



Durian or: How I Learned To Stop Worrying And Love Programming Languages

How do we get from:

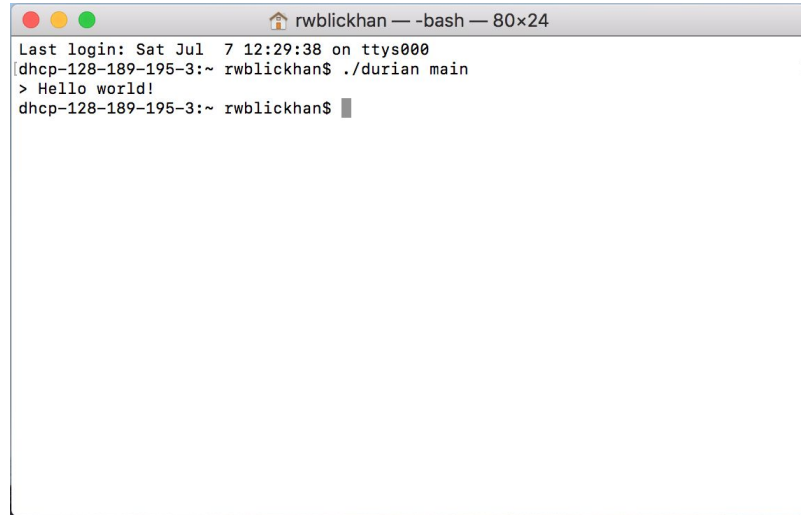
to:



The screenshot shows an nvim editor window titled "rwblickhan — nvim main.dur — 80x24". The editor contains a C program with four lines of code: `1 main() {`, `2 print "Hello World!"`, `3 }`, and `4` followed by a cursor. The left margin shows line numbers 1 through 14. At the bottom, a status bar displays "main.dur", "4,0-1", and "All". A message bar at the very bottom says "\"main.dur\" 4L, 37C written".

```
1 main() {
2     print "Hello World!"
3 }
4
```

main.dur 4,0-1 All
"main.dur" 4L, 37C written



The screenshot shows a terminal window titled "rwblickhan — -bash — 80x24". It displays the login process, the execution of the program, and the output. The text in the terminal is: `Last login: Sat Jul 7 12:29:38 on ttys000`, `[dhcp-128-189-195-3:~ rwblickhan$./durian main]`, `> Hello world!`, and `dhcp-128-189-195-3:~ rwblickhan$` followed by a cursor.

```
Last login: Sat Jul 7 12:29:38 on ttys000
[dhcp-128-189-195-3:~ rwblickhan$ ./durian main]
> Hello world!
dhcp-128-189-195-3:~ rwblickhan$
```

Machine code

- Processors can only run what's called "machine code"

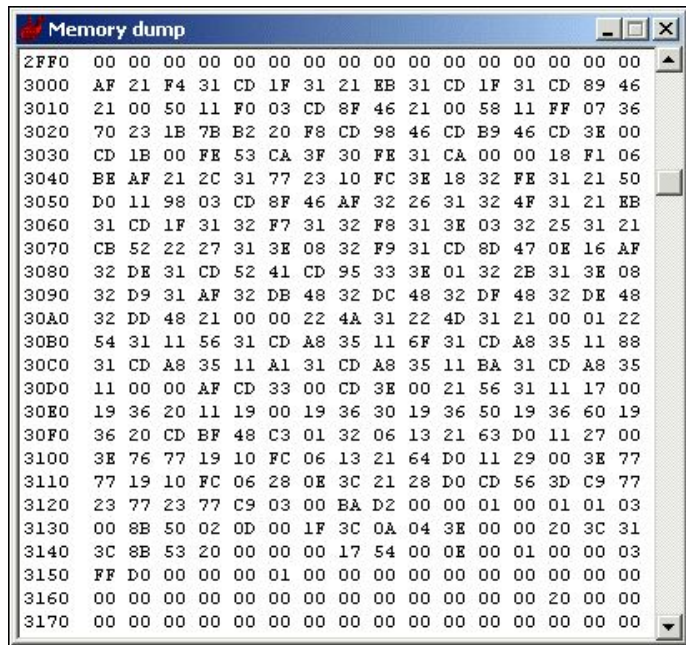
[illegible]

Bytecode

- Analogous to machine code, but there is no real life chip that is capable of executing it

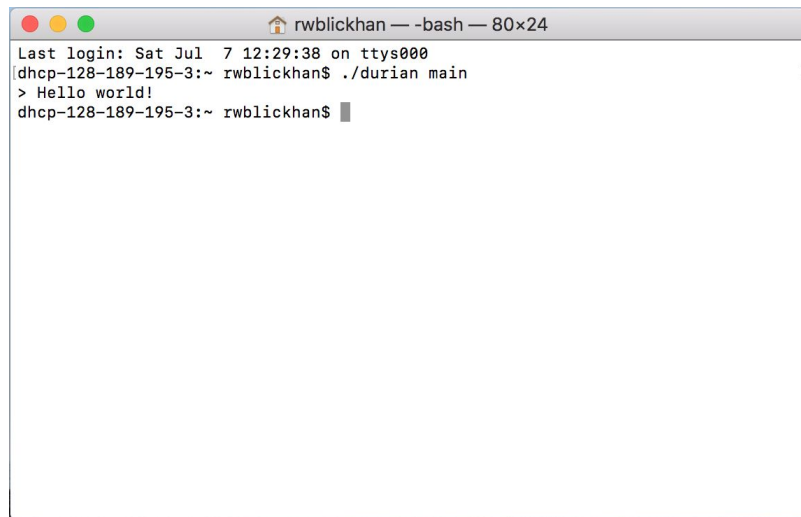
Virtual Machine

- Software to execute bytecode, analogous to how a real computer executes machine code



A screenshot of a 'Memory dump' window. The title bar is blue with a red icon and the text 'Memory dump'. The window contains a list of memory addresses and their corresponding hexadecimal values. The addresses range from 2FF0 to 3170 in increments of 10. The values are displayed in two columns, with the first column showing the address and the second column showing the hex data. The data appears to be a sequence of bytes, some of which are zero.

Address	Hex Data
2FF0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
3000	AF 21 F4 31 CD 1F 31 21 EB 31 CD 1F 31 CD 89 46
3010	21 00 50 11 F0 03 CD 8F 46 21 00 58 11 FF 07 36
3020	70 23 1B 7B B2 20 F8 CD 98 46 CD B9 46 CD 3E 00
3030	CD 1B 00 FE 53 CA 3F 30 FE 31 CA 00 00 18 F1 06
3040	BE AF 21 2C 31 77 23 10 FC 3E 18 32 FE 31 21 50
3050	D0 11 98 03 CD 8F 46 AF 32 26 31 32 4F 31 21 EB
3060	31 CD 1F 31 32 F7 31 32 F8 31 3E 03 32 25 31 21
3070	CB 52 22 27 31 3E 08 32 F9 31 CD 8D 47 0E 16 AF
3080	32 DE 31 CD 52 41 CD 95 33 3E 01 32 2B 31 3E 08
3090	32 D9 31 AF 32 DB 48 32 DC 48 32 DF 48 32 DE 48
30A0	32 DD 48 21 00 00 22 4A 31 22 4D 31 21 00 01 22
30B0	54 31 11 56 31 CD A8 35 11 6F 31 CD A8 35 11 88
30C0	31 CD A8 35 11 A1 31 CD A8 35 11 BA 31 CD A8 35
30D0	11 00 00 AF CD 33 00 CD 3E 00 21 56 31 11 17 00
30E0	19 36 20 11 19 00 19 36 30 19 36 50 19 36 60 19
30F0	36 20 CD BF 48 C3 01 32 06 13 21 63 D0 11 27 00
3100	3E 76 77 19 10 FC 06 13 21 64 D0 11 29 00 3E 77
3110	77 19 10 FC 06 28 0E 3C 21 28 D0 CD 56 3D C9 77
3120	23 77 23 77 C9 03 00 BA D2 00 00 01 00 01 01 03
3130	00 8B 50 02 0D 00 1F 3C 0A 04 3E 00 00 20 3C 31
3140	3C 8B 53 20 00 00 00 17 54 00 0E 00 01 00 00 03
3150	FF D0 00 00 00 01 00 00 00 00 00 00 00 00 00 00
3160	00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00
3170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00




A screenshot of a terminal window. The title bar is grey with a home icon and the text 'rwblinkhan -- bash -- 80x24'. The terminal shows the following text: 'Last login: Sat Jul 7 12:29:38 on ttys000', '[dhcp-128-189-195-3:~ rwblinkhan\$./durian main]', '> Hello world!', and 'dhcp-128-189-195-3:~ rwblinkhan\$'. The prompt is a dollar sign.

Interpreting bytecode

```
Terminal — python — 80x24
Last login: Sat Jul 21 11:25:43 on ttys001
tom:~$ python
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> import dis
>>> def super_fancy_fn():
...     print("Hello World!")
...
[>>> dis.dis(super_fancy_fn)
2          0 LOAD_GLOBAL              0 (print)
          2 LOAD_CONST                1 ('Hello World!')
          4 CALL_FUNCTION             1
          6 POP_TOP
          8 LOAD_CONST                0 (None)
         10 RETURN_VALUE

>>> █
```


So how do we get from source code to bytecode?



```
1 main() {
2     print "Hello World!"
3 }
4
```

main.dur 4L, 37C written

[illegible]

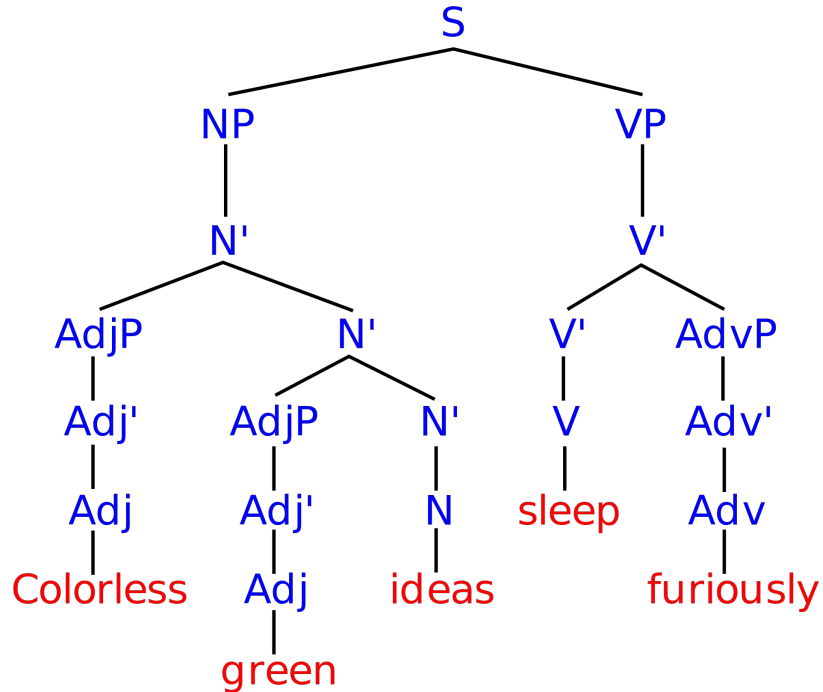
Take a byte...

- Bytecode is like machine code, except for our virtual machine
- Generate bytecode based on the structure of the tree by walking through the AST

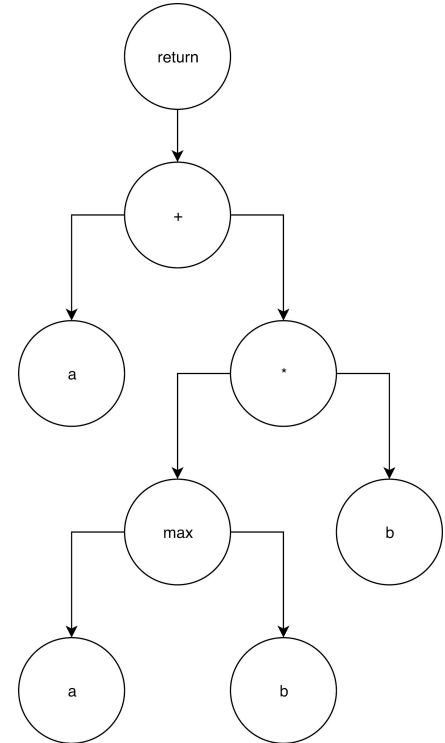


Concrete syntax vs abstract syntax vs semantics

Colourless green ideas sleep furiously.



return a + max(a,b) * b



Strings to Syntax

- **Lexer/scanner**
 - Convert source code into stream of *tokens*
 - Tokens: keywords (def, while), operators (+, *), identifiers (a, max), punctuation, ...
- **Parser**
 - Converts stream of tokens into *abstract syntax trees (ASTs)*

```
if ( x > 0 ) printf("the sum is %d\n", x + 7);
```

```
if( [x] > [0] ) printf("the sum is %d\n", [x] + [7]);
```

♪ What do you do with a drunken syntax ♪

- Pretty printing (okay, that's not that useful...)
- Typechecking (not in Durian!)
- Optimization
- Treewalk interpreter (slow!)
- Bytecode

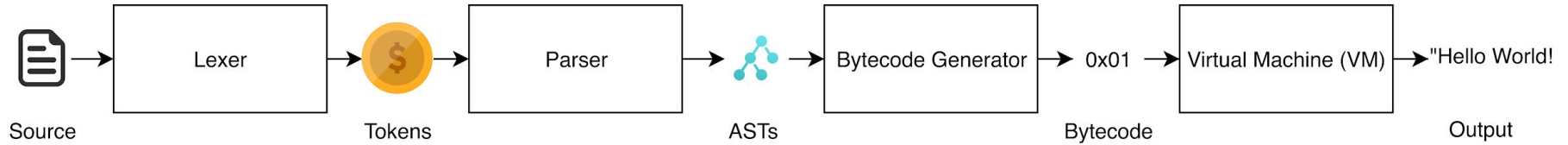
How?

- Interpreter pattern!
- Visitor pattern!
- Pattern matching (functional programming)!

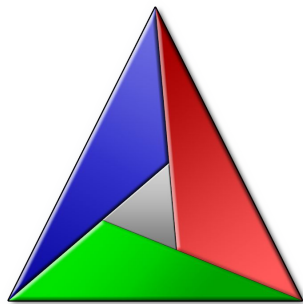
Optimisations

- Lua skips the *AST* step entirely, generating bytecode directly from the tokens.
 - This significantly complicates the implementation, but helps make the Lua interpreter around 100 kB in size.
- Python, and most other languages, have a step between parsing and generating bytecode where they optimise by “folding constants”, “interning strings”, and removing dead code.
- *Just-in-time (JIT) compilation*: generate bytecode, then optimize “hot” code while running

Compilation Pipeline



Tech Stack



CMake