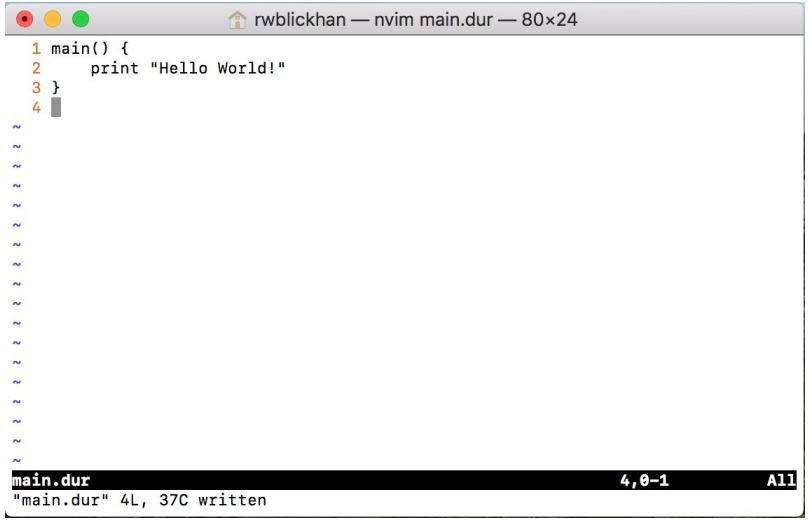
*

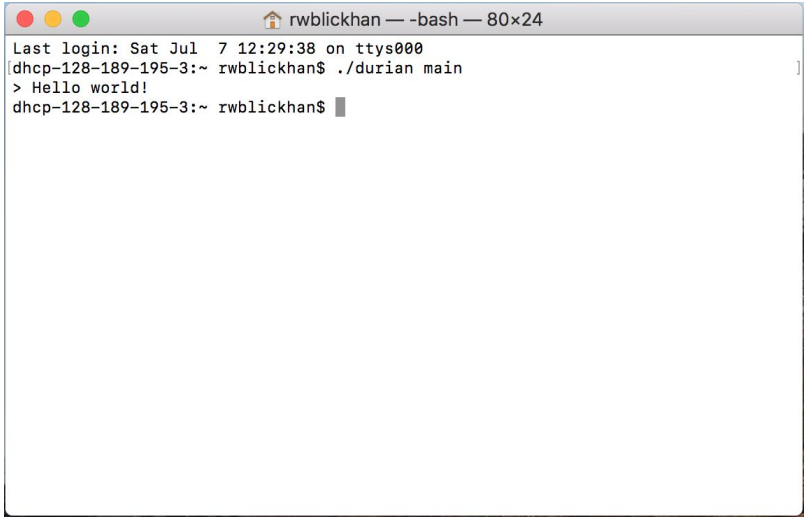
Durian or: How I Learned To Stop Worrying And Love Programming Languages

*Yes, I know that's a pineapple… there is no durian emoji

How do we get from:

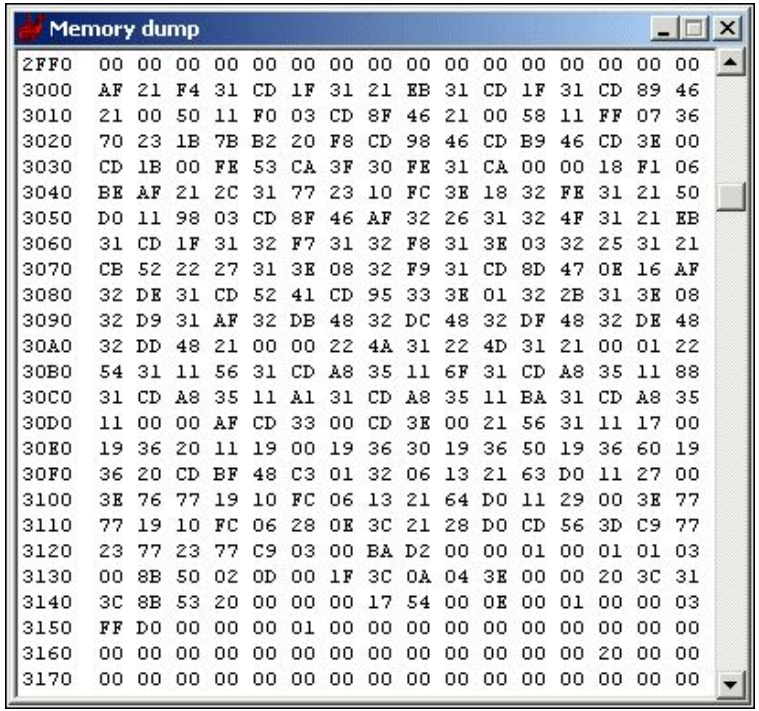


to:



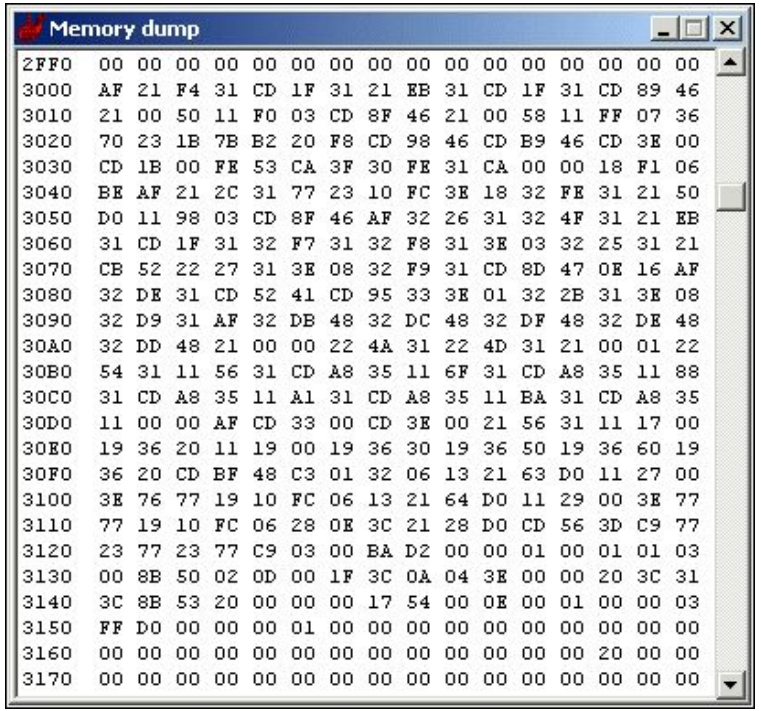
Machine code

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



Compiling

- Translating a higher level language to machine code

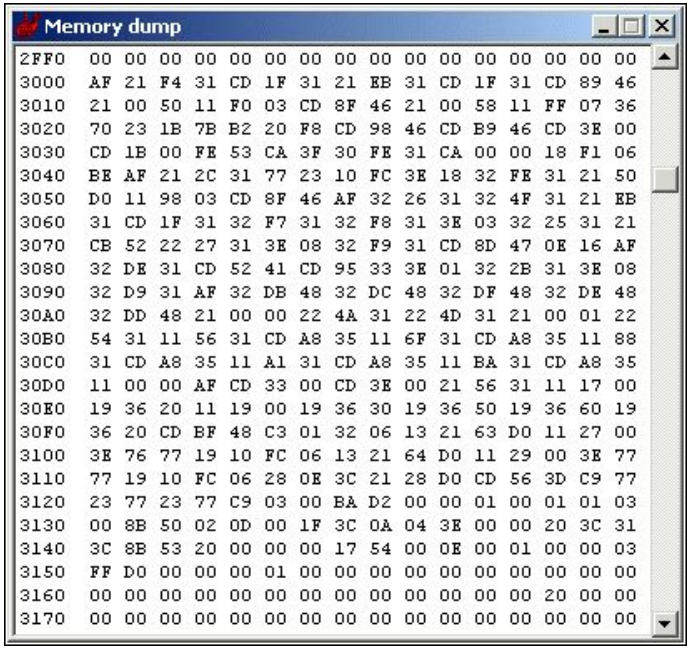


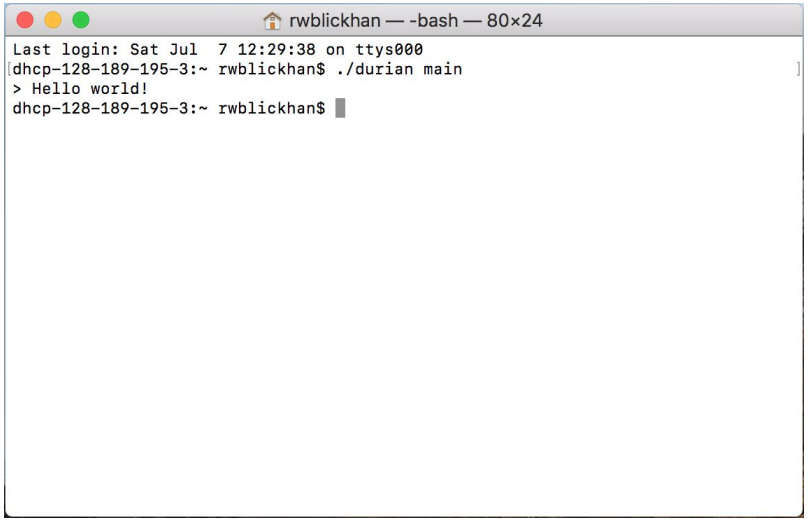
Bytecode

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

Virtual Machine

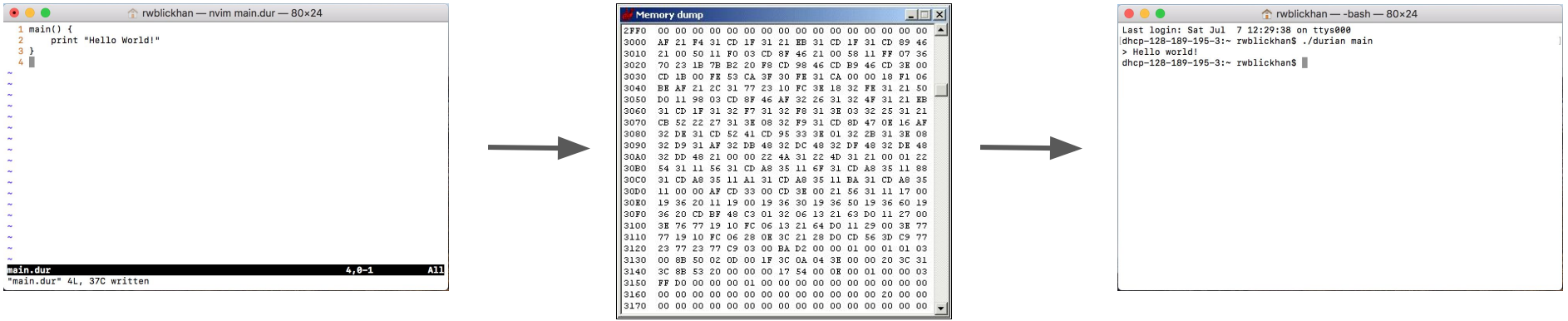
- Software to execute bytecode, analogous to how a real computer executes machine code



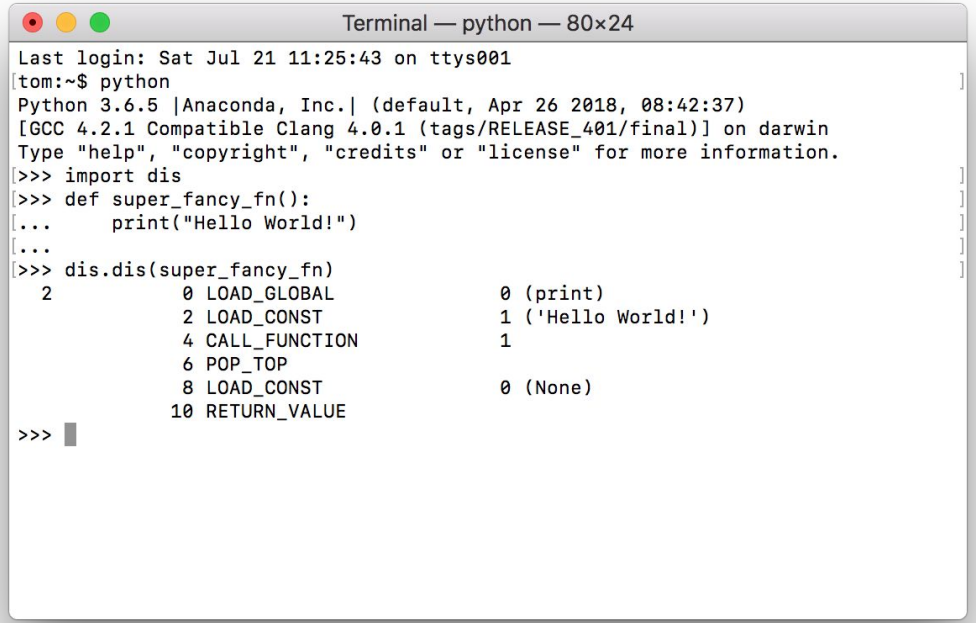


Interpreting bytecode

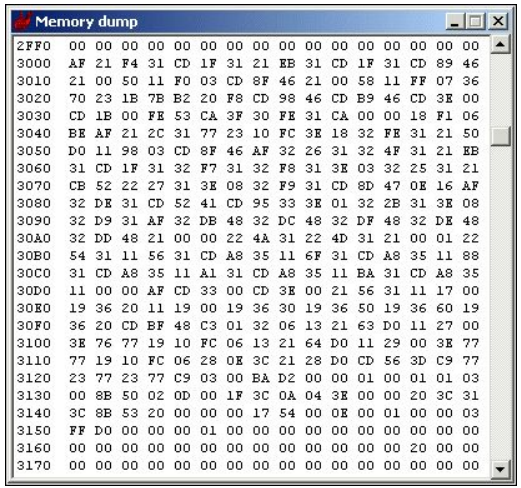
- Compiling source code to bytecode, then execute it on a virtual machine



Interpreting bytecode



So how do we get from source code to bytecode?

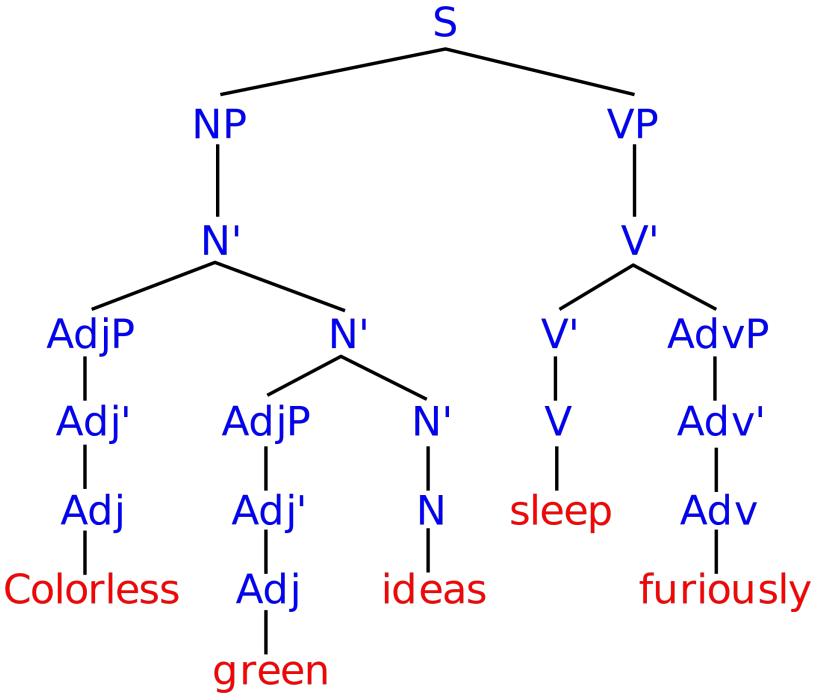


Take a byte...

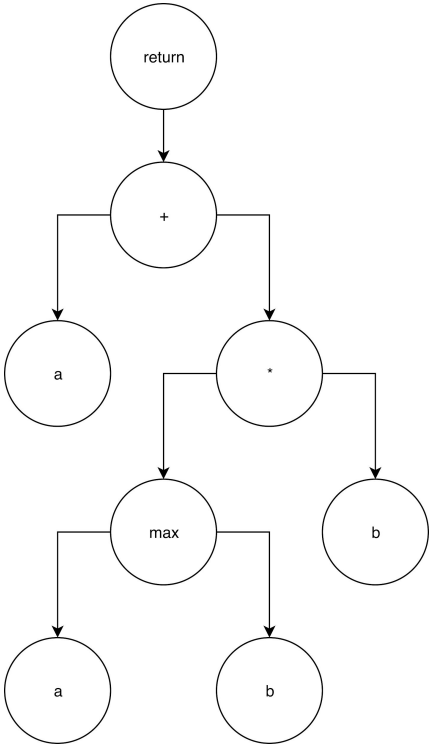
- Bytecode is like machine code, except for our virtual machine
- Generate bytecode based on the structure of the tree by walking through the AST

Concrete syntax vs abstract syntax vs semantics

Colourless green ideas sleep furiously.

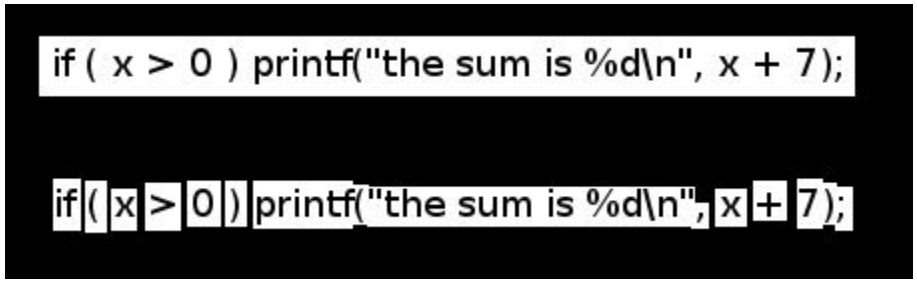


return a + max(a,b) * b



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)*



♪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

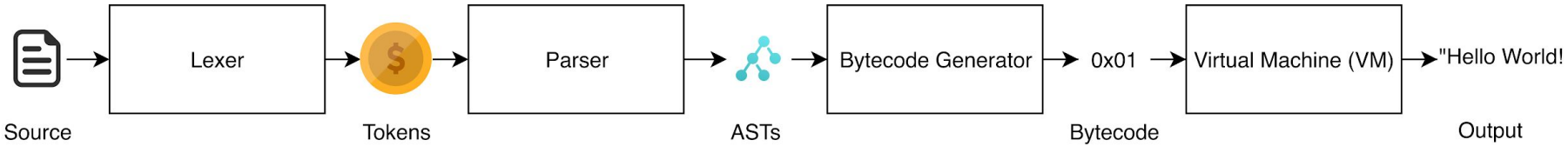
How?

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

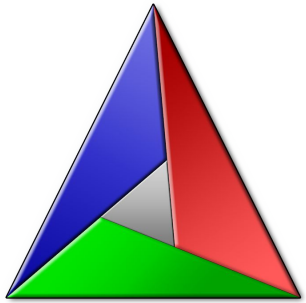
Optimisations

- 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