# 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.

Chapter 1 : Introduction

NOTES

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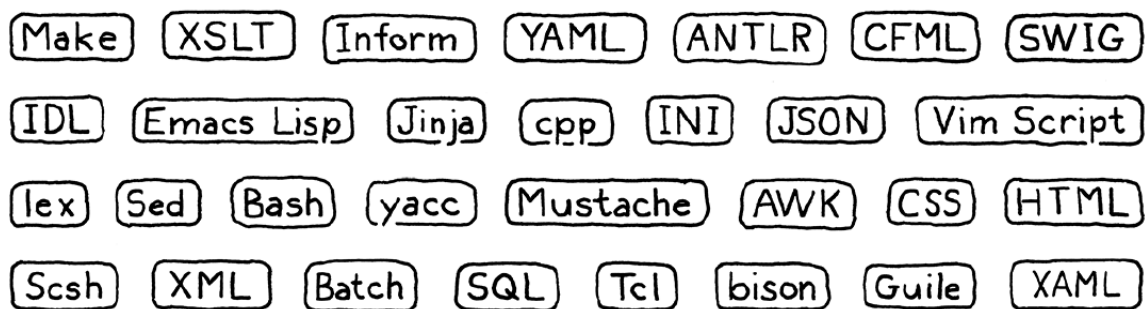
Figure 1: domain-specific languages

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

   ```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;
   ```

```c
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      }
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      }
```

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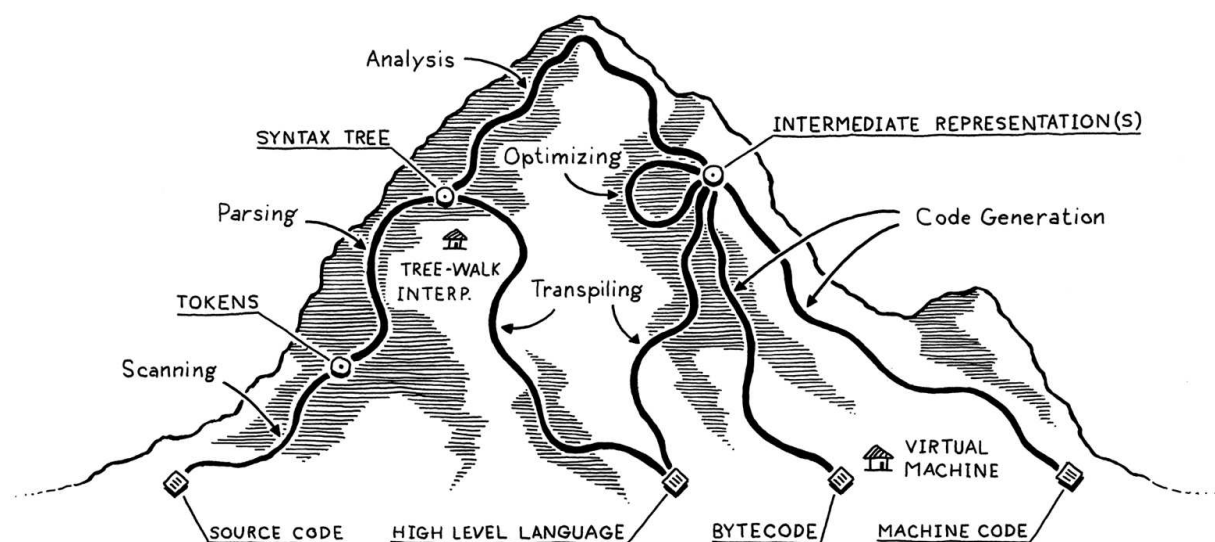
```
1  gcc doubly-linked-list.c
```

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

With Intermediate Representation (IR)

Front-end

1. C

– IR

2. Fortran

– IR

3. Pascal

– IR

Back-end

4. IR

– x86
– ARM
– SPARC

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

CHALLENGES