Compiler Nots

This compilation highlights key points from each chapter that I deem essential for memorization. Following each chapter, you'll find solutions to the challenges presented, providing a comprehensive understanding of the material.

`Status` `Processing`

References

- Crafting Interpreters book
- Crafting Interpreters Repo

Table of Contents

Chapter 1 Introduction

NOTES

1. little languages = domain-specific languages : These are pidgins tailor-built to a specific task.
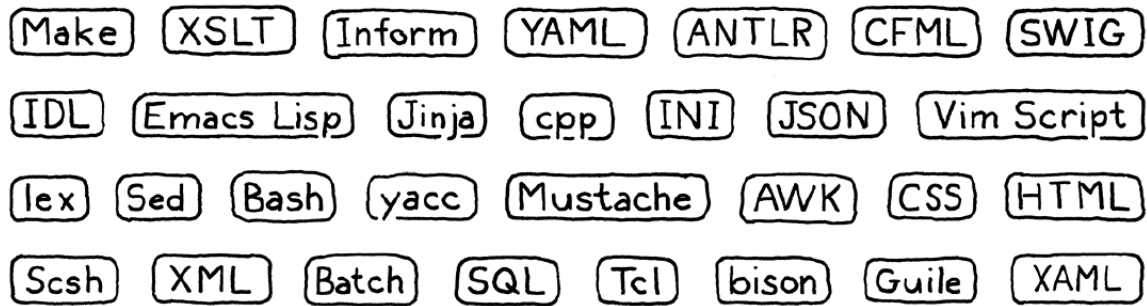
Figure 1: domain-specific languages

2. A compiler reads files in one language. translates them, and outputs files in another language. You can implement a compiler in any language, including the same language it compiles, a process called *self-hosting*.

3. You can't compile your compiler using itself yet, but if you have another compiler for your language written in some other language, you use that one to compile your compiler once. Now you can use the compiled version of your own compiler to compile future versions of itself and you can discard the original one compiled from the other compiler. This is called *bootstrapping* from the image of pulling yourself up by your own bootstraps.

4. Bytecode is an intermediate representation (IR) of a program's source code that is generated by a compiler or an interpreter before it is executed. It is a low-level, platform-independent set of instructions that can be executed by a virtual machine (VM) or interpreter. Bytecode is often used to bridge the gap between high-level programming languages and machine code.

CHALLENGES

1. There are at least six domain-specific languages used in the little system I cobbled together to write and publish this book. What are they?

   HTML , CSS , SQL , JSON , XML , XAML , Bash

2. Get a "Hello, world!" program written and running in Java. Set up whatever Makefiles or IDE projects you need to get it working. If you have a debugger, get comfortable with it and step through your program as it runs.

   Chapter1.java

```
1  public class Chapter1 {
2    public static void main(String[] args) {
3        System.out.println("Hello, World!");
```

```
4    }
5  }
```

Compile the program with the following command:

```
1  javac Chapter1.java
```

Run the compiled program with:

```
1   java Chapter1
```

3. Do the same thing for C. To get some practice with pointers, define a doubly-linked list of heap-allocated strings. Write functions to insert, find, and delete items from it. Test them.

doubly-linked-list.c

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct Node {
6      char* data;
7      struct Node* prev;
8      struct Node* next;
9  } Node;
10
11 typedef struct {
12     Node* head;
13     Node* tail;
14     int size;
15 } LinkedList;
16
17 void initList (LinkedList *list){
18     list->head = NULL;
19     list->tail = NULL;
20     list->size = 0;
21 }
22
23 void insert(LinkedList *list , const char* data) {
24     Node *newNode = (Node*) malloc(sizeof(Node));
25     newNode->data = strdup(data);
26     newNode->prev = list->tail; // list->tail to add the new node
           at end
27     newNode->next = NULL; // now the new node at the end of the
           list
28     if(list->tail != NULL) {
29         list->tail->next = newNode;
30     }
```

```c
31      list->tail = newNode;
32      if(list->head == NULL) {
33          list->head = newNode;
34      }
35      list->size++;
36  }
37
38  Node *find (LinkedList *list , const char* data) {
39      Node *cuur = list->head;
40      while (cuur != NULL) {
41          if (strcmp(cuur->data, data) == 0) {
42              return cuur;
43          }
44          cuur = cuur->next;
45      }
46      return NULL;
47  }
48
49  void deleteNode(LinkedList *list, Node *node) {
50      if (node == NULL) {
51          return;
52      }
53      if(node->prev != NULL) {
54          node->prev->next = node->next;
55      } else {
56          list->head = node->next;
57      }
58      if(node->next != NULL) {
59          node->next->prev = node->prev;
60      } else {
61          list->tail = node->prev;
62      }
63
64      free(node->data);
65      free(node);
66      list->size--;
67  }
68
69  void freeList (LinkedList *list) {
70      Node *curr = list->head;
71      while (curr != NULL) {
72          Node *next = curr->next;
73          free(curr->data);
74          free(curr);
75          curr = next;
76      }
77      list->head = NULL;
78      list->tail = NULL;
79      list->size = 0;
80  }
81
```

```c
82  int getSize (LinkedList *list) {
83      return list->size;
84  }
85
86  void printList (LinkedList *list) {
87      Node *cuur = list->head;
88      while (cuur != NULL) {
89          printf("%s ",cuur->data);
90          cuur = cuur->next;
91      }
92      printf("\n");
93  }
94
95  int main () {
96      LinkedList list;
97      initList(&list);
98      insert(&list, "Hello");
99      insert(&list, "Hussein");
100     insert(&list, "Hussein2");
101
102     printf("Original List: ");
103     printList(&list);
104
105     int size = getSize(&list);
106     printf("Size: %d\n" , size);
107
108     Node *foundNode = find(&list, "Hussein2");
109     if (foundNode != NULL) {
110         printf("Found: %s\n", foundNode->data);
111         deleteNode(&list, foundNode);
112         printf("After Deletion: ");
113         printList(&list);
114     } else {
115         printf("Not Found.\n");
116     }
117
118     freeList(&list);
119
120     return 0;
121  }
```

Compile the program with the following command:

```
1  gcc doubly-linked-list.c
```

Run the compiled program with:

```
1  .\doubly-linked-list
```

Chapter 2 A Map of the Territory

NOTES

1. This book is about a language's implementation, which is distinct from the language itself in some sort of Platonic ideal form. Things like *stack*, *bytecode*, and *recursive descent*, are nuts and bolts one particular implementation might use.
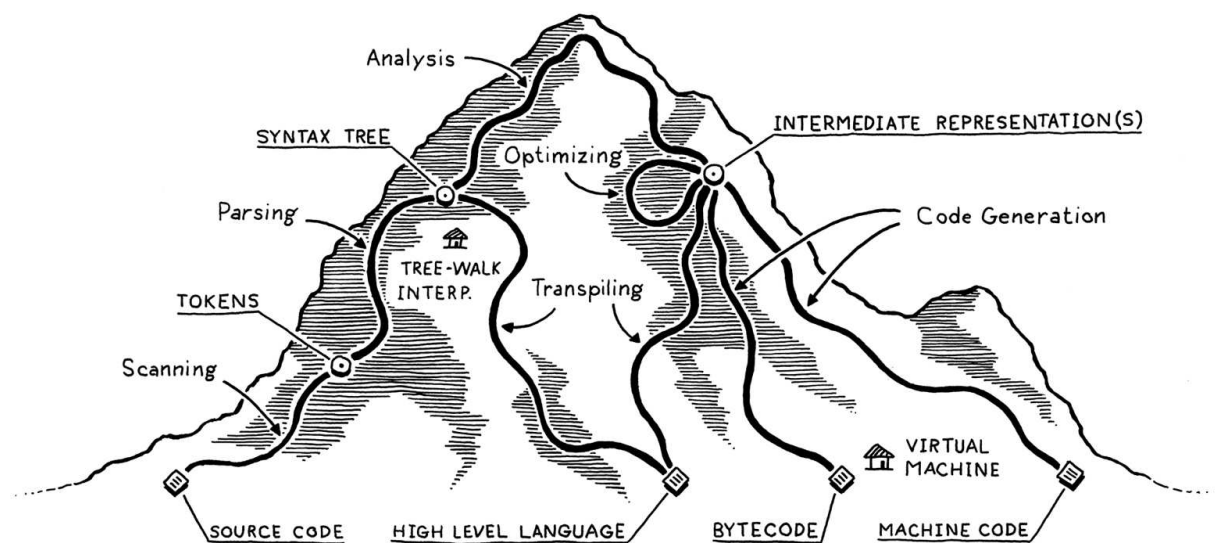

Figure 2: overall view

2. A scanner (or lexer) takes in the linear stream of characters and chunks them together into a series of something more akin to *words*.

3. A parser takes the flat sequence of tokens and builds a tree structure that mirrors the nested nature of the grammar. These trees have a couple of different names—*parse tree* or *abstract syntax tree* —depending on how close to the bare syntactic structure of the source language they are. In practice, language hackers usually call them *syntax trees*, *ASTs*, or often just *trees*. The parser's job also includes letting us know when we do by reporting *syntax errors*.

4. The language we'll build in this book (lox) is dynamically typed, so it will do its type checking later, at runtime.

5. The front end of the pipeline is specific to the source language the program is written in. The back end is concerned with the final architecture where the program will run.

6. Intermediate representation lets you support multiple source languages and target platforms with less effort. Say you want to implement *Pascal*, *C* and *Fortran* compilers and you want to target *x86*, *ARM*, and, I dunno, *SPARC*. Normally, that means you're signing up to write **nine full compilers**:

Pascal→x86, C→ARM, and every other combination. A shared intermediate representation reduces that dramatically. You write one front end for each source language that produces the IR. Then one back end for each target architecture. Now you can mix and match those to get every combination.

- Source Languages: C, Fortran, Pascal
- Target Architectures: x86, ARM, SPARC

Without Intermediate Representation (IR)

1. C

  – x86
  – ARM
  – SPARC

2. Fortran

  – x86
  – ARM
  – SPARC

3. Pascal

  – x86
  – ARM
  – SPARC

> 9 full compilers

With Intermediate Representation (IR)

Front-end

1. C

  – IR

2. Fortran

  – IR

3. Pascal

  – IR

Back-end

4. IR

   – x86
   – ARM
   – SPARC

   > 3 full compilers

7. If we generate real *machine code*, we get an executable that the OS can load directly onto the chip. *Native code* is lightning fast, but generating it is a lot of work. Today's architectures have piles of instructions, complex pipelines, and enough historical baggage to fill a 747's luggage bay.

8. **Execution Environments:**

   - **JVM:**
     – Executes bytecode.
   - **.Net:**
     – Uses CLR (Common Language Runtime).
     – Executes IL (Microsoft Intermediate Language).

9. **Bytecode Compilation Options:**

   If your compiler produces bytecode, you have two primary options:

   - **Option 1: Native Code Compilation**

     – Write a mini-compiler for each target architecture.
     – Convert the bytecode to native code for the specific machine.

   - **Option 2: Virtual Machine (VM) Execution**

     – Write a virtual machine (VM) program.
     – The VM emulates a hypothetical chip supporting your virtual architecture at runtime.
     – Running bytecode in a VM is slower than translating it to native code ahead of time.
     – Offers simplicity and portability.
     – Implement the VM in a language like C, allowing the language to run on any platform with a C compiler.
     – This approach is employed by the second interpreter developed in this book.

10. If the language is run inside an interpreter or VM, then the runtime lives there. This is how most implementations of languages like Java, Python, and JavaScript work.

11. *Syntax-directed translation* is a structured technique for building these all-at-once compilers. You associate an action with each piece of the grammar, usually one that generates output code. Then, whenever the parser matches that chunk of syntax, it executes the action, building up the target code one rule at a time.

12. The fastest way to execute code is by compiling it to machine code, but you might not know what architecture your end user's machine supports. What to do? > You can do the same thing that the HotSpot *JVM*, *Microsoft's CLR* and most *JavaScript interpreters* do. On the end user's machine, when the program is loaded—either from source in the case of JS, or platform-independent bytecode for the JVM and CLR—you compile it to *native* for the architecture their computer supports. Naturally enough, this is called *just-in-time compilation*. Most hackers just say "JIT".

13. Compilers and Interpreters

    • *Compiling* is an implementation technique that involves translating a source language to some other—usually lower-level—form. When you generate bytecode or machine code, you are compiling. When you transpile to another high-level language you are compiling too.
    • When we say a language implementation *is a compiler*, we mean it translates source code to some other form but doesn't execute it.
    • when we say an implementation "is an interpreter", we mean it takes in source code and executes it immediately. it runs programs "from source".

14. CPython is an interpreter, and it has a compiler

CHALLENGES

1. Pick an open source implementation of a language you like. Download the source code and poke around in it. Try to find the code that implements the scanner and parser. Are they hand-written, or generated using tools like Lex and Yacc? (.l or .y files usually imply the latter.)

2. Just-in-time compilation tends to be the fastest way to implement a dynamically typed language, but not all of them use it. What reasons are there to not JIT?

    Complexity, Portability, and Compilation Overhead.

3. Most Lisp implementations that compile to C also contain an interpreter that lets them execute Lisp code on the fly as well. Why?

    to provide a more interactive and flexible development environment

Chapter 3 The Lox Language

NOTES

1. Lox is dynamically typed.

2. Automatic memory management

3. There are two main techniques for managing memory: reference counting and tracing garbage collection (usually just called "garbage collection" or "GC").

4. Data Types:

Booleans

```
1  true; // Not false.
2  false; // Not *not* false.
```

Numbers: Lox only has one kind of number: double-precision floating point.

```
1  1234; // An integer.
2  12.34; // A decimal number.
3
4  .2; // not allowed in lox
5  2.; // not allowed in lox
```

Strings

```
1  "I am a string";
2  ""; // The empty string.
3  "123"; // This is a string, not a number.
```

Nil

```
1  return nil; // similar to returning null in other languages
```

5. Expressions:

Arithmetic

```
1  add + me;
2  subtract - me;
3  multiply * me;
4  divide / me;
5
6  -negateMe;
```

```
1  less < than;
2  lessThan <= orEqual;
3  greater > than;
4  greaterThan >= orEqual;
5
6  1 == 2; // false.
7  "cat" != "dog"; // true.
8
9  314 == "pi"; // false.
10
11 123 == "123"; // false.
```

- look at this function in the interpreter

```
1     // the 0 in lox is true not false, if U want it be false edit the
          function below
2     // < check-operands
3     // > is-truthy
4     private boolean isTruthy(Object object) {
5       if (object == null)
6         return false;
7       if (object instanceof Boolean)
8         return (boolean) object;
9       return true;
10    }
```

Logical operators : The reason and and or are like control flow structures is because they short circuit. Not only does and return the left operand if it is false, it doesn't even evaluate the right one in that case.

```
1  !true; // false.
2  !false; // true.
3
4  true and false; // false.
5  true and true; // true.
6
7  false or false; // false.
8  true or false; // true.
```

Precedence and grouping

```
1  var average = (min + max) / 2;
```

6. Statements

```
1  print "Hello, world!";
2
3  "some expression";
4
5  {
6  print "One statement.";
7  print "Two statements.";
8  }
```

7. Variables

```
1  var imAVariable = "here is my value";
2  var iAmNil;
3
4  var breakfast = "bagels";
5  print breakfast; // "bagels".
6  breakfast = "beignets";
7  print breakfast; // "beignets".
```

8. Control Flow

```
1  if (condition) {
2  print "yes";
3  } else {
4  print "no";
5  }
6
7  var a = 1;
8  while (a < 10) {
9  print a;
10  a = a + 1;
11  }
12
13  for (var a = 1; a < 10; a = a + 1) {
14  print a;
15  }
```

9. Functions

- An *argument* is an actual value you pass to a function when you call it.

- A *parameter* is a variable that holds the value of the argument inside the body of the function.

```
1  makeBreakfast(bacon, eggs, toast);
2
3  makeBreakfast();
4
5  // a and b called parameters
6  fun printSum(a, b) {
```

```
7        print a + b;
8    }
9
10   // 1 and 2 called arguments
11   printSum(1,2);
```

Closures

```
1    fun addPair(a, b) {
2    return a + b;
3    }
4
5    fun identity(a) {
6    return a;
7    }
8    print identity(addPair)(1, 2); // Prints "3".
9
10   fun outerFunction() {
11       fun localFunction() {
12           print "I'm local!";
13       }
14       localFunction();
15   }
16
17   fun returnFunction() {
18       var outside = "outside";
19       fun inner() {
20       print outside;
21       }
22       return inner;
23   }
24   var fn = returnFunction();
25   fn();
```

10. Classes

```
1    class Breakfast {
2        // var x = 5; // this is not allowed in lox
3        cook() {
4            print "Eggs a-fryin'!";
5        }
6        serve(who) {
7            print "Enjoy your breakfast, " + who + ".";
8        }
9    }
10
11   // Store it in variables.
12   var someVariable = Breakfast;
13   // Pass it to functions.
14   someFunction(Breakfast);
```

```
15
16  var breakfast = Breakfast();
17  print breakfast; // "Breakfast instance".
```

Inheritance: using a less-than (<) operator

```
1  class Brunch < Breakfast {
2      drink() {
3          print "How about a Bloody Mary?";
4      }
5  }
6
7  var benedict = Brunch("ham", "English muffin");
8  benedict.serve("Noble Reader");
9
10  class Brunch < Breakfast {
11      init(meat, bread, drink) {
12          super.init(meat, bread);
13          this.drink = drink;
14      }
15  }
```

The Standard Library : built-in function clock() that returns the number of seconds since the program started.

CHALLENGES

1. Write some sample Lox programs and run them (you can use the implementations of Lox in my repository). Try to come up with edge case behavior I didn't specify here. Does it do what you expect? Why or why not?

```
1   /*
2   fun makeCounter() {
3     var i = 0;
4     fun count() {
5       i = i + 1;
6       print i;
7     }
8
9     return count;
10  }
11
12  //fun scope(a) {
13  //  print a; // parameter
14  //  var a = "local";
15  //  print a; // local
16  //}
```

```
17
18  fun thrice(fn) {
19    for (var i = 1; i <= 3; i = i + 1) {
20      fn(i);
21    }
22  }
23
24  thrice(fun (a) {
25    print a;
26  });
27  // "1".
28  // "2".
29  // "3".
30
31  var counter = makeCounter();
32  //counter(); // "1".
33  //counter(); // "2".
34
35  //scope("parameter");
36
37  fun scope(a) {
38    print a;
39    var a = "local";
40    print a;
41  }
42
43  print 10;
44  scope(5);
45
46  /*
47
48  /*
49  var a = "global";
50  {
51    fun showA() {
52      print a;
53    }
54
55    showA();
56    var a = "block";
57    showA();
58    {
59      var a = "block2";
60      showA();
61    }
62    showA();
63  }
64  */
65
66  /*
67  var a = 5;
```

```
 68
 69 {
 70   print a;
 71   var a = a;
 72   print a;
 73   a = 6;
 74   print a;
 75 }
 76 */
 77
 78
 79
 80 /*
 81 var a = 5;
 82 var a = 6;
 83 print a;
 84 */
 85
 86
 87
 88 /*
 89 fun bad() {
 90 var a = "first";
 91 var a = "second";
 92 print a;
 93 }
 94
 95 bad();
 96 */
 97
 98 //break;
 99
100 /*
101 while (true) {
102   if (5 > 0 ) {
103     print 6;
104     break;
105   }
106   var a = 6;
107   print a;
108 }
109 var a = 5;
110 print a;
111 */
112
113 /*
114 while (true) {
115   var a = 6;
116   print a;
117
118   if (5 > 0) {
```

```
119     break;
120   }
121 }
122
123 var a = 5;
124 print a;
125 */
126
127 /*
128 if (a > 1) {
129   print a;
130   break;
131 }
132 */
133
134 /*
135 class DevonshireCream {
136   serveOn() {
137     return "Scones";
138   }
139 }
140 print DevonshireCream; // Prints "DevonshireCream".
141
142 class Bagel {}
143 var bagel = Bagel();
144 print bagel; // Prints "Bagel instance".
145
146 class Bacon {
147   eat() {
148     print "Crunch crunch crunch!";
149   }
150 }
151
152 Bacon().eat(); // Prints "Crunch crunch crunch!".
153 */
154
155 /*
156 class Doughnut {
157   cook() {
158     print "Fry until golden brown.";
159   }
160 }
161 class BostonCream < Doughnut {
162   cook() {
163     var method = super.cook;
164     method();
165     super.cook();
166     print "Pipe full of custard and coat with chocolate.";
167   }
168 }
169 BostonCream().cook();
```

```
170  */
171
172  /*
173  class A {
174    method() {
175    print "Method A";
176    }
177  }
178  class B < A {
179    method() {
180      print "Method B";
181    }
182    test() {
183      super.method();
184    }
185  }
186  class C < B {}
187
188  C().test();
189  */
190
191  /*
192  fun fib(n) {
193    if (n < 2) return n;
194    return fib(n - 1) + fib(n - 2);
195  }
196
197  var before = clock();
198  print fib(40);
199  var after = clock();
200  print after - before;
201  */
202
203  while (true) {
204    if (5 > 0 ) {
205      print 6;
206      break;
207    }
208  }
```

2. This informal introduction leaves a lot unspecified. List several open questions you have about the language's syntax and semantics. What do you think the answers should be?

3. Lox is a pretty tiny language. What features do you think it is missing that would make it annoying to use for real programs? (Aside from the standard library, of course.)

Chapter 4 Scanning

NOTES

1. The scanner takes in raw source code as a series of characters and groups it into a series of chunks we call tokens.

2. Lox is a scripting -high level- language, which means it executes directly from source.

3. Each of these blobs of characters is called a lexeme:

Figure 3: blobs of characters (lexeme)

4. Note that the ! and = are not two independent operators. You can't write ! = in Lox and have it behave like an inequality operator.

5. We've got another helper:

```
1  private char peek() {
2    if (isAtEnd()) return '\0';
3    return source.charAt(current);
4  }
```

> It's sort of like advance(), but doesn't consume the character. This is called *lookahead*. Since it only looks at the current unconsumed character, we have one character of lookahead. ***The smaller this number is, generally, the faster the scanner runs.*** The rules of the lexical grammar dictate how much lookahead we need. Fortunately, most languages in wide use only peek one or two characters ahead.

Mid Qs : What is the lookahead of the scanner ? is it better to have larger or smaller lookaheads ? What is the lookahead of Lox ?

- Lookahead in Scanners:

Meaning: The lookahead of a scanner refers to the number of characters it can examine ahead of its current position without actually consuming them. It aids in making informed decisions about token formation.

- Optimal Size:

Smaller lookaheads (1-2 characters) are generally preferred for efficiency. Larger lookaheads might be necessary for certain language constructs but can impact speed.

- Lox's Lookahead

Lox's scanner uses a lookahead of 1 character. This is sufficient for its lexical grammar, as it doesn't have complex constructs that require extensive lookahead.

6. Since we only look for a digit to start a number, that means -123 is not a number literal. Instead, -123, is an expression that applies - to the number literal 123.

```
1  print -123.abs();
```

This prints -123 because negation has lower precedence than method calls. We could fix that by making - part of the number literal. But then consider:

```
1  var n = 123;
2  print - n.abs();
```

This still produces -123, so now the language seems inconsistent. No matter what you do, some case ends up weird.

7. We don't allow a leading or trailing decimal point, so these are both invalid:

```
1  .1234
2  1234.
```

8. We don't allow this :

```
1  123.sqrt();
```

9. Consider this in lox:

```
1  var a = 5;
2
3  print -a; // -5
4  print --a; // 5
5  print ---a; // -5
```

10. `java case 'o': if (peek()== 'r'){ addToken(OR); } break;`

Consider what would happen if a user named a variable *orchid*. The scanner would see the first two letters, or, and immediately emit an or keyword token. This gets us to an important principle called **maximal munch**. *When two lexical grammar rules can both match a chunk of code that the scanner is looking at, whichever one matches the most characters wins.*

11. Maximal munch means we can't easily detect a reserved word until we've reached the end of what might instead be an identifier.

CHALLENGES

1. The lexical grammars of Python and Haskell are not regular. What does that mean, and why aren't they?

   > A lexical grammar defines the basic building blocks of a programming language, such as tokens (keywords, identifiers, literals, etc.), and specifies how these tokens are combined to form valid programs. The reason why the lexical grammars of Python and Haskell are not regular can be attributed to the expressive power and flexibility that these languages provide.

2. Aside from separating tokens—distinguishing print foo from printfoo — spaces aren't used for much in most languages. However, in a couple of dark corners, a space does affect how code is parsed in CoffeeScript, Ruby, and the C preprocessor. Where and what effect does it have in each of those languages?

3. Our scanner here, like most, discards comments and whitespace since those aren't needed by the parser. Why might you want to write a scanner that does not discard those? What would it be useful for?

4. Add support to Lox's scanner for C-style /*….. */ block comments. Make sure to handle newlines in them. Consider allowing them to nest. Is adding support for nesting more work than you expected? Why?

- Scanner.java

```java
case '/':
            if (match('/')) {
                // A comment goes until the end of the line.
                while (peek() != '\n' && !isAtEnd())
                    advance();
            } else if (match('*')) {
                MultilineComment();
            } else {
                addToken(SLASH);
            }
            break;
```

```java
    private void MultilineComment() {

        int nestLevel = 1;

        while (true) {
            if (isAtEnd()) {
                Lox.error(line, "Unterminated multiline comment.");
                return;
```

```
 9                      }
10
11                  if (peek() == '/' && peekNext() == '*') {
12                      // Consume the '*' and '/' characters.
13                      advance();
14                      advance();
15                      nestLevel++;
16                      return;
17                  } else if (peek() == '*' && peekNext() == '/') {
18                      advance();
19                      advance();
20                      nestLevel--;
21
22                      if (nestLevel == 0) {
23                          return;
24
25                      }
26                  } else if (peek() == '\n') {
27                      line++;
28                  }
29
30                  advance();
31              }
32          }
```

Chapter 5 Representing Code

NOTS

1. If you start with the rules, you can use them to generate strings that are in the grammar. Strings created this way are called derivations because each is "derived" from the rules of the grammar.

2. Rules are called productions because they produce strings in the grammar.

3. A terminal is a letter from the grammar's alphabet.

4. A nonterminal is a named reference to another rule in the grammar.

```
expression      → literal
                | unary
                | binary
                | grouping ;

literal         → NUMBER | STRING | "true" | "false" | "nil" ;
grouping        → "(" expression ")" ;
unary           → ( "-" | "!" ) expression ;
binary          → expression operator expression ;
operator        → "==" | "!=" | "<" | "<=" | ">" | ">="
                | "+"  | "-"  | "*" | "/" ;
```

Figure 4: grammar