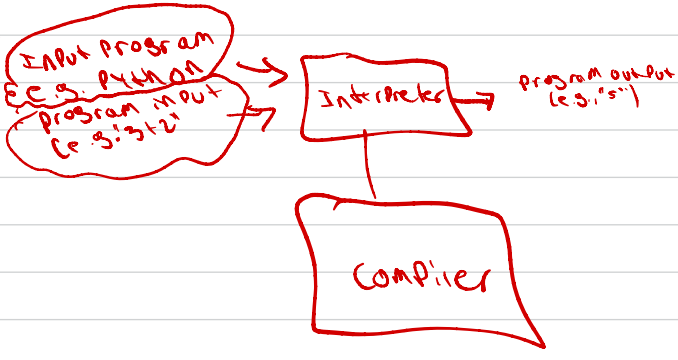
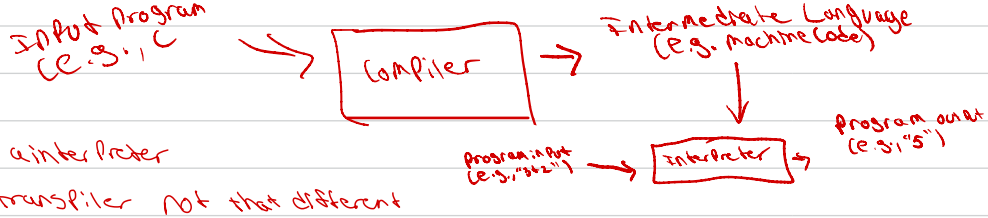
How Programming languages run?

Interpreted style



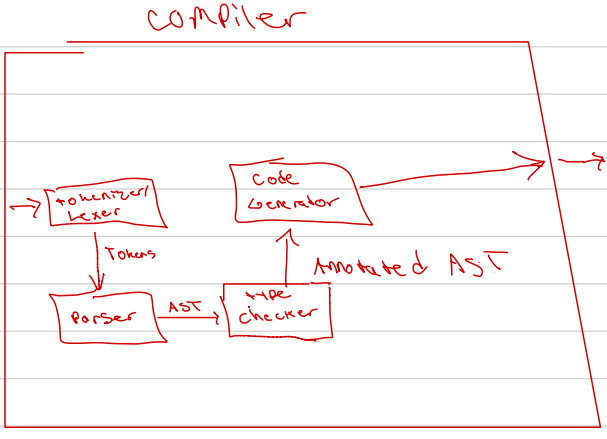
Compiled style

C example (machine code)



- processor is interpreter
- transpile/transpiler not that different from a compiler

Compilers



Possible tokens:
if
while
for

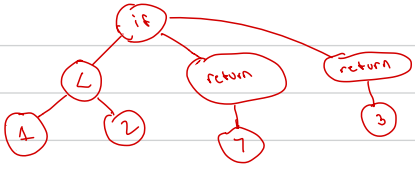
- What is a token?

individual tokens are individual words,

Parser: group tokens together & form like a data structure (tree)

Parser output: Abstract syntax tree (AST)

```
if (1<2) {
    return 7;
} else {
    return 3;
}
```



Grammar (BNF / Backus-Naur Form)
Context free grammar

| = or

digit ::= '0' | '1'

number ::= digit | digit number // BNF form
number ::= digit* // EBF form - * means 0 or more of the thing
expression ::= number | expression + expression

example numbers:

0
1
01
1
101

example expressions:

1101
101 + 110
(1+111) + (01+10)

Language design:

- Integers & Booleans
- Declare & initialize variables
- Perform typical arithmetic/ logical operations

var is a variable

type ::= 'int' | 'bool'
vardec ::= '(' 'vardec' type var expression ')'
expression ::= num | 'true' | 'false' |
'(' op expression expression ')'

loop ::= '(' 'while' expression statement ')'
assign ::= '(' '=' var expression ')'
statement ::= vardec | loop | assign

int x = 0;
while (x < 10) {
x = x + 1;
}

op ::= '+' | '-' | '&&' | '||' | '<'
program ::= vardec

(vardec int x 0)
(while (< x 10)
(= x (+ x 1)))

(vardec int x 7)
(vardec bool y true)
(vardec int a (+ 12))
(vardec bool b (&& false true)

#MyLang

grammar dentition

★ Language design:

- Integers & Booleans
- Declare & initialize variables
- Perform typical arithmetic/ logical operations

var is a variable

type ::= 'int' | 'bool'

vardec ::= '(' 'vardec' type var expression ')'

expression ::= num | 'true' | 'false' | '(' op expression expression ')'

loop ::= '(' while expression statement ')'

assign ::= '(' '=' var expression ')'

statement ::= vardec | loop | assign

Jacoco - Library

Shows what part of the code is getting tested