Introduction

Jia Xiaodong

Last revised August 16, 2021

How are programs run? *

- Stored program computer
- In essence, just 1's and 0's
- Set up macros to ease the pain (assembly)
- Set up more macros (low level languages)
- Either compile it back for the computer to read, or get something to interpret it

Your laptop is a stored program computer. Most computers today are. They have somewhere where programs sit, and a processor reads them out to execute them. At the lowest level these are in 1's and 0's. This is also called *machine code*. It's what the machine understands and manipulates.

Unfortunately humans aren't machines and programming in that method is extremely painful so you can simply devise a scheme to translate commands to those 1's and 0's.

Yet those are still cumbersome, so you can set up human readable rules that will be translated to those instructions.

These instructions can either be *compiled* back down to machine code, or they can be *interpreted* by another program. Source is an interpreted language. So there is a program that runs your program. At the end of the chain however it is still machine code, so it is not magic.

Expressions

- Programs are made of sequences of *statements*.
- All statements are things such as
  - `const asd = 123;`
  - Blocks `{program}`,
  - Expressions
- And expressions are made of primitives like
  - `1; "hi"; true; ...`
  - Unary operators `-5; !true;`
  - Binary operators `5 - 2; 10 < 5; "a" === "b";`
  - Ternary operator `expr1 ? expr2 : expr3`
- Expressions produce results.

Refer to the language spec for detailed specs on the language. It written in a more formal syntax but it's not too difficult to get an idea of it's really talking about.

It is important that in most programming languages equality (as in 5 is equal to 5) is not expressed by =. Unfortunately historically = is given the role of assignment (think of it as ←). Hence `a=5` does not mean a **is equal to** 5, but a **gets the value of** 5. Equality is delegated to == or === most of the time.

Expressions always evaluate to something. For blocks, this result is the result of the last statement, or that of the first **`return`** statement encountered.

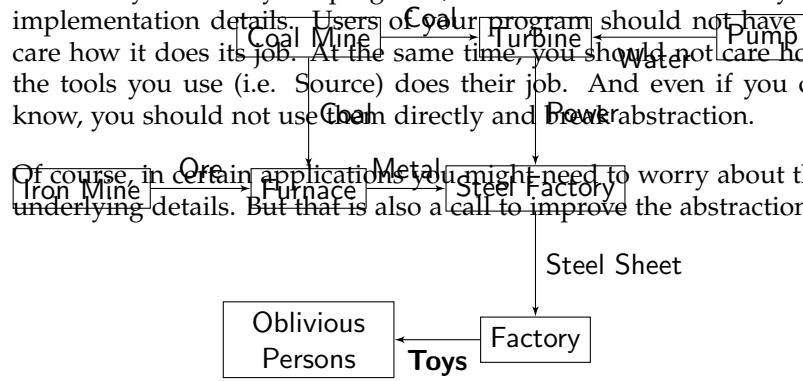
Abstraction

Abstraction, the wagon of progress.

The person drinking the water does not need to worry about how the water reached him. The person applying the theorem does not need to know the proof.

So when you write your programs, it should hide the underlying implementation details. Users of your program should not have to care how it does its job. At the same time, you should not care how the tools you use (i.e. Source) does their job. And even if you do know, you should not use them directly and break abstraction.

Of course, in certain applications you might need to worry about the underlying details. But that is also a call to improve the abstraction.



Functions

Example

What does this do?

```
function norm(x, y) {
    return math_sqrt(x * x + y * y);
}
```

Good to know

- `display`
- `math_PI`, `math_log`, ...

The `norm` function evaluates the Euclidean norm (or in more layman terms, that Pythagoras theorem thing). It would be good if we could type `norm(3, 4)` instead of `math_sqrt(3 * 3 + 4 * 4)`. Not only is it just for convenience — do you know how to perform `math_sqrt` if it isn't there?

Conditionals

The **if–else** statement:

```
if (expr) {
    program;
} else if (expr2) {
    program2;
} else {
    program3;
}
```

Programs require flow control to redirect execution. The **if–else** statement is one form of flow control.

This does exactly what it says it does. `expr` is evaluated first, if it is true, then `program` is evaluated and the entire **if–else** statement evaluates to that result. Otherwise, `expr2` is evaluated next, and if that is true, then `program2` is evaluated, and so on...

Note that unlike other languages, Source insists that **else** *always* follows **if**. **else if** is optional.

Ternary operator

```
predicate ? consequent : alternative
```

Examples

- `5 < 2 ? 10 : 100;`
- `"a" < "b" ? 1 : 2;`
- `"a" < "A" ? 1 : 2;`

The ternary operator is the only ternary operator.If `predicate` evaluates to true, then the operator evaluates to `consequent`. Otherwise it evaluates to `alternative`.

For the reason why `"a"` is larger than `"A"`, it is partly due to how these characters are represented. In most modern systems they are represented with UTF-8 (Unicode). Search for the list of unicode characters up on Wikipedia and find the section on ASCII, and it will show you why this result is given. (ASCII is another widespread character encoding scheme, older and simpler than Unicode).

Short circuiting

```
predicate ? consequent : alternative
```

Example

What does this do?

```
1 === 2 && display("No")
```

Good to know

Some operators are also lazy!

```
1 === 2 ? display("No") : 1;
```

The principle of short-circuiting operators is simply this: if, during any point of evaluation, you know for sure what the result will be, then there is no point continuing. What this also means is that certain side-effects caused by evaluation may not occur. The given example shows this in action.

Though this may seem contrived, often times this may catch you off guard! It may be dangerous if you don't know this exists. "Be the change you want to see in the world" — don't rely on a comparison to do things other than comparing!

Modulo

- We can do + - * /
- New operator: %