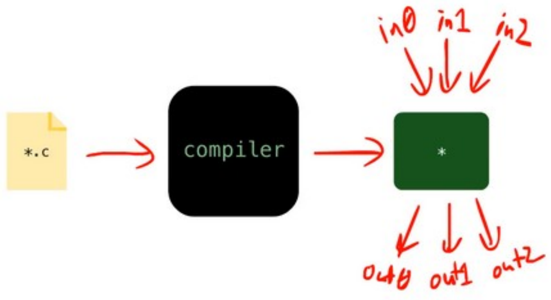
How do computers read code? [1]

1. In your first programming class, you were probably introduced to ***the idea of a compiler***. Here's how it works. You first write out your program, but a computer can't read it yet. So, in order to actually run your program, you first need to pass it through this special program called the compiler. Then, out pops a new version of your program that *can* be read by a computer. It was probably then tested by running it with a bunch of inputs and expected outputs or something.



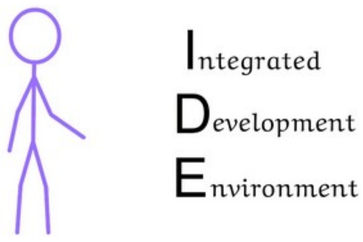
So there are ***two versions*** of your program:

- the one you wrote but a computer can't read,
- and the magically generated one that a computer can read.

Except, it's better than magic. The compiler is a complex machine that bridges the gap between human-readable code and computer-readable code.

2. So, ***what exactly is the compiler doing***? And what does the executable version of your program actually look like?

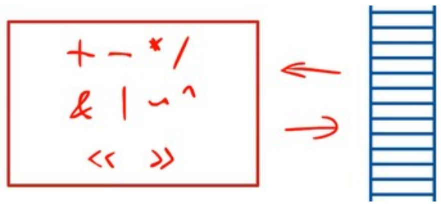
- What? You say you have no idea what I'm talking about? Oh, I think I know what might have happened. In your first programming class, you probably just used an IDE, in which case this whole process was hidden from you. When you click the run button, your work is saved, your program is built, and it runs automatically. So now that we're up to speed, what does the executable look like, and how does the compiler…



- What? You still don't know what I'm talking about? All you did was Python scripting??? **sigh** Oh my god, there are so many edge cases!

OK, this is what happens to programs in general, deal with it, how does it work, I don't know, let's find out!

3. At a low level, computers processors can only do a small number of things. They can read and write to memory, and they can do math with numbers they are holding. Modern processors do other things too, but this is basically it. Now, an ***executable program*** – the one generated by the compiler – is just a list of instructions for the processor to follow, written in binary.



The instructions are things like:

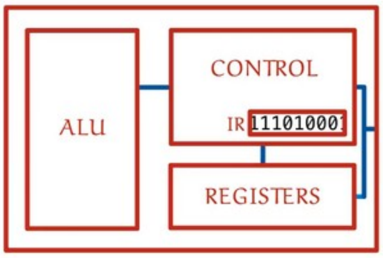
- read these bytes from memory,
- do stuff to them,
- write bytes to memory,
- jump forward this many lines,
- jump back this many lines, but only if this flag is set,

… stuff like that. A program, expressed in a list of binary instructions, is called ***machine code***, and this is the kind of program that your computer can actually read.

```
11100101100010010100100001010101
10001001111011000111110110001001
01000101110001111110100001110101
00000000000000000000000011111000
11111100010001011100011100000000
00000000000000000000000000000000
01000101100010110000111011101011
01000101101011110000111111111000
11111000010001011000100111101100
00000001111111000100010110000011
00111011111111000100010110001011
11101010011111001110100001000101
01011101111110000100010110001011
```

1 https://www.youtube.com/watch?v=QXjU9qTsYCc

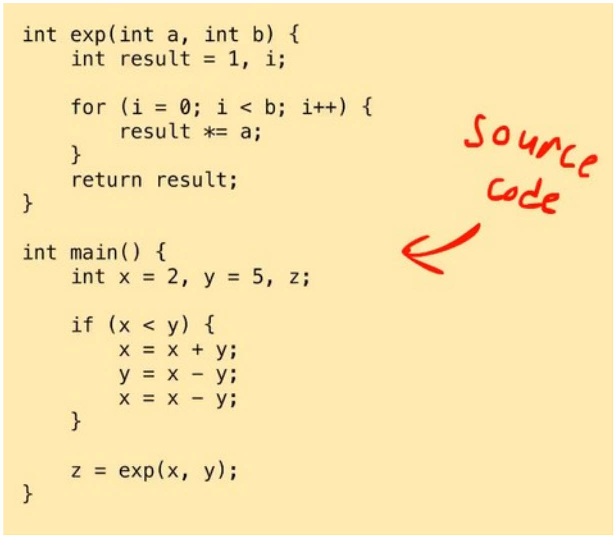
4. But why does a ***computer processor*** only read programs that look like this? Why this specifically? Well in short, here's how a processor works. The processor already contains the circuitry to do all of these instructions, but the correct circuitry only gets connected together when the corresponding instruction gets fed into it. The ones and zeroes in the instruction cause certain transistors to open or close, which ends up connecting the correct circuitry together to execute *that* instruction.



If you want to look more into how this works, you might wanna check out *Crash Course Computer Science*, particularly episodes 5 through 8. (Episodes 3 and 4 are also helpful if you need a refresher on binary and logic gates… though you could just watch my video too!) And for the record, *Crash Course* isn't paying me to recommend them; I just really like this series. But in short, just know that executable programs look like *this*.

5. But when you first learned to program, you didn't need to know anything about these complicated machine code things like memory management, operations on bytes, or conditional jumps. Programming was about variables, and `if`-statements, and loops, and functions. Well, these things are just ***higher-level constructs*** that make it easier for humans to think about programming.

A program, expressed in this form, is called ***source code***. It's the version of a program that a human understands, and thus the version that most humans actually write code in. The compiler's job is to take this source code that is human-readable, and turn it into machine code that is computer-readable. But how does it do that? How does it turn a string of text into a list of instructions in binary?



```
int exp(int a, int b) {
    int result = 1, i;

    for (i = 0; i < b; i++) {
        result *= a;
    }
    return result;
}

int main() {
    int x = 2, y = 5, z;

    if (x < y) {
        x = x + y;
        y = x - y;
        x = x - y;
    }

    z = exp(x, y);
}
```

6. Let's look at some examples. Here's ***a pretty simple program***:
   - declare a variable of type integer that we'll call x,
   - then assign it a value of 3.

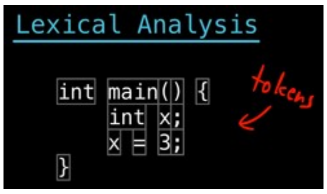
```
int main() {
    int x;
    x = 3;
}
```

For now, this program only exists as source code. I know it looks like it has some kind of structure, but for the computer, it's just a meaningless sequence of characters – it's just text.

```
int·main()·{\n\tint·x;\n\tx·=·3;\n}\n
```

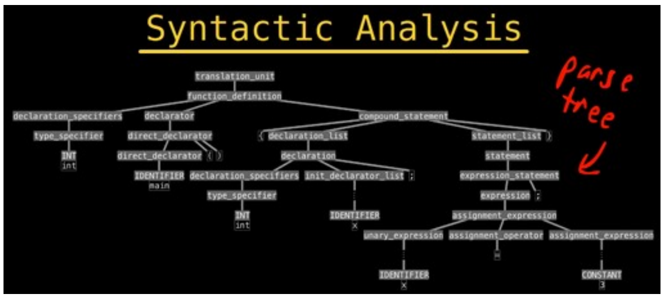
And I know that this program doesn't really do anything useful, but we're starting simple.

7. Let's pass this source code into the compiler and see what it does:
   - The compiler first divides the text up into individual ***tokens***. It's kind of like the compiler is figuring out what the "words" are in this program.
   - Then, the tokens are organized into a hierarchical structure known as a ***parse tree***, which is like figuring out what the "grammar" is in this program – the structure.

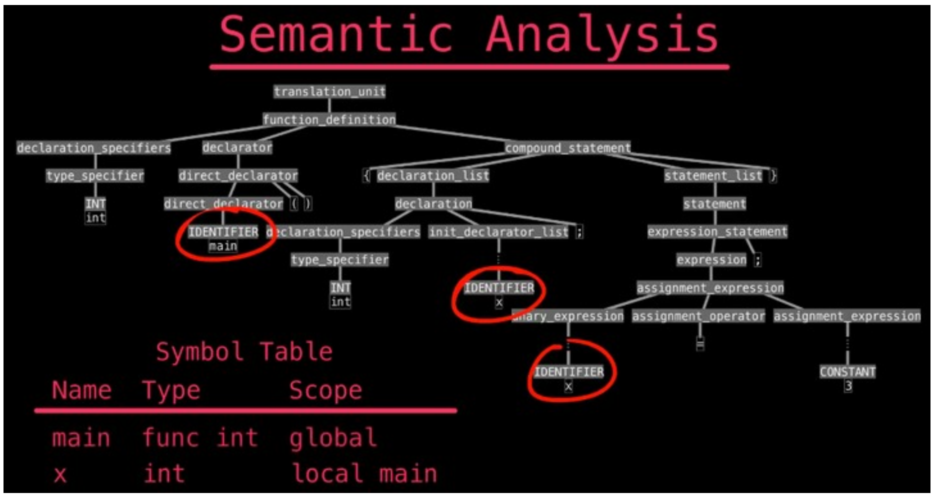


Lexical Analysis

```
int main() {
    int x;
    x = 3;
}
```



- Then, the compiler records ***context*** about the program, including variable and function names. This is the stuff that a computer needs to keep track of in different parts of the program. In our case, the only context we need is the variable x. Oh, and the main function too, but that's less important for us.



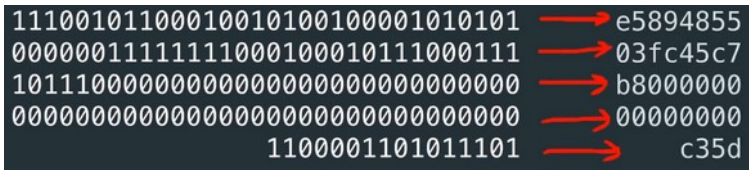
Symbol Table

Name	Type	Scope
main	func int	global
x	int	local main

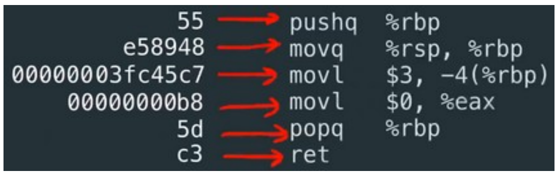
- The final step is to traverse the tree, and figure out some ***machine code*** that would effectively do the same thing as this particular source code.

I just wanna clarify that, typically, compilers don't go directly from the parse tree to the machine code. There's usually a few intermediate steps that we're going to skip over.

8. This is what the machine instructions look like in binary. It's a little hard to read and interpret, so let's shorten it by writing in hexadecimal. Actually, it's still a pain to read… Let's write it out as ***assembly code***, which is a more human-readable version of the machine code.

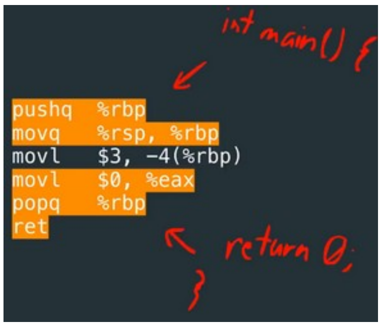


```
11100101100010010100100001010101 → e5894855
00000011111111000100010111000111 → 03fc45c7
10111000000000000000000000000000 → b8000000
00000000000000000000000000000000 → 00000000
                1100001101011101 →     c35d
```



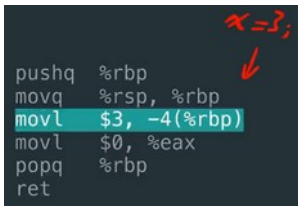
```
              55 → pushq  %rbp
          e58948 → movq   %rsp, %rbp
00000003fc45c7   → movl   $3, -4(%rbp)
      00000000b8 → movl   $0, %eax
              5d → popq   %rbp
              c3 → ret
```

That's better! Now, we're going to ignore this stuff here (just know that it's responsible for starting and ending the main function, which determines where the program starts and ends):



```
int main() {
pushq  %rbp
movq   %rsp, %rbp
movl   $3, -4(%rbp)
movl   $0, %eax
popq   %rbp
ret
return 0;
}
```

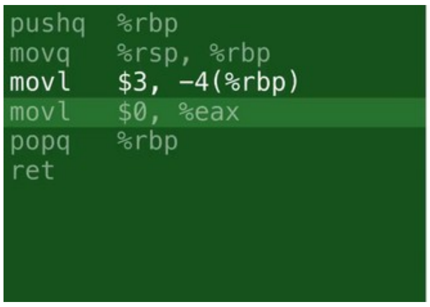
- This is the instruction that corresponds to the assignment of the variable x.



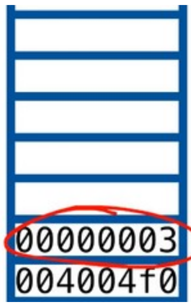
```
x=3;
pushq  %rbp
movq   %rsp, %rbp
movl   $3, -4(%rbp)
movl   $0, %eax
popq   %rbp
ret
```

- This instruction says to "move" the number 3 into this memory location.

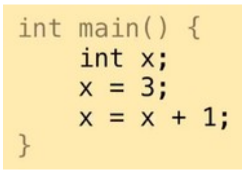
Let's ***run*** the program and see what happens… And as we'd expect, the number 3 got put into *that* memory location. It looks like the compiler decided that this part of memory is where the variable x lives. And that's it! The compiler took our source code, which said to take a variable x and assign it the value 3, and translated it into machine code, which says to place the value 3 in this part of memory.



```
pushq  %rbp
movq   %rsp, %rbp
movl   $3, -4(%rbp)
movl   $0, %eax
popq   %rbp
ret
```

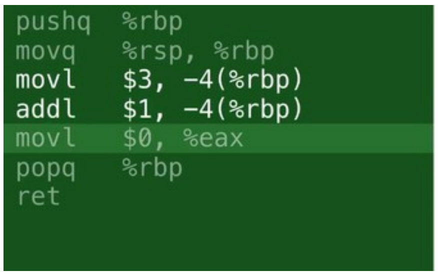


9. So, our program is kind of boring right now. What happens if we change it? Let's add a line to ***increment*** x by 1, after it's been assigned 3. We assign it the value of x's current value plus 1. Now, we pass it into the compiler again to generate a new version of our machine code. The compiler finds tokens, ...parses, ...contextualizes, ... and generates.

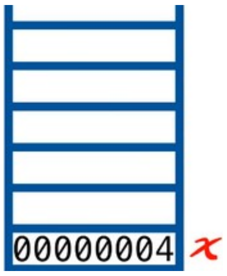


```
int main() {
    int x;
    x = 3;
    x = x + 1;
}
```

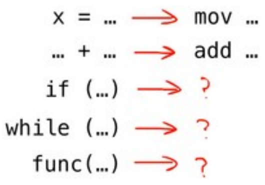
It looks like there's only a single new instruction: to *add* the value of 1 to this memory location. After all, this is the location that the compiler decided that x lives. Modifying the variable x is equivalent to modifying the value stored in this memory location. Let's run it... The value 3 appears in that memory location designated for the variable x, and then the value becomes 4.



```
pushq   %rbp
movq    %rsp, %rbp
movl    $3, -4(%rbp)
addl    $1, -4(%rbp)
movl    $0, %eax
popq    %rbp
ret
```

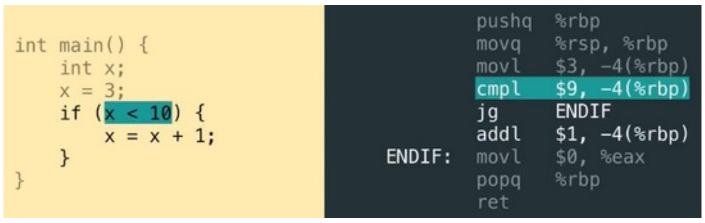


10. Now so far, we've seen how variable assignments and simple mathematical operations get translated. But it's not so simple for if-statements, loops and functions. There are no machine instructions that are direct equivalents. Instead, we need to emulate their behavior with instructions that do exist. Let's start with an **if**-***statement***:



- in an if-statement, we only execute the code in *this* block, if *this* condition is true;
- if the condition is false, we skip over this code.

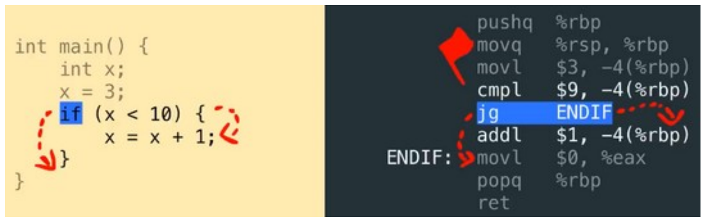
In assembly, the code inside the block gets translated normally, but before it, we have some instructions for evaluating the condition, and then we have a ***conditional jump*** instruction: in this case, to jump past the block we want to skip – but only if we're supposed to skip it.



```
int main() {
    int x;
    x = 3;
    if (x < 10) {
        x = x + 1;
    }
}
```

```
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3, -4(%rbp)
        cmpl    $9, -4(%rbp)
        jg      ENDIF
        addl    $1, -4(%rbp)
ENDIF:  movl    $0, %eax
        popq    %rbp
        ret
```

The processor knows whether or not we're supposed to take this jump, based on the result of the previous instruction. That instruction temporarily set some ***flags*** in the processor, so we could remember the result by the time we got to this conditional jump to skip over this block. If we're not supposed to skip it, then the processor ignores the jump, and continues normally, conveniently executing the code inside the block. [Did you] Noticed that these machine instructions are effectively doing the same thing as our if-statement?



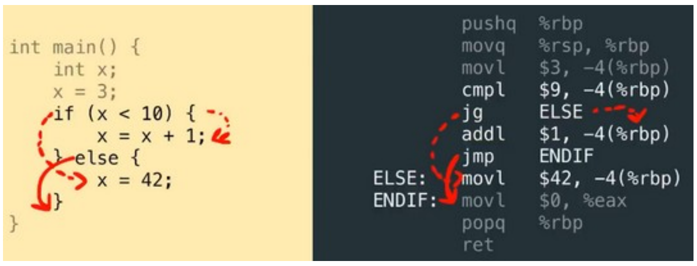
```
int main() {
    int x;
    x = 3;
    if (x < 10) {
        x = x + 1;
    }
}
```

```
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3, -4(%rbp)
        cmpl    $9, -4(%rbp)
        jg      ENDIF
        addl    $1, -4(%rbp)
ENDIF:  movl    $0, %eax
        popq    %rbp
        ret
```

11. Let's see how jumps can emulate other source code behavior. There's the **if-else**-***statement***:

- only execute this block if this condition is true;
- otherwise, execute *this* block.

In assembly, we have the code in the first block and the code in the second block. Before the first block, we have a comparison and a conditional jump, like what we had earlier. In between the blocks, we have a regular non-conditional jump instruction. This is so that execution skips the second block if it just finished the first one, which is kind of how the if-else-statement works if you think about it.



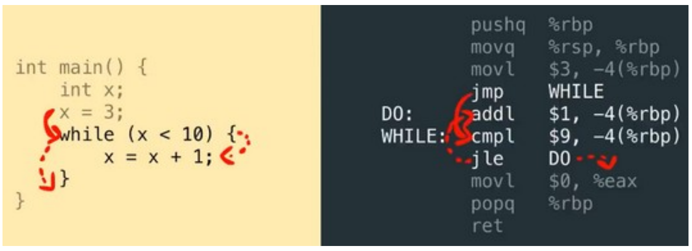
```
int main() {
    int x;
    x = 3;
    if (x < 10) {
        x = x + 1;
    } else {
        x = 42;
    }
}
```

```
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3, -4(%rbp)
        cmpl    $9, -4(%rbp)
        jg      ELSE
        addl    $1, -4(%rbp)
        jmp     ENDIF
ELSE:   movl    $42, -4(%rbp)
ENDIF:  movl    $0, %eax
        popq    %rbp
        ret
```

Then there's the **`while`-*loop***:

- only execute this block if this condition is true,
- and *keep* executing it over and over again until it's not true anymore.

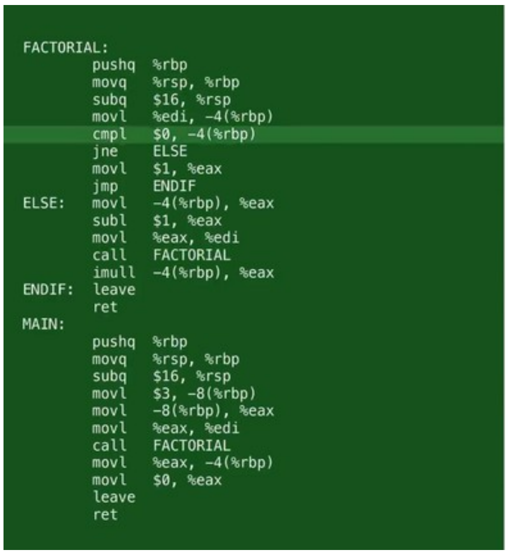
In assembly, we have the block's code, the instructions to evaluate the condition, and the jumps emulating the loop.



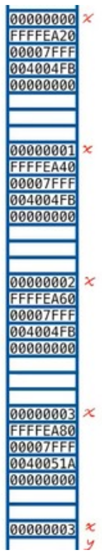
```
int main() {
    int x;
    x = 3;
    while (x < 10) {
        x = x + 1;
    }
}
```

```
            pushq   %rbp
            movq    %rsp, %rbp
            movl    $3, -4(%rbp)
            jmp     WHILE
DO:         addl    $1, -4(%rbp)
WHILE:      cmpl    $9, -4(%rbp)
            jle     DO
            movl    $0, %eax
            popq    %rbp
            ret
```

12. ***Functions*** are a little more complicated. Basically, functions encapsulate a code block, so that it can be used in multiple parts of a program. Most programmers out there should know that they also isolate context and they can do recursion and stuff. This is what its equivalent assembly code looks like.



```
FACTORIAL:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        cmpl    $0, -4(%rbp)
        jne     ELSE
        movl    $1, %eax
        jmp     ENDIF
ELSE:   movl    -4(%rbp), %eax
        subl    $1, %eax
        movl    %eax, %edi
        call    FACTORIAL
        imull   -4(%rbp), %eax
ENDIF:  leave
        ret
MAIN:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $3, -8(%rbp)
        movl    -8(%rbp), %eax
        movl    %eax, %edi
        call    FACTORIAL
        movl    %eax, -4(%rbp)
        movl    $0, %eax
        leave
        ret
```

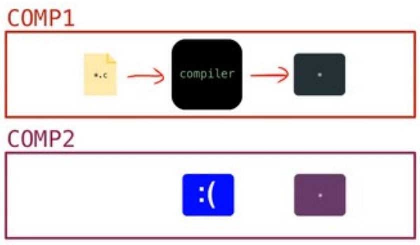


Let's ***run*** it to see what it does. Hitting the function call:

- we save all context into memory,
- allocate new space on top of it,
- execute the function code (which may involve calling more functions),
- and pop back down once it's done.

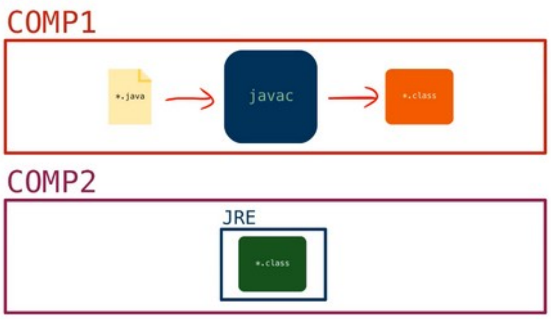
This makes it possible for functions to call other functions, or even themselves. You just push more memory, and pop back down when you're done.

13. So, that's how compilers work! They take your source code, and make machine code. But there's ***one problem***. If you compile your program on one computer, and then copy it over to another computer and try to run it, it might not work. If the new computer has a different operating system or has a different processor model, it probably uses different machine instructions.



So, if you want your program to be able to run on this new computer, you'd better be able to compile to *that* computer's machine code. And if your program's users might run your software on different platforms, unless you're distributing the source, you're gonna need to keep a copy of an executable for every platform you want to run on – *every single one!*

14. Some languages like ***Java*** sneak around this issue. Instead of machine code, *Java* gets compiled to an intermediate representation known as *bytecode*. And then the bytecode can get sent to other computers where it gets converted to that specific computer's machine code when the program is run via an interpreter. It's a bit of a compromise: you get better portability, though it's less efficient. But regardless, the language you write your code in is compatible with a wide variety of processors and operating systems.

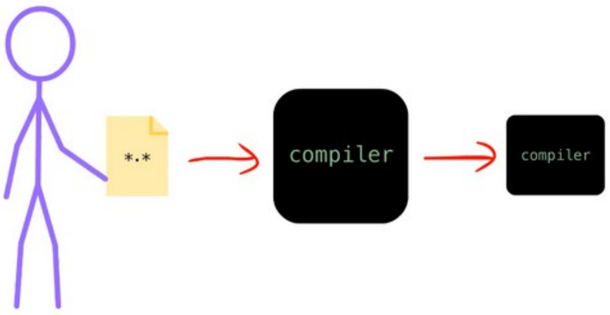


15. Can you imagine what it was like back in the day when computer programming meant putting assembly or even machine code onto ***punch cards***? Not only would you need to figure out the correct holes to punch for each instruction, you wouldn't even be able to use your program on a different computer model, because the other model expected you to punch different holes.

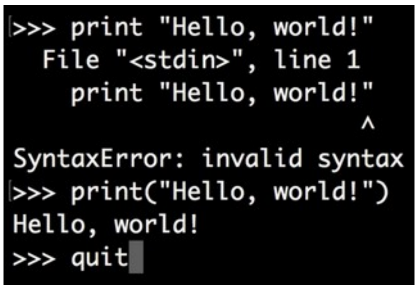
Technically, this card contains fortran code. But I mean, can you really tell?

Things are a little nicer these days. You just write a program, compile it, and test it with inputs and outputs… **if only it were that simple** …all thanks to the people who wrote the special program: the compiler.

16. But, remember that the ***compiler*** is a program itself. If people use compilers to develop programs, how was the *compiler* developed? Well, it was probably written and compiled in another language, or even in the same language, compiling a compiler with a previous version of itself. If we follow this chain backwards, at some point, we reach the origins of development tools: programs, written directly in machine code, that help you write other programs – literally automating part of the process of creating automation.



The ***history of computer languages*** is pretty complex. No wonder it took decades to get to where we are now! Remember that the next time you're writing code. We have all these beautiful things like syntax highlighting, static analysis, object-oriented programming, functional programming, libraries linkers, build tools, and debuggers. But it's still amazing that we can just tell our computers to follow our exact instructions… at the push of a button. Nope, wait… *there* we go: the *push* of a button. Programming isn't so hard…



```
>>> print "Hello, world!"
  File "<stdin>", line 1
    print "Hello, world!"
                        ^
SyntaxError: invalid syntax
>>> print("Hello, world!")
Hello, world!
>>> quit
```