Computer Systems 1

Lecture 19

Programming Languages

Dr. John T. O'Donnell
School of Computing Science
University of Glasgow

Topics

1. Survey about programming in Python
2. Languages and systems
   - Syntax — easier but less important
   - goto
   - Semantics — harder but more important
3. Lists, +, +=, iterators, and for loops
   - Mysteries
   - Are two nodes with the same value identical?
   - Low level list manipulation
   - The + operator
   - The += operator
   - Iterators
   - for loops
   - Revisiting the mysteries
4. Compilers

Announcements

- Drop in tutorials will continue this week
- Similar schedule as previous weeks, but not for today (Tuesday)
- Advanced topics: Friday, 15 March, 4-5pm, F121
- The lab this week is for working on the assessed exercise
- Survey about programming in Python

Survey about programming in Python — Part 2

- This is completely optional
- There is a research group investigating the process of learning a programming language
- We are running two surveys
  1. Starting Tuesday 5 March at 1pm, closing Tuesday 12 March at 12:00 noon
  2. The lecture on Tuesday 12 March will be about programming language semantics: relating Python to compilation patterns and machine language that's today!
  3. Followup survey starting Tuesday 12 March at 1pm, closing Tuesday 19 March at 12:00 noon
  4. The last lecture of the course is Friday 21 March, and will discuss the results
- Participation is optional and anonymous.
- It will not affect your grade in any way, but we hope you find it interesting and helpful in learning programming languages

A request

- The lecture today won't be able to cover all the slides
- Please read through the slides, and also try out the Python program on Moodle
- After you've done that, please try the second part of the survey (there's a link to the survey on Moodle)
- Remember this is optional and anonymous and will not affect your assessment
- But we hope you'll find it interesting, and also learn more about Python

Advice on software engineering

What should a software engineer study if they want to learn how to write efficient code?

Hyde, R. (2009). The Fallacy of Premature Optimization. *Ubiquity*, Association for Computing Machinery, February 2009. https://doi.org/10.1145/1569886.1513451

What should a software engineer study? CS1S!

Hyde says (quotation):

1. The first subject to master is machine organization (elementary computer architecture). Because all real programs execute on real machines, you need to understand how real machines operate if you want to write efficient code for those machines.
2. The second subject to study is assembly language programming. Though few programmers use assembly language for application development, assembly language knowledge is critical if you want to make the connection between a high-level language and the low-level CPU. . . . It doesn't really matter which assembly language you learn nor does it matter which CPU's instruction set you study. What you really need to learn are the basic operational costs of computation.
3. The third important subject a software engineer should study is basic compiler construction, to learn how compilers translate high-level language statements into machine code.

Syntax, semantics, compilation

- Primary aspects of a programming language
  - Syntax is the *form* of a program
  - Semantics is the *meaning* of the program
  - Compilation (or interpretation) is how the language is implemented so it can run on a computer

Syntax

- Syntax is the form of a program
- Did you spell the keywords correctly? Is the punctuation right?
- Syntax is easy
  - The rules are clear cut
  - If in doubt, just look it up
  - Example: various languages have different names for the same thing: *Bool*,*Boolean*, *Logical*. These differences are superficial

Syntax errors

- Compilers insist that the syntax is right
  - In English, if you spell a word wrong or have a missing comma you'll (probably) still be understood
    - ★ But — see Eats, Shoots and Leaves by Lynn Truss. Note the comma! What was meant was "The panda eats shoots and leaves", not "The panda eats, shoots, and leaves"
- Can't a compiler be equally forgiving?
  - There were experiments with compilers that guess what the programmer meant
  - It was a disaster: the compiler nearly always guessed correctly...
  - But occasionally it would guess wrong
    - ★ How can you debug a program when the compiler didn't translate what you wrote, but something different?
    - ★ You have to debug code that is not in the file! And you cannot see it!
    - ★ You want the compiler to insist that the syntax is absolutely correct

Example of syntax: operator precedence

- Expressions can contain many operations, but the computer can do only one operation at a time
- We can make the operations explicit by using parentheses around each operation
- You don't have to write the parentheses, but the compiler needs to know where they go!
- a + b + c is parsed as (a + b) + c
- a + b * c is parsed as a + (b * c)

Deeper example of syntax: ambiguity

- A language is *ambigous* if a sentence in the language can have two different meanings
- English is full of ambiguity
- Programming languages are designed to avoid ambiguity, most of the time
- If ambiguity is possible, the compile needs to know how to resolve it, *and so does the programmer*

Ambiguity in if-then-else

This is ambiguous: which if does the "else S2" belong to?

```
if b1 then if b2 then S1 else S2
```

There are two interpretations, and they lead to different results

```
if b1 then { if b2 then S1 else S2 }
  b1 = true,  b2 = true     S1
  b1 = true,  b2 = false    S2
  b1 = false, b2 = true
  b1 = false, b2 = false
```

```
if b1 then { if b2 then S1 } else S2
  b1 = true,  b2 = true     S1
  b1 = true,  b2 = false
  b1 = false, b2 = true     S2
  b1 = false, b2 = false    S2
```

How does Python prevent ambiguity?

- The structure of the program is determined by indentation
  - This means it is essential to indent the program properly
  - It makes the program structure highly visible to the programmer
- Some languages use braces to indicate structure (e.g. C, Java)
  - The compiler ignores the indentation, and uses the braces
  - Programmers tend to focus on the indentation and may overlook the braces
  - This is more error-prone

goto

- Usually programs are more readable and more reliable if written with while loops, if-then-elif-elif-else, for loops, and similar higher level constructs
- Programs that jump around randomly with goto statements are harder to understand, and likely to contain bugs
- This gave the goto statement a bad reputation
- But
  - The goto statement is simply a jump instruction, and it is *essential* for use at low level
  - There are some circumstances where goto may be the best solution, but these are rare

goto spelled differently

- Some programming languages provide restricted forms of goto
  - break
  - continue
- These are goto statements without a label, but with a predefined destination they go to
  - Advantage — you may recognise the pattern being used by a programmer
  - Disadvantage — you need to know for sure to where your goto goes

The break statement

- C and its descendants have break, as does Python
- Break is a goto that goes to the end of the innermost loop it's in
- In Python you can have an else clause to execute when a for or while loop finishes, but this is skipped if you terminate the loop with break
- What if you want to break out of several loops?
  - There is no good way to do this in Python
  - It's best to restructure your program, to avoid break

Where does break go?

```
for i in range (1, 5):
    print (i)
    for j in range (20, 22):
        print (j)
        if i == 3:
            break
        for k in range (40, 43):
            print (k)
            if k == 41:
                break
            print (k)
```

The continue statement

- In C and Python, continue goes to the end of the current loop which then continues executing
- It's a way of staying in the loop but skipping the statements after the continue
- Warning! Several programming languages have a statement that is *spelled* continue — but it does nothing and is equivalent to the pass statement in Python

Break and continue: translation to low level

```
loop1
   if (i<n)=False then goto loop1done
   ...
   if ... then goto loop1       ; This is a continue statement
   ...
   if ... then goto loop1done   ; This is a break statement
   ...
   goto loop1
loop1done
```

- continue goes to the top of the innermost loop containing the continue: it skips the rest of *this iteration*
- break goes to right after the end of the innermost loop containing the break: it *exits this loop*

Semantics

- Semantics means *the meaning*
- The *semantics* of a language is *what a program in the language means*
- The semantics of a program is *the meaning of the program*
  - Given its inputs, what are its outputs?

How is the semantics of a language defined?

- Using natural language to describe it
  - Vague description in English of what each construct does
  - A carefully written description in English, written to be as precise as possible
- Using mathematics or program transformation
  - Denotational semantics: a precise mathematical specification. Given a program, it gives a mathematical function from program inputs to outputs
  - Operational semantics: gives a sequence of reduction rules that give a precise model of the program's execution
  - Transformational semantics: a translation from a program into a simpler language, where the semantics is assumed to be clear

Why do the translation from high to low level?

- This is the semantics of the high level constructs
- It explains precisely what the high level means
- It shows what the compiler will do with the program (compilers do intermediate level translations like this; many compilers use several intermedate steps)
- It makes explicit the execution time of the construct
- The low level is very close to assembly language
- It's easier to go from high level to assembly language in two small steps, rather than one giant leap

Watch out for loose explanations

- Here's a quotation from https://docs.python.org/3.7/tutorial/introduction.html
  - "The while loop executes as long as the condition (here: $a < 10$) remains true"
- That is wrong!
- It says that if a is changed in the middle of the loop, making it less than 10, the loop will stop executing
- But the computer does not continuously monitor $a < 10$ and break out of the loop as soon as it becomes false
- To see what happens, look at the compilation pattern: the translation to low level "goto" form
- The while loop checks the boolean $a < 10$ at the top of the loop; if false it exits the loop, and if true it executes *the entire body of the loop* even if the boolean became false in the middle

Semantics of while

```
while b do S
```

is translated to

```
L1: if not b then goto L2
    S
    goto L1
L2:
```

- If you understand the meaning of :=, goto, if b then goto, you can understand the meaning of *every high level construct*
- b is an arbitrary boolean expression
- S is an arbitrary statement
- L1 and L2 are "fresh" labels: they can't be used anywhere else

Mysteries

- How are lists represented?
- What do the + and += operators do?
- What is an iterator, and how does it work?
- What does a for loop do? How does it compare with a while loop?

Let's do some matrix calculations

Set up a list of 5 elements

```
coords = [[0,0]] * 5
print ('coords = ', coords)  # [[0,0], [0,0], [0,0], [0,0], [0
```

The result is

```
coords =  [[0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]
```

Looks good! Now set some sub-elements:

```
coords[3][1] = 7
coords[4][0] = 5
```

Result may not be what we expected, or wanted:

```
coords[2] =  [5, 7]
coords =  [[5, 7], [5, 7], [5, 7], [5, 7], [5, 7]]
```

What's going on?

What does += mean in Python?

- Google search for += Python
  - "The expression a += b is shorthand for a = a + b, where a and b can be numbers, or strings, or tuples, or lists (but both must be of the same type)."
- Stack overflow search for += Python
  - "+= adds another value with the variable's value and assigns the new value to the variable."
  - "It adds the right operand to the left. x += 2 means x = x + 2. It can also add elements to a list"
- These statements are all wrong

Is a += b shorthand for a = a + b?

Try a = a + [3, 4]

```
print ('Defining a = a + [3, 4]')
a = [1, 2]
b = a
print ('a = ', a, '  b = ', b)
a = a + [3, 4]
print ('a = ', a, '  b = ', b)
```

The results: a = [1, 2, 3, 4] b = [1, 2] Now try a += [3, 4]

```
print ('Defining a += [3, 4]')
a = [1, 2]
b = a
print ('a = ', a, '  b = ', b)
a += [3, 4]
print ('a = ', a, '  b = ', b)
```

The results: a = [1, 2, 3, 4], b = [1, 2, 3, 4]

How do we figure out problems like this?

- Need to understand all the fundamental concepts of the language
- It's best to study authoritative source systematically
- Develop a clear model for what the language constructs really mean
- And here's some advice: from Stack Overflow
  - "it's a basic operator for python (and many other languages too), you should start with google, if you never read any python references. "
- This may be ok if you already understand all the foundations and just want to look up a detail, but otherwise *this is poor advice*
- Let's look in more detail at lists...

Are two nodes with the same value identical?

Define b = a — this is pointer assignment

```
a = [1, 2, 3, 4]
b = a
print ('(1)  a = ', a, ' b = ', b)
```

Both a and b have the value [1, 2, 3, 4], but are the actual nodes in their representations distinct, or shared?

Try modifying an element of a and see if it changes b, and vice versa

```
a[1] = 100
b[3] = 300
print (' a = ', a, ' b = ', b)
```

Run this, and you'll see that changing an element of either a or b also changes the corresponding element of the other. The values of a and b are simply pointers, and they point to the same node.

Low level list manipulation

- You can assign pointers: if a is a list, b = a makes b point to the same node a points to

```
; b = a,    where a is a list
   load   R1,a[R0]     ; R1 = a
   store  R1,b[R0]     ; b := a
```

- You can write expressions that create a new list, but don't modify any existing lists: a = a + b
- You can modify an existing list structure
  - a.copy()
  - a.append(b)
  - a.extend(b)
- All of these are implemented with while loops that traverse a

b = a.copy()

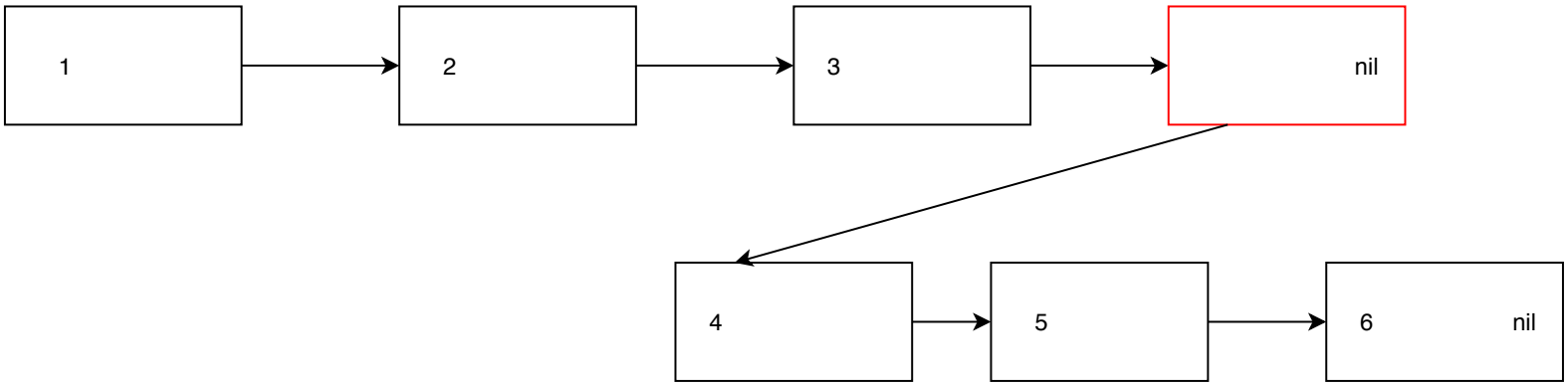
Traverse a and make a new node for each node in a; link the new nodes together to form the result. The nodes in b have the same values as the nodes in a, but they are distinct nodes

```
p := a
b := nil
while p /= nil do
    nn := newnode()
    *nn.value := *p.value
    *nn.next := nil
    *b.next := nn
    p := *p.next
```

(It's a little more complicated, you have to remember the beginning of b — using a header node is helpful.) Now if you modify one of the lists (a, b) the other list is not affected

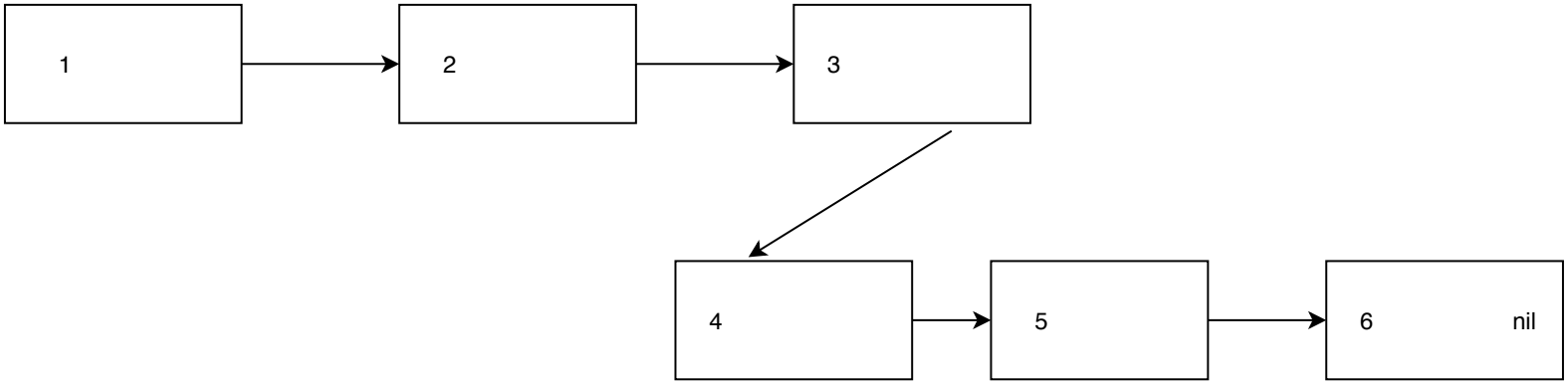
List is represented as nodes

Appending to a list



- a = [1, 2, 3]
- b = [4, 5, 6]
- a.append(b)
- A new node is created: call newnode()
- The newnode has value=b and next=nil
- The last node in a had next=nil; that is changed to point to the new node
- Result: a = [1, 2, 3, [4, 5, 6]]

Extending a list



- a = [1, 2, 3]
- b = [4, 5, 6]
- a.extend(b)
- The last node in a had next=nil
- That nil pointer was changed to point to b
- Result: a = [1, 2, 3, 4, 5, 6]

The + operator

- You can concatenate two lists a and b with a + b
- The + operator does not modify either a or b
- It creates a new list, with copies of the nodes in a, followed by b
- Since it doesn't modify existing containers, you can use + on both
  - mutable containers, e.g. lists
  - immutable containers, e.g. string

The + operator

- You can concatenate two lists a and b with a += b
- The += operator does not copy a, it modifies a by extending it
- Since += modifies existing containers, you cannot use += on immutable containers, e.g. strings

List with + and +=

- You can write a + b where a and b are lists; if either is a list, both must be lists
- You can write a += b where
  - a is a list
  - b is either a list or an iterator

Iterators

- A container class can have an iterator
- There are built-in default iterators for lists and numbers
- You can define your own class and iterator
  - You define a method called `__iter__` which creates a new iterator and initializes it
  - An iterator provides a method called `__next__` and you can invoke this with xyz.next()
  - The implementation of next

Defining an iterator for even numbers

```
class EvensClass:
  def __iter__(self):
    self.state = 0
    return self

  def __next__(self):
    x = self.state
    self.state += 2
    return x

Evens = EvensClass()
EvenIterator = iter(Evens)
```

Using the iterator

```
for i in EvenIterator:
    print (i)
    if i > 15:  # Try commenting this out: the iterator is unb
        break

print ('The for loop has finished, now call next directly')
print(next(EvenIterator))
```

Running the even iterator

```
Defining iterator for even numbers
0
2
4
6
8
10
12
14
16
The for loop has finished, now call next directly
18
```

Extending a list with an iterator

You can use += to extend a list with an iterator

```
a = [1, 2, 3, 4]
a += range(20,25)
print ('a = ', a)
```

But you can't use + on a list and an iterator:

```
a = [1, 2, 3, 4]b = [1, 2, 3, 4]
# b = b + range(20,25)  # This line gives syntax error
```

for loops in Algol and descendants

```
sum := 0
for i := 0 to n-1 do sum := sum+i
print (sum)
```

The translation needs 4 machine language instructions for the loop control

```
            sum := 0
            i := 0
avacodo     if i>= n then goto avacododone
            S
            i := i + 1
            goto avacado
avacododone
            print (sum)
```

for loops in Python

- A Python for loop traverses a sequence defined by an iterator
- If the iterator terminates, so does the for loop
- The iterator could also go on forever
- The iterator could give a sequence of numbers, or it could traverse a list, or some other container
- This is convenient but it hides what's actually going on

Revisiting the mysteries

Focus on:

- What low level operations are being used: pointer assignment? copy?
- Know when you have two pointers to the same node
- Know when newnode has been called

```
a = [1, 2]
b = a
a = a + [3, 4]
print ('a = ', a, '  b = ', b)
# Here, the results are final values:  a =  [1, 2, 3, 4]   b =  [1, 2]

a = [1, 2]
b = a
a += [3, 4]
print ('a = ', a, '  b = ', b)

# Now the results are: a = [1, 2, 3, 4], b = [1, 2, 3, 4]
```

In the second example, a += b modified the structure of a. Since b points to the same node as a, this change also affects b.

Compilers

- We have been writing algorithms in high level language notation and then translating it manually to assembly language
- A compiler is a software application that performs this translation automatically
- A programming language is a precisely defined high level language
- The compiler makes programming easier by allowing you to think about your algorithm more abstractly, without worrying about all the details of the machine

Source and object

- The original high level language program is called the source code — it's what the programmer writes
- The final machine language program which the compiler produces is called the object code — it's what the machine executes

Compilation

- A compiler translates statements in a high level language into assembly language
- In developing an assembly language program, it's best to begin by writing high level pseudo-code (this becomes a comment) and then translate it
- This approach helps keep the program readable, and reduces the liklihood of getting confused
- Each kind of high level statement corresponds to a standard pattern in assembly language. *Follow these patterns!*

How a compiler works

- Your high level source program is just a character string — the computer cannot execute it directly
- The compiler reads the source program, checks its syntax, and analyses its structure
- Then it checks the types of all the variables and procedures
- Most advanced compilers translate the program to an intermediate "goto form", just as we are doing
- The program is finally translated to assembly language: "code generation"
- The assembler translates the assembly language to machine language (the Sigma16 application contains an assembler)

Major tasks in compilation

- Parsing — check the source program for correct syntax and work out the program structure (similar to "diagramming sentences" in English grammar)
- Type checking — work out the data type of each variable (integer, character, etc.) — then
  - Make sure the variables are used consistently
  - Generate the right instructions for that data type
- Optimisation — analyse the program to find opportunities to rearrange the object code to make it faster
- Code generation — produce the actual machine instructions

Parsing

- The syntax of a language is its set of grammar rules
- If the source program contains a syntax error, the compiler will not understand what you mean
  - if x<y then a = 1 — ok, can be translated
  - if x?y than b = 2 — syntax errors! what does it mean?
- If there is a syntax error, you do not want the compiler to guess what the meaning is — that leads to unreliable software

Types

- Type checking is one of the most important tasks the compiler performs
- There are many data types supported by a computer
  - Binary integer
  - Two's complement integer
  - Floating point
  - Character
  - Instruction
- The computer hardware works on words, and the machine does not know what data type a word is.
- It's essential to use the right instruction, according to the data type

Integer and floating point

- An integer is a whole number: 23, -47
- A floating point number may have a fraction and exponent: 7.43×10^{28}
- We have seen how an integer is represented: two's complement
- Floating point representation is different from two's complement
- Most computers have separate instructions for arithmetic on integer and floating point
  - add R1,R2,R3 — integer addition
  - addf R1,R2,R3 — floating point addition
- The machine doesn't know what the bits in the registers mean — you must use the right instruction according to the data type

Typechecking

- The compiler checks that each variable in the source program is used correctly
  - If you add a number to a string, it's a type error
- Then it generates the correct instructions for the type
  - For integer variables it uses add
  - For floating point variables it uses addf
  - *These are different instructions!*
- This eliminates one of the commonest kinds of error in software

Write programs at a high level

Every rule has some rare exceptions, but almost always these are good principles:

- Write conditionals using if-then or if-then-else
- Write loops using the most appropriate construct
  - Often, a while loop is best
  - To traverse an array, a for loop is best
- Avoid goto statements (and if your language has break statements, avoid those)
- Use a straightforward style that's easy to read

Include the high level code in your program, as full line comments

Use patterns to translate to low level

- Low level algorithm contains just
  - Assignment statements
  - goto label
  - if b then goto label
- Each high level construct is translated to low level using a fixed pattern

Include the low level code in your program, as full line comments — after the high level code

Translate low level to assembly language

- Each low level statement should be a full line comment, followed by the instructions needed to implement that statement
  - $x := y + z \quad \Rightarrow$

```
; x := y + z
    load    R1,y[R0]     ; R1 := y
    load    R2,z[R0]     ; R2 := z
    add     R1,R1,R2     ; R1 := y + z
    store   R1,x[R0]     ; x := y + z
```

  - goto phase2 $\Rightarrow$

```
; goto phase2
    jump    phase2[R0]   ; goto phase2
```

  - if $x < y$ then goto phase3 $\Rightarrow$

```
; if x<y then goto phase3
    load    R1,x[R0]     ; R1 := x
    load    R2,y[R0]     ; R2 := y
    cmp     R1,R2        ; compare x, y
    jumpge  phase3[R0]   ; if x<y then goto phase3
```

Practical programming tip

1. Write the high level algorithm
2. Hand execute it to be sure it's correct
3. Type it into your file, as full line comments
4. Translate it to low level form
5. Hand execute it to be sure it's correct
6. Type it into your file, as full line comments
7. Duplicate the low level code (copy and paste)
8. Now go through the second copy of the low level code, and insert the assembly language after each statement
9. Hand execute it to be sure it's correct
10. Run it on the emulator, in single step mode, and check that the program does what you predicted in your hand execution

Compiling

https://xkcd.com/393/