

# Programming Languages



# Interpreters and Compilers



# Interpreters



- Some underlying language X [e.g., Python]
- Used to execute code in a language Y [e.g., Scheme]
- Why?
  - More languages = better
  - (Lets you interpret new languages)

```
scm> (define (square x) (* x x))
scm> (accumulate + 0 5 square) ; 0 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2
55
scm> (accumulate + 5 5 square) ; 5 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2
60
```

# Compilers

- Take code in language Y and convert it to machine code
  - This is called an “executable”
- Can be run stand-alone from that point on
- How all languages eventually work



# Compiler Pros and Cons

- Compiler Pros
  - Faster to run the code
  - Don't need to send an interpreter to your users
    - just the executable that the compiler produces
- Interpreter Pros
  - Easier to implement

But is the real world this simple???

No

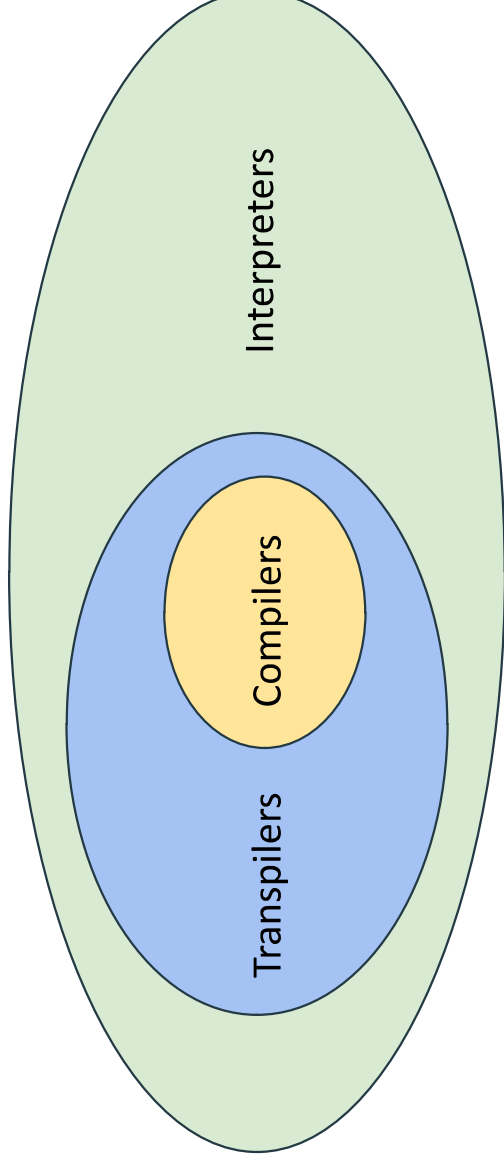
# Transpilers

- What about a *transpiler*
- Converts code from one language to another
- E.g., convert something like “(define x 2)” to “x = 2”\*
- Is this a compiler or interpreter?

\* actually something like `x = 2; _result = "x"`

# Transpilers / Compilers / Interpreters

- Compilers are obviously Transpilers
- Transpilers are Interpreters





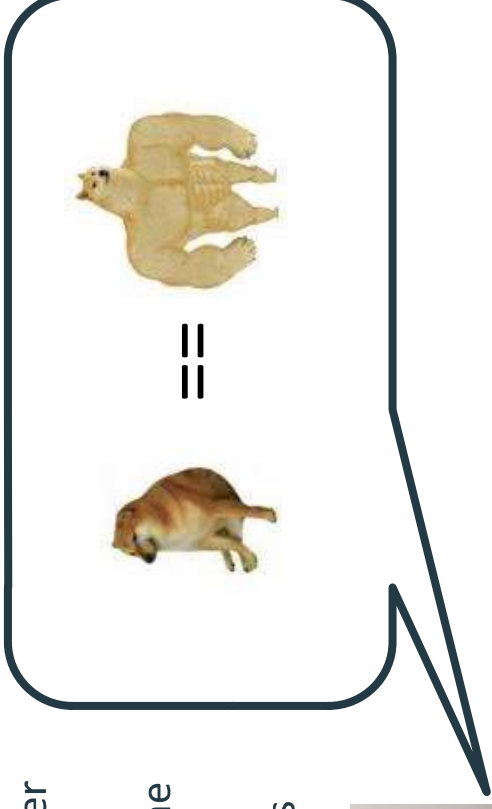
But is the real world so simple???

No

(the answer is never yes)

# Interpreters are Compilers???

- You can use an interpreter as a compiler pretty easily!!
- Just send the entire interpreter with the code, now you have an “executable”
- This is one of the Futamura Projections



# How Does Python Work?



# It's interpreted!

- `python3` is an interpreter written in the programming language C
  - C is a compiled language
- python frames are allocated as objects in C
  - It is handled like a dictionary
- there is a `python_eval` function written in C →

```
PyObject *
_PyEval_EvalCode(PyThreadState *tstate,
PyObject *co, PyObject *globals, PyObject *locals,
PyObject *const_args, Py_ssize_t argcount,
PyObject *const_kwnames, PyObject *const_kwargs,
Py_ssize_t kwcount, int kwstep,
PyObject *const_defs, Py_ssize_t defcount,
PyObject *kwdefs, PyObject *closure,
PyObject *name, PyObject *qualname)
{
    assert(is_tstate_valid(tstate));

    PyCodeObject *co = (PyCodeObject*)_co;

    if (!name) {
        name = co->co_name;
    }
    assert(name != NULL);
```

## How does **is** work?

- C has an **is** operator, confusingly called **==**
- Thus Python **is** can be directly implemented in terms of C's **==**

## How does + work?

- Can operator called +
- But it's bad
- In C,  $1073741824 + 1073741824 == -2147483648$ 
  - The integer “overflows” so values that are too big become negative
- So Python's + is implemented as a *function*
  - Basically it does the addition algorithm you learned in elementary school

## How does **if** work?

- In terms of C's if!

# Programming Language Features





# Variables

- More or less all languages have variables
- But not all do!
- In MIPS here's how to write the squared\_distance function
  - MIPS is an assembler language, very close to what the machine reads

```
squared_dist:
mul $t0 $a0 $a0
mul $t1 $a1 $a1
add $v0 $t0 $t1
jr $ra

# $t0 = first argument ** 2
# $t1 = second argument ** 2
# set return value to $t1 + $t2
# return
```

# Functions

- Not all languages have functions
- These are really a convenience
- Technically you can write anything in python without any functions

# Recursion

- Technically, you can have functions without recursion!!
- (This hasn't been a thing since the 70s)
- The original FORTRAN always reuses function frames, so it cannot handle recursion at all
  - No concept of multiple frames of the same function being open at the same time

# Memory Management

- Python and Scheme both have what is known as “memory management”
- Code on the right doesn’t “leak memory”
  - When the hailstone\_sum frame closes elements can be deallocated
  - They can be removed from memory
- In languages like C you would have to do that manually

```
def hailstone_sum(x):  
    elements = [x]  
    while x > 1:  
        if x % 2 == 0:  
            x //= 2  
        else:  
            x = 3 * x + 1  
        elements.append(x)  
    return sum(elements)
```

# Type Safety

- Python and Scheme both have what is known as “type safety”
- For example, “2” + 3 gives an error
- Some languages try to do something “reasonable” instead, like “23”
  - Java/JS do this
- C, on the other hand, *adds the memory location “2” is at to 3*
  - This is called *type unsafe*

# Exceptions

- Control flow that can break the bounds of a function
- Usually for error handling
- Python uses `raise`

# Macros

- Scheme has macros, but as an alternative to functions
- TeX only has macros!!
  - Common typesetting system, and the basis of LaTeX
  - Kinda an HTML for PDFs
- C has “macros” but they are just basic text substitution
  - `#define until(x) while(not (x))`

# Now For The Weird Stuff

We didn't cover everything in this class! :(



# Coroutines

- When two functions call trade control off to each other
- Python does this with generators, which can actually send data both ways

```
def counter():  
    value = 0  
    while True:  
        value += (yield value)  
  
c = counter()  
assert next(c) == 0  
assert c.send(2) == 2  
assert c.send(5) == 7  
assert c.send(2) == 9
```

# Lazy Evaluation

- In most languages, we have what's known as "eager evaluation"
  - Evaluate expressions as they come up
- In some languages, we can have "lazy evaluation"
  - Evaluate expressions when needed
- Tail recursion is kinda a (very limited) version of this
- Streams are a more fleshed-out version of this

# Lazy Evaluation (errors)

- This leads to interesting error behavior
- In this example
  - `error` is like `python's raise`
  - `!!` is like python's list indexing

```
Prelude> let x = [2, error "hi", 3]
Prelude> x !! 0
2
Prelude> x !! 2
3
Prelude> length x
3
Prelude> x !! 1
*** Exception: hi
CallStack (from HasCallStack):
  error, called at <interactive>:8:13 in interactive:Ghci8
Prelude> x
[2,*** Exception: hi
CallStack (from HasCallStack):
  error, called at <interactive>:8:13 in interactive:Ghci8
Prelude>
```

# Lazy Evaluation (infinite data)

- Here's fib-stream in Haskell!
  - `:` is `cons`
  - `tail` is `cdr`
  - `zipWith` is a HOF that is like `zip` but uses a 2-argument function rather than putting the items in pairs
- This is like streams in scheme, but *everything* is lazy
- Can't handle effects very well

```
Prelude> fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
Prelude> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

# Parallelism

- You can run multiple pieces of code at the same time!
- Saves time!
- Allows multiple programs to happen at the same time!

```
from multiprocessing import Pool

def fancy_math(x, y=87239487239084701827):
    for i in range(x):
        x = x ** 2 % y
    return x

inputs = list(range(4000))

# this takes 2.62s on my server
outputs = [fancy_math(x) for x in inputs]

# this takes 0.46s on my server
outputs = Pool().map(fancy_math, inputs)
```

# Conclusion



# Infinite Diversity in Infinite Combinations

- There are thousands of programming languages
- Each unique in their own way
- Lots of language features that slowly make their way into the big languages

Table of programming languages

Language	Year	Paradigm	Typing	Memory	Concurrency	Web	Mobile	Game	AI	Other
1. C	1972	Procedural	Static	Manual	Single-threaded	Yes	Yes	Yes	Yes	Yes
2. C++	1985	Procedural	Static	Manual	Single-threaded	Yes	Yes	Yes	Yes	Yes
3. C#	2000	Object-oriented	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
4. Java	1995	Object-oriented	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
5. JavaScript	1995	Object-oriented	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
6. Python	1990	Object-oriented	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
7. Ruby	1995	Object-oriented	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
8. PHP	1995	Object-oriented	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
9. Perl	1987	Object-oriented	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
10. Haskell	1990	Functional	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
11. Lisp	1958	Functional	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
12. Prolog	1972	Logic	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
13. Fortran	1957	Procedural	Static	Manual	Single-threaded	Yes	Yes	Yes	Yes	Yes
14. COBOL	1960	Procedural	Static	Manual	Single-threaded	Yes	Yes	Yes	Yes	Yes
15. Basic	1964	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
16. Pascal	1970	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
17. Modula-2	1982	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
18. Ada	1982	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
19. Eiffel	1988	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
20. D	2001	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
21. Go	2007	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
22. Rust	2010	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
23. Swift	2014	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
24. Kotlin	2012	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
25. Scala	2009	Object-oriented	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
26. F#	2005	Object-oriented	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
27. R	1999	Statistical	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
28. MATLAB	1984	Statistical	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
29. Octave	1988	Statistical	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
30. REXX	1985	Procedural	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
31. AWK	1977	Text processing	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
32. Sed	1976	Text processing	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
33. Vim	1988	Text editing	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
34. Emacs	1976	Text editing	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
35. TeX	1985	Typesetting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
36. LaTeX	1985	Typesetting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
37. Lua	1993	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
38. Perl 6	2005	Object-oriented	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
39. Raku	2015	Object-oriented	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
40. Nim	2011	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
41. Crystal	2014	Object-oriented	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
42. Zig	2016	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
43. V	2010	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
44. Duktape	2010	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
45. LuaJIT	2012	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
46. ChakraCore	2012	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
47. V8	2008	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
48. SpiderMonkey	2003	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
49. JavaScriptCore	2003	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
50. WebKit	2003	Rendering	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
51. Qt	1995	GUI	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
52. GTK+	1998	GUI	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
53. SDL	1998	Game	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
54. SFML	2004	Game	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
55. Unreal Engine	1998	Game	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
56. Unity	2005	Game	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
57. Godot	2013	Game	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
58. OpenFL	2014	Game	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
59. Flix	2015	Functional	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
60. Futhur	2016	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
61. Mojo	2018	Procedural	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
62. PyPy	2006	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
63. IronPython	2008	Scripting	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
64. Jython	2003	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
65. MicroPython	2012	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
66. CircuitPython	2016	Scripting	Dynamic	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
67. Arduino	2001	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
68. Raspberry Pi	2012	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
69. BeagleBone Black	2011	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
70. OpenMoko	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
71. OpenMoko Freerunner	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
72. OpenMoko Libretto	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
73. OpenMoko N100	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
74. OpenMoko N110	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
75. OpenMoko N120	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
76. OpenMoko N130	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
77. OpenMoko N140	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
78. OpenMoko N150	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
79. OpenMoko N160	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
80. OpenMoko N170	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
81. OpenMoko N180	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
82. OpenMoko N190	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
83. OpenMoko N200	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
84. OpenMoko N210	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
85. OpenMoko N220	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
86. OpenMoko N230	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
87. OpenMoko N240	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
88. OpenMoko N250	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
89. OpenMoko N260	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
90. OpenMoko N270	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
91. OpenMoko N280	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
92. OpenMoko N290	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
93. OpenMoko N300	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
94. OpenMoko N310	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
95. OpenMoko N320	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
96. OpenMoko N330	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
97. OpenMoko N340	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
98. OpenMoko N350	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
99. OpenMoko N360	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes
100. OpenMoko N370	2007	Microcontroller	Static	Automatic	Single-threaded	Yes	Yes	Yes	Yes	Yes

# Python and Scheme are Pretty Advanced

- You've learned most of the common programming language features in this class
- Python incorporates many features other languages don't
- Scheme has many of the rest



# New Languages to Learn

- I'd suggest the following to learn more features and ways of thinking
  - Haskell -- lazy evaluation and functional programming
  - Prolog -- declarative programming
  - Go -- parallelism
  - Java -- large project management (you'll do this in 61B)
  - C -- close-to-the-machine programming (you'll do this in 61C)