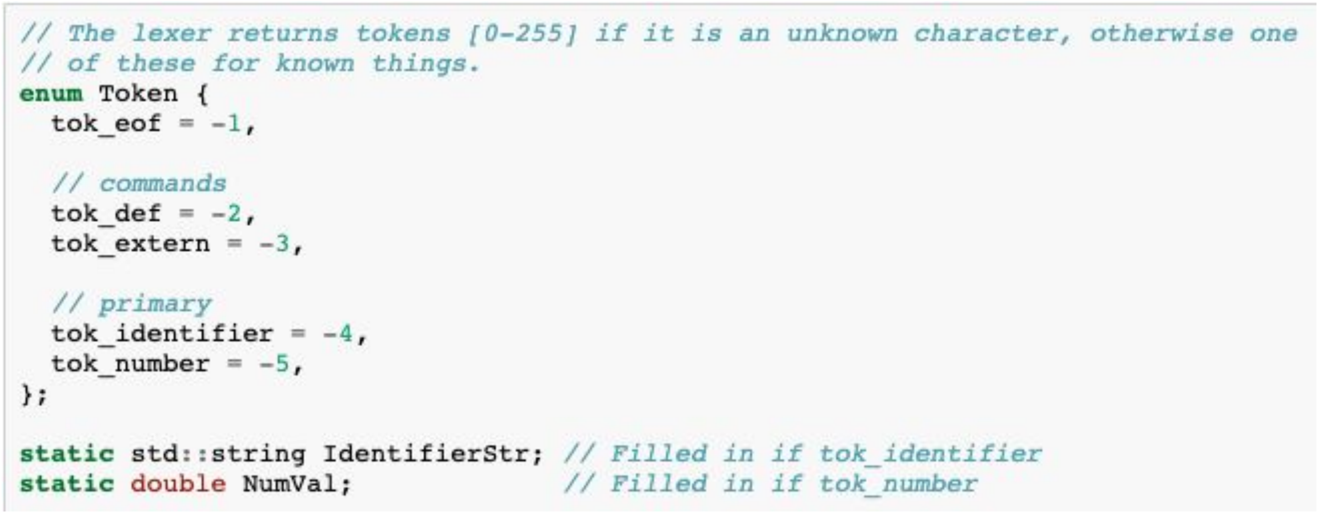


```
    // Keywords
    FUNCTION = "FUNCTION"
    LET      = "LET"
    TRUE     = "TRUE"
    FALSE    = "FALSE"
    IF       = "IF"
    ELSE     = "ELSE"
    RETURN   = "RETURN"

    var keywords = map[string]TokenType{
    "fn":     FUNCTION,
    "let":    LET,
    "true":   TRUE,
    "false":  FALSE,
    "if":     IF,
    "else":   ELSE,
    "return": RETURN,
}
```

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

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

Tokenization

When it comes to implementing a language, the first thing needed is the ability to process a text file and recognize what it says. The traditional way to do this is to use a “lexer” (aka ‘scanner’) to break the input up into “tokens”. Each token returned by the lexer includes a token code and potentially some metadata (e.g. the numeric value of a number). First, we define the possibilities:



```
// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2,
  tok_extern = -3,

  // primary
  tok_identifier = -4,
  tok_number = -5,
};

static std::string IdentifierStr; // Filled in if tok_identifier
static double NumVal;             // Filled in if tok_number
```

The actual implementation of the lexer is a single function named `gettok`. The `gettok` function is called to return the next token from standard input. Its definition starts as:

```
/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();
```

The next thing `gettok` needs to do is recognize identifiers and specific keywords like “def”. Kaleidoscope does this with this simple loop:

```
if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
  IdentifierStr = LastChar;
  while (isalnum((LastChar = getchar())))
    IdentifierStr += LastChar;

  if (IdentifierStr == "def")
    return tok_def;
  if (IdentifierStr == "extern")
    return tok_extern;
  return tok_identifier;
}
```

When reading a numeric value from input, we use the C `strtod` function to convert it to a numeric value that we store in `NumVal`.

```
if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
  std::string NumStr;
  do {
    NumStr += LastChar;
    LastChar = getchar();
  } while (isdigit(LastChar) || LastChar == '.');

  NumVal = strtod(NumStr.c_str(), 0);
  return tok_number;
}
```

Next we handle comments:

```
if (LastChar == '#') {
  // Comment until end of line.
  do
    LastChar = getchar();
  while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

  if (LastChar != EOF)
    return gettok();
}
```

We handle comments by skipping to the end of the line and then return the next token. Finally, if the input doesn’t match one of the above cases, it is either an operator character like ‘+’ or the end of the file. These are handled with this code:

```
  // Check for end of file.  Don't eat the EOF.
  if (LastChar == EOF)
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}
```

Compilers are upgraded through a process called bootstrapping. At first you have your old system compiler that produces slow code and doesn't support all the new language features. You then use this old system compiler to build the new version of the compiler, hoping that the old compiler is able to build the new compiler (it supports all needed features). This produces a new compiler that produces fast code and supports all the new features, however the compiler itself is slow, because it was compiled with a compiler that produces slow code. In addition, the new compiler may be buggy because your old compiler was buggy, or perhaps the new compiler release has a bug.

The next step is to use your new slow compiler that produces fast code, and then build the new compiler again. This produces a fast compiler that produces fast code. However, the first compiler we built could be buggy, and the compiler we just built using it may be defect. We need to verify the correctness of our new fast compiler that produces fast code.

To solve that problem, we build the compiler a third time. Once we have built the third compiler using the second compiler, it should produce the very same output as the first compiler building the second, as both times we are using compilers that produce fast code and use the same source code. The compiler build system will then verify that the second and third compilers are identical, which gives you confidence in the bootstrap. If the second and third compilers are not identical, the bootstrap failed and you have encountered a compiler bug. Bootstrapping takes three times as long as just building a regular compiler, but it makes sure your toolchain is stable.

The last thing to do is run the compiler test suite so you can verify that it works correctly.