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

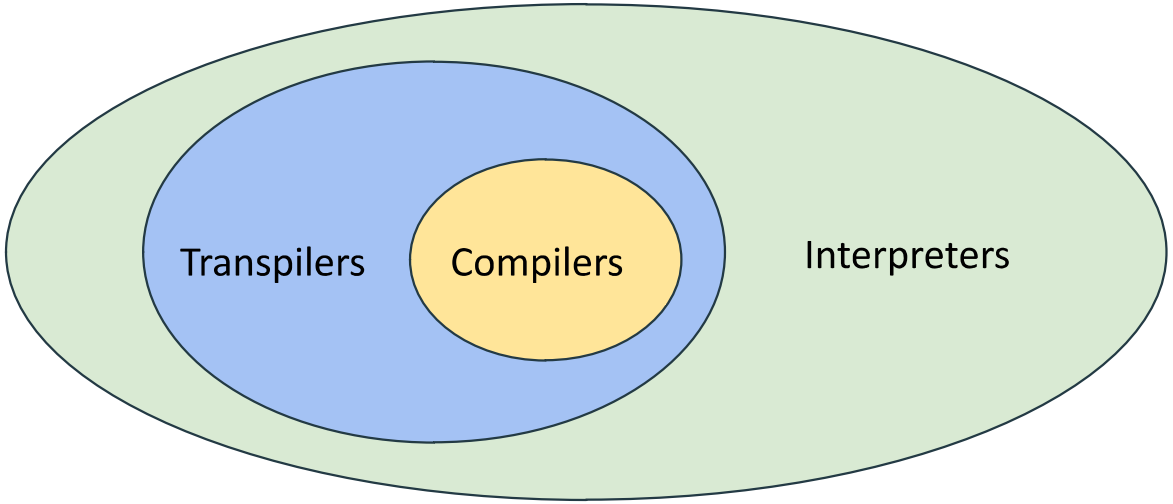
Transpilers

- 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



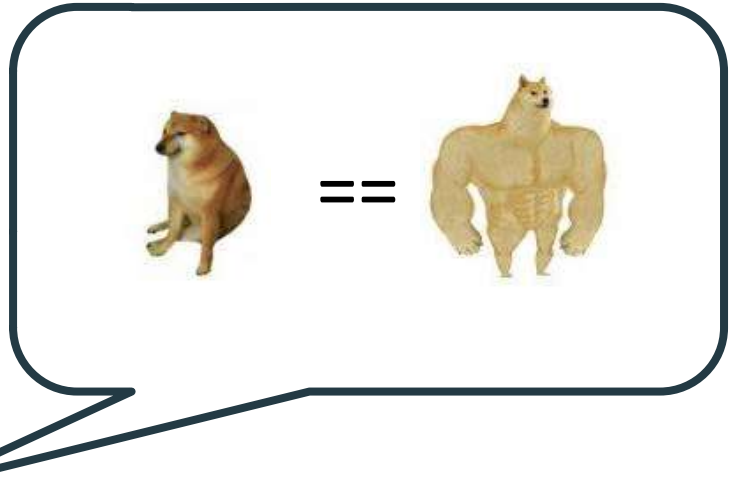
But is the real world so simple???

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?

- C an 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          # $t0 = first argument ** 2
mul $t1 $a1 $a1          # $t1 = second argument ** 2
add $v0 $t0 $t1          # set return value to $t1 + $t2
jr $ra                   # 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

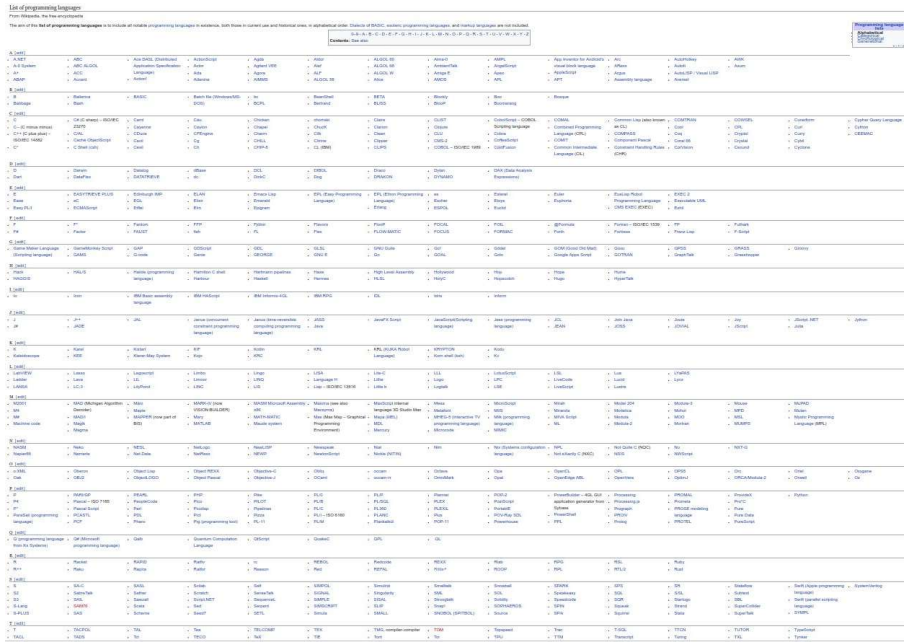
- You can run multiple pieces of code at the same time!
- Saves time!
- Allows multiple programs to happen at the same time!

```python
from multiprocessing import Pool

def fancy_math(x, y=872394872390847018 27):
    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



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)