



# New Relic®

Inside (C)Python  
Tom Lee @tglee

OSCON 2012  
19th July, 2012



# Overview



- About me!

- Implemented unified try/except/finally for Python 2.5 (Georg Brandl's PEP 341)
- Support for compiling ASTs to bytecode from within Python programs for 2.6
- Wannabe compiler nerd.

- Terminology & brief intro to compilers:

- Grammars, Scanners & Parsers
- Parse Trees & Abstract Syntax Trees
- General architecture of a bytecode compiler

- How to add new syntax to Python:

- Python compiler architecture & summary
- Case study in adding a new keyword
- Demo!



# “Python” vs. “CPython”

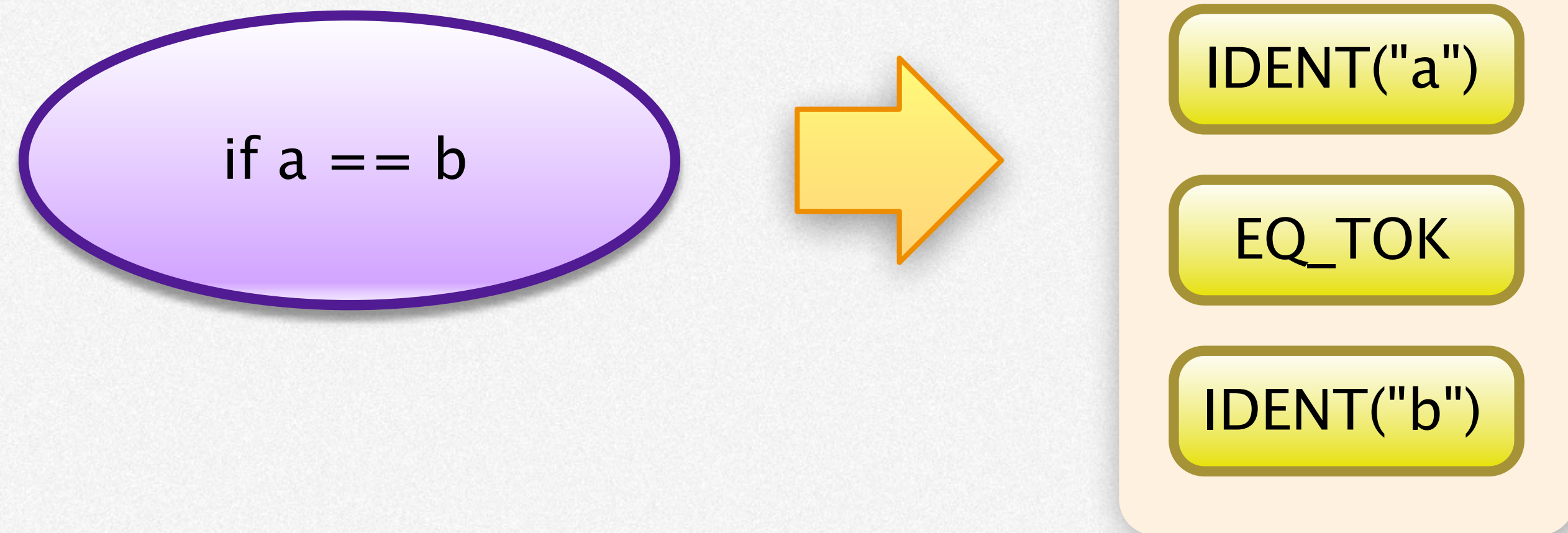


- When I say “Python” in this presentation, I mean **CPython**.
- Other Python implementations:
  - Jython
  - PyPy
  - IronPython
  - Stackless Python
  - More...
- Many of the concepts here are likely applicable, but the details may differ wildly
- Refer to the slides for my **Inside PHP** presentation to see how things may differ.
- Python 3.x



# Compilers 101: Scanners

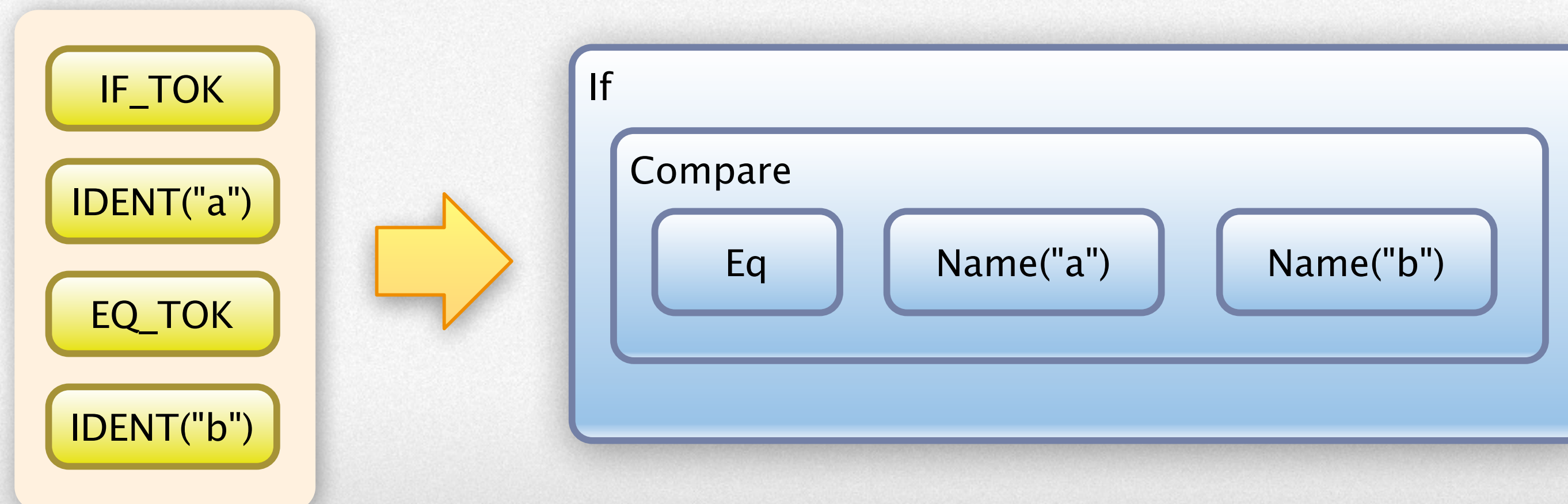
- Or **lexical analyzers**, or **tokenizers**
- **Input:** raw source code
- **Output:** a stream of **tokens**





# Compilers 101: Parsers

- **Input:** a stream of tokens from the scanner
- **Output** is implementation dependent
- The goal of a parser is to structure to the sequence of tokens from the scanner.
- Parsers are frequently **automatically generated** from a grammar/DSL
  - See **parser generators** like Yacc/Bison, ANTLR, etc. or e.g. **parser combinators** in Haskell, Scala, ML.





# Compilers 101: Context-free grammars



- Or simply “**grammar**”
- A grammar describes the complete syntax of a (programming) language.
- Usually expressed in Extended Backus-Naur Form (EBNF)
  - Or some variant thereof.
- Variants of EBNF used for a lot of DSL-based **parser generators**
  - e.g. Yacc/Bison, ANTLR, etc.



# Compilers 101: Parse Tree

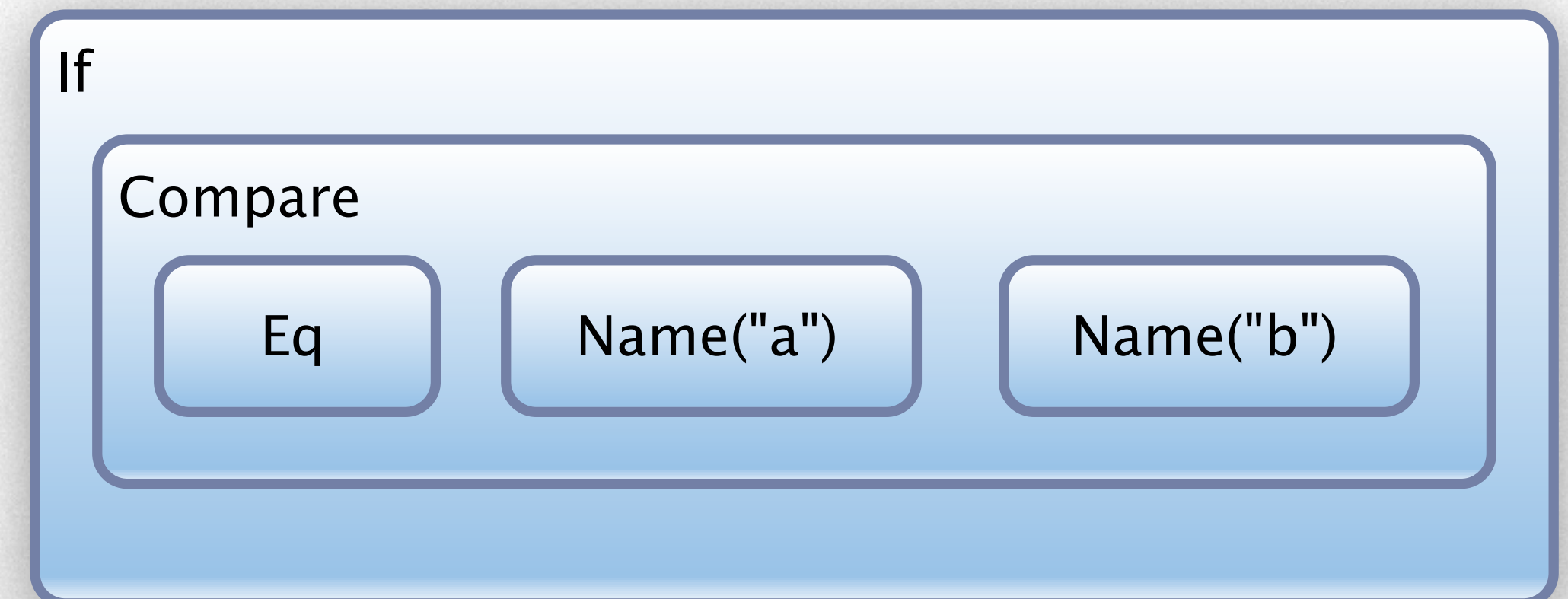
- Or **concrete syntax tree**.
- “... *an ordered, rooted tree that represents the [full] syntactic structure of a string according to some formal grammar.*”
  - [http://en.wikipedia.org/wiki/Parse\\_tree](http://en.wikipedia.org/wiki/Parse_tree)
- Python’s parser constructs a **parse tree** based on **Grammar/Grammar**
  - Must be manually converted to **Abstract Syntax Tree** before we do anything fun.



# Compilers 101: Abstract Syntax Tree

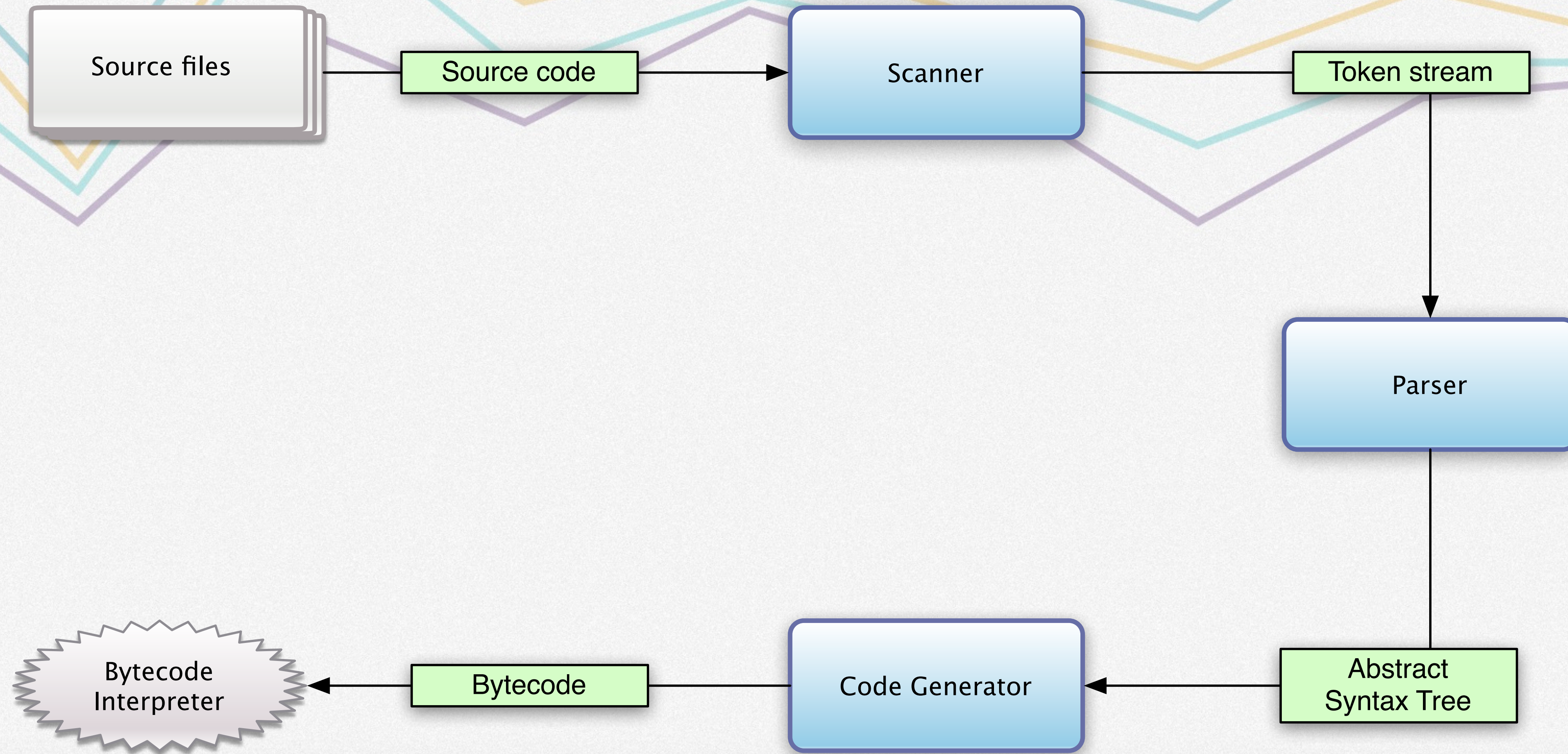
- **AST**

- Rooted tree data structure.
- In-memory, intermediate representation of the input (source) program.
  - “Abstract” in that it doesn’t necessarily include every detail that appears in the real syntax.
- A **code generator** can traverse an AST and emit target code.
  - e.g. Python byte code
- Certain optimizations are also convenient at the AST level.





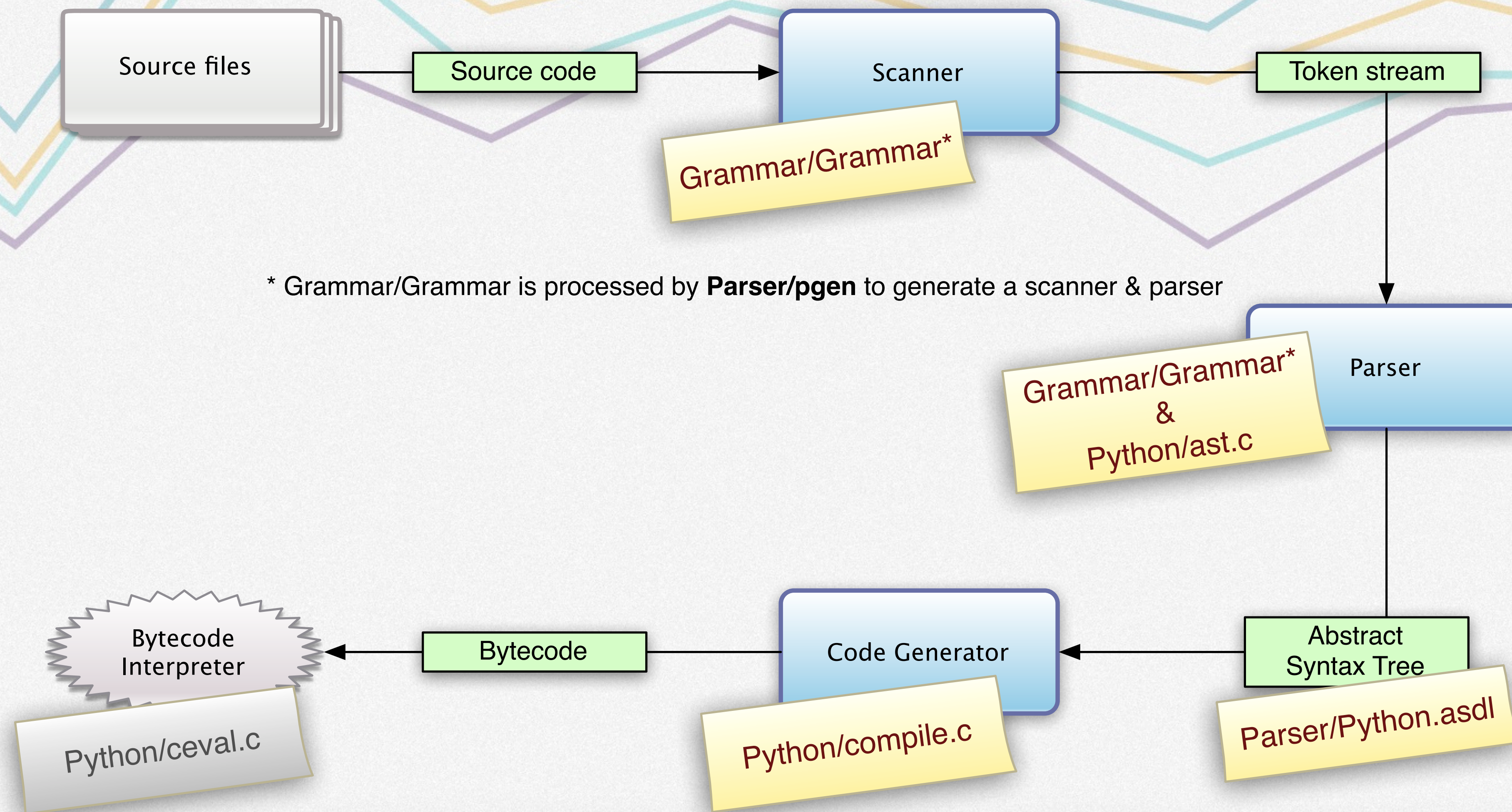
# Generalized Compiler Architecture\*



\* Actually a generalized \*bytecode\* compiler architecture



# Generalized \*Python\* Compiler Architecture





# Case Study: The “unless” statement



```
will_give_you_up = False
```

```
unless will_give_you_up:  
    print("Never gonna give you up.")
```

```
will_let_you_down = True
```

```
unless will_let_you_down:  
    print("Letting you down :'(")
```

```
-- output --
```

```
Never gonna give you up.
```

It's basically  
“if not ...”



# Adding the “unless” construct to (C)Python



1. Describe the syntax of the new construct
2. Add new AST node(s) if necessary
3. Write a parse tree to Abstract Syntax Tree (AST) transform
4. Emit bytecode for your new AST node(s)



# Before you start...

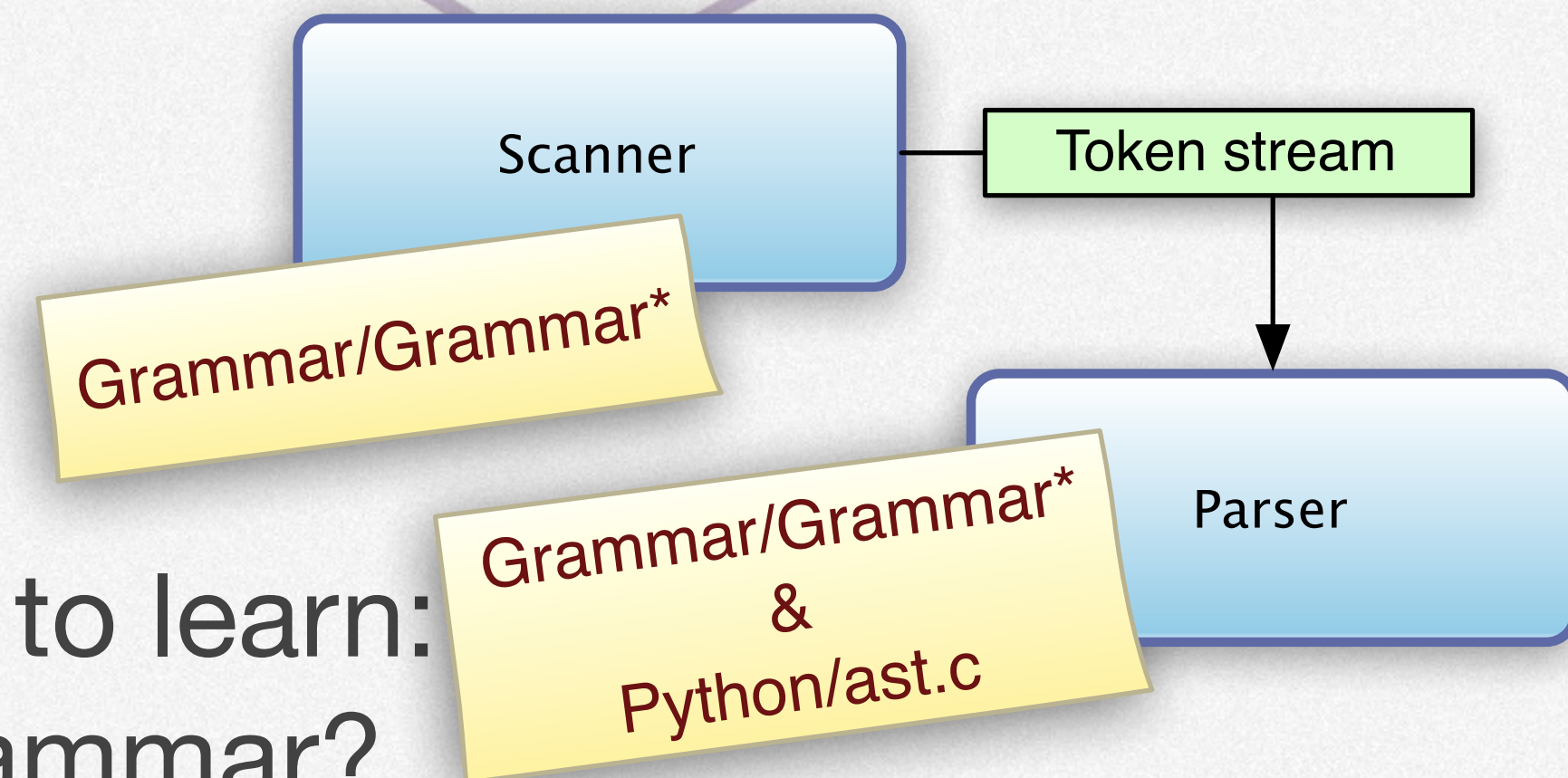


- You'll need the usual gcc toolchain, GNU Bison, etc.
  - **Debian/Ubuntu** apt-get install build-essential
  - **OSX** Xcode command line tools should give you most of what you need.
- Grab the Python code from Mercurial
  - I'm working with the stable Python 3.2 branch
    - Only because tip was breaking in weird & wonderful ways at the time of rehearsal!
  - The steps described here should also work for any version of Python  $\geq 2.5.x$ 
    - Beware of cyclic dependencies in the build process when modifying the grammar.



# 1. Describe the syntax of the new construct

- Or: describe how your new language construct “looks” in source code
- Modify the grammar in **Grammar/Grammar**
  - EBNF-ish DSL
  - Used by **Parser/pgen**, Python’s custom parser generator, to generate the C code that drives Python’s **parser** and **scanner**.
- Riffing on existing constructs is a great way to learn: how does an **if** statement look like in the grammar?
- Let’s specify the syntax of **unless**





## 2. Add new AST node(s) if necessary

- **Parser/Python.asdl**

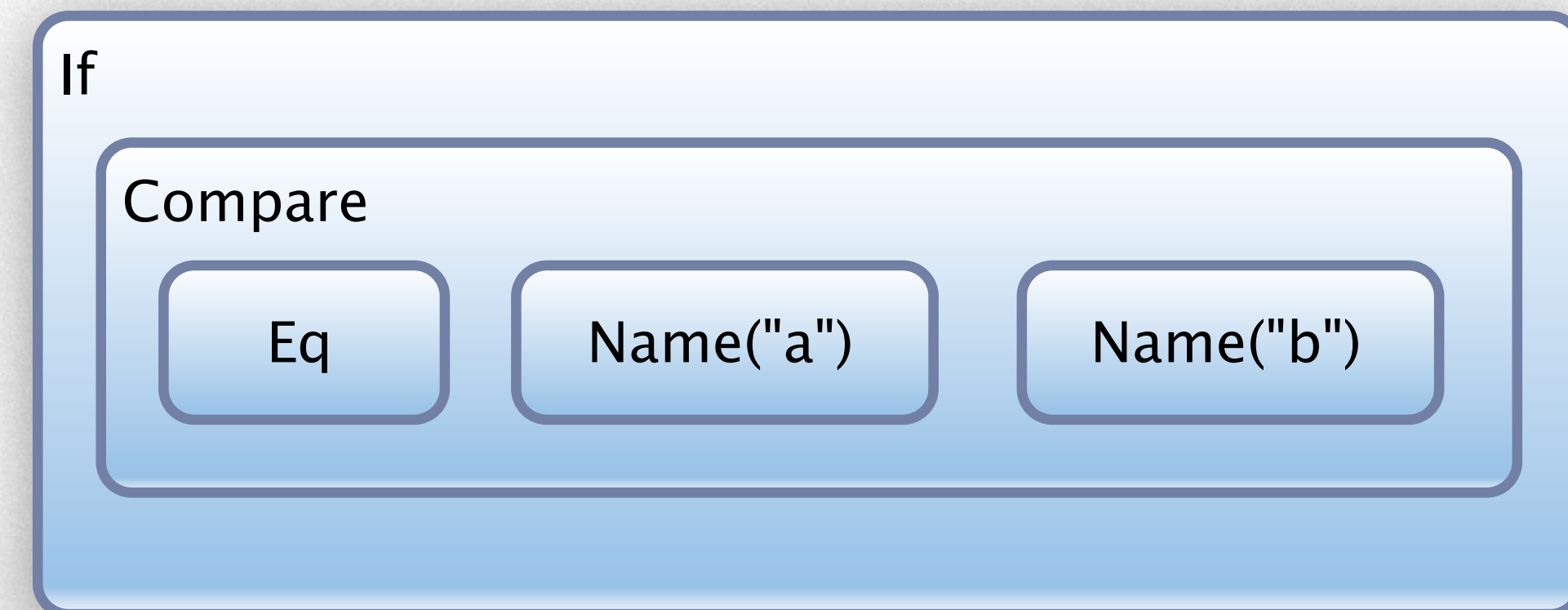
- *Zephyr Abstract Syntax Definition Language* -- a DSL for describing ASTs.
- <http://asdl.sourceforge.net/>
- The Python build process generates C code describing the AST.
  - See **Parser/asdl\_c.py** and **Parser/asdl.py** for details of how this code is generated.

- Can you represent your new Python construct using existing AST nodes?

- Great! Skip this step.

- Let's go ahead and add an **Unless** node anyway.

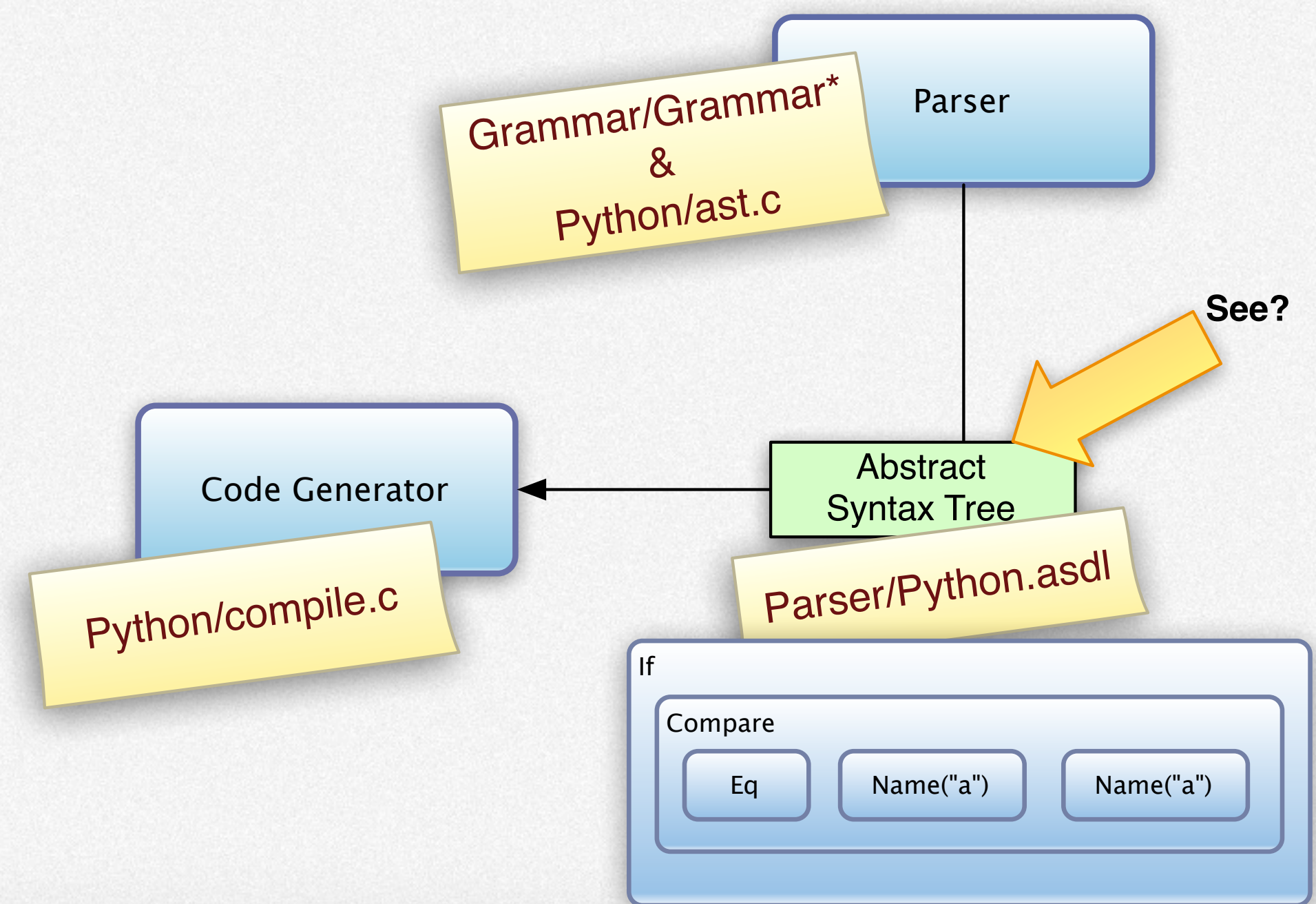
- Demonstration using **If** and **Not** nodes if we have time ...





# 3. Parse tree to AST transformation

- Recall the parser that is generated from Grammar/Grammar
- The generated parser constructs a **parse tree** without any effort on our part ...
- ... BUT! earlier we saw that the code generator takes an **AST** as input.
- We need to **manually transform** the parse tree into an AST.
- **Python/ast.c**
- Again, look to the existing **if** statement for inspiration.





## 4. Emit bytecode for the new AST node

- **This step is only necessary if you added a new AST node in step 2.**
- At this point our AST is ready to compile to bytecode.
- Traverse the tree, emitting bytecode as we go.
- What bytecode instructions do we need to use?
  - What does the bytecode for a simple **if** statement look like? How would we modify it?



# Intermission: Python VM + bytecode

- Python has a **stack based virtual machine**.
- Values get pushed onto a data stack.
- Opcodes manipulate the stack.
  - e.g. binary op: pop two values, compute result, push new value.
- Control flow using unconditional & conditional jumps.

```
data = []
```

```
def push(x):  
    data.append(x)
```

```
def pop():  
    return data.pop()
```

```
def add():  
    push(pop() + pop())
```

```
def sub():  
    push(pop() - pop())
```

```
push(5)      # [5]  
push(6)      # [5, 6]  
add()        # [11]
```



# Bytecode for “if x == y: ...”



LOAD\_NAME x

LOAD\_NAME y

COMPARE\_OP PyCmp\_EQ

POP\_JUMP\_IF\_FALSE end

...

...

...

x == y

if ...: ...

end



# Bytecode for “if x == y: ...”



LOAD\_NAME x

LOAD\_NAME y

COMPARE\_OP PyCmp\_EQ

POP\_JUMP\_IF\_FALSE end

...

...

...

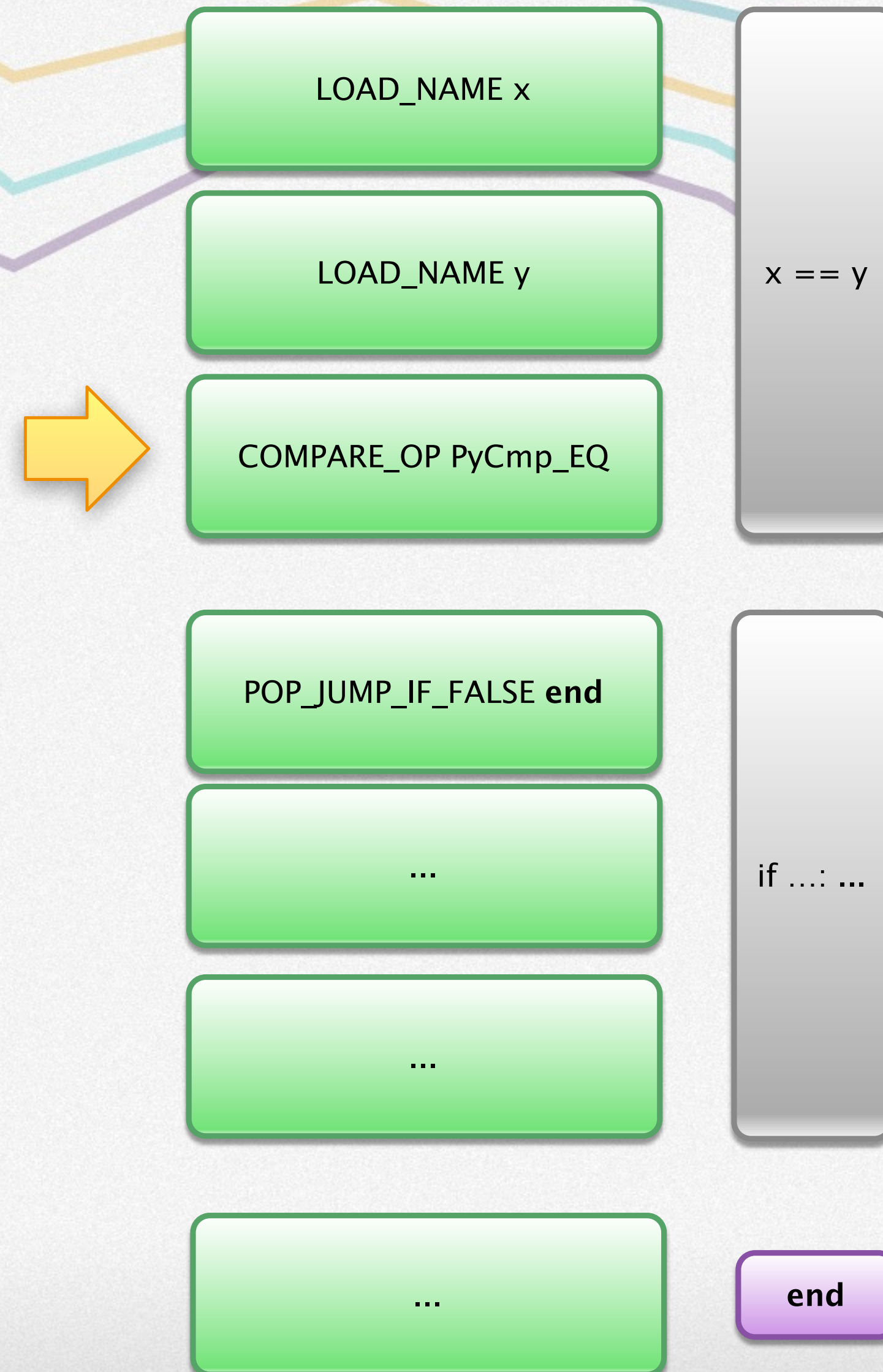
x == y

if ...: ...

end

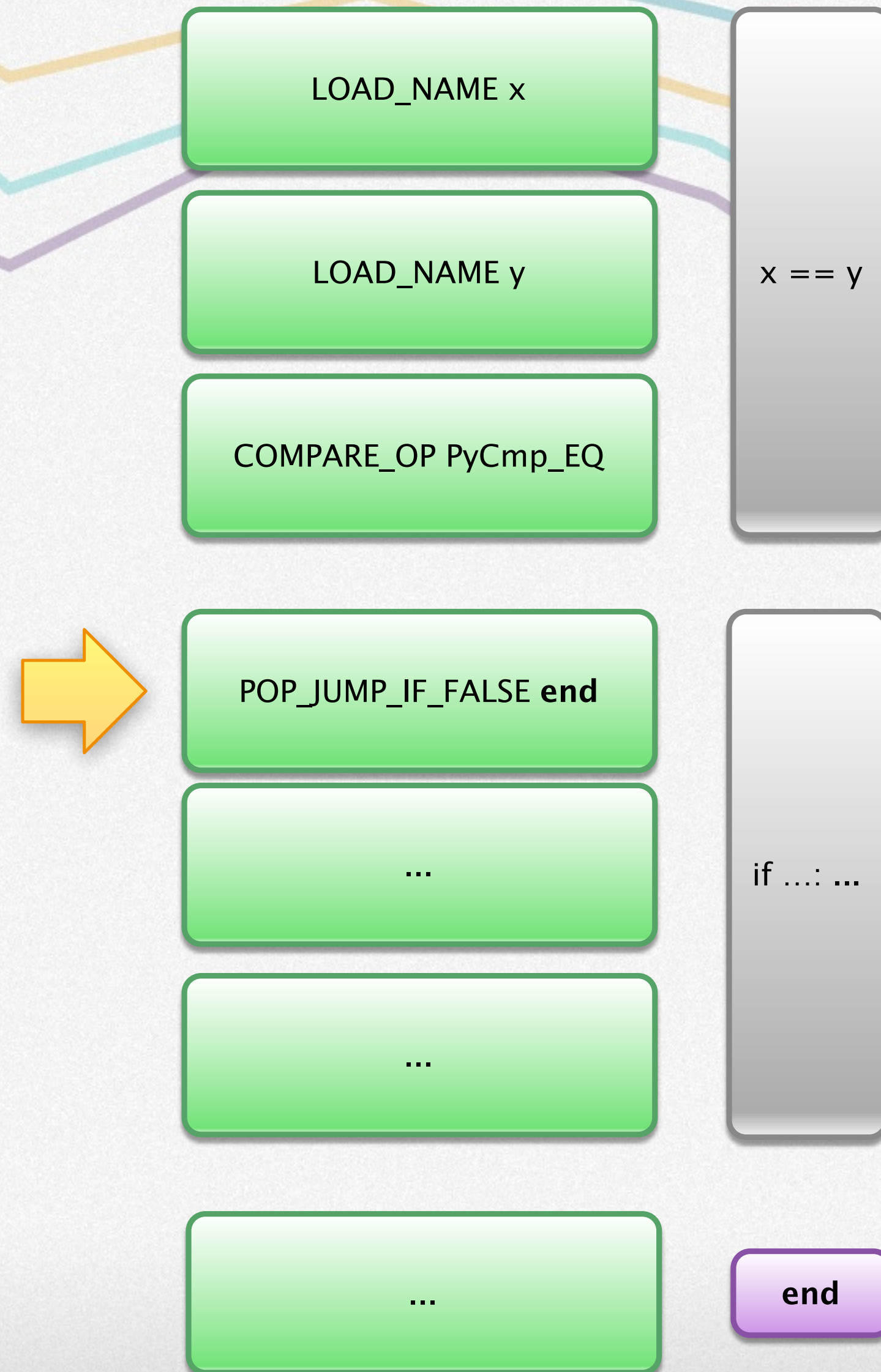


# Bytecode for “if x == y: ...”



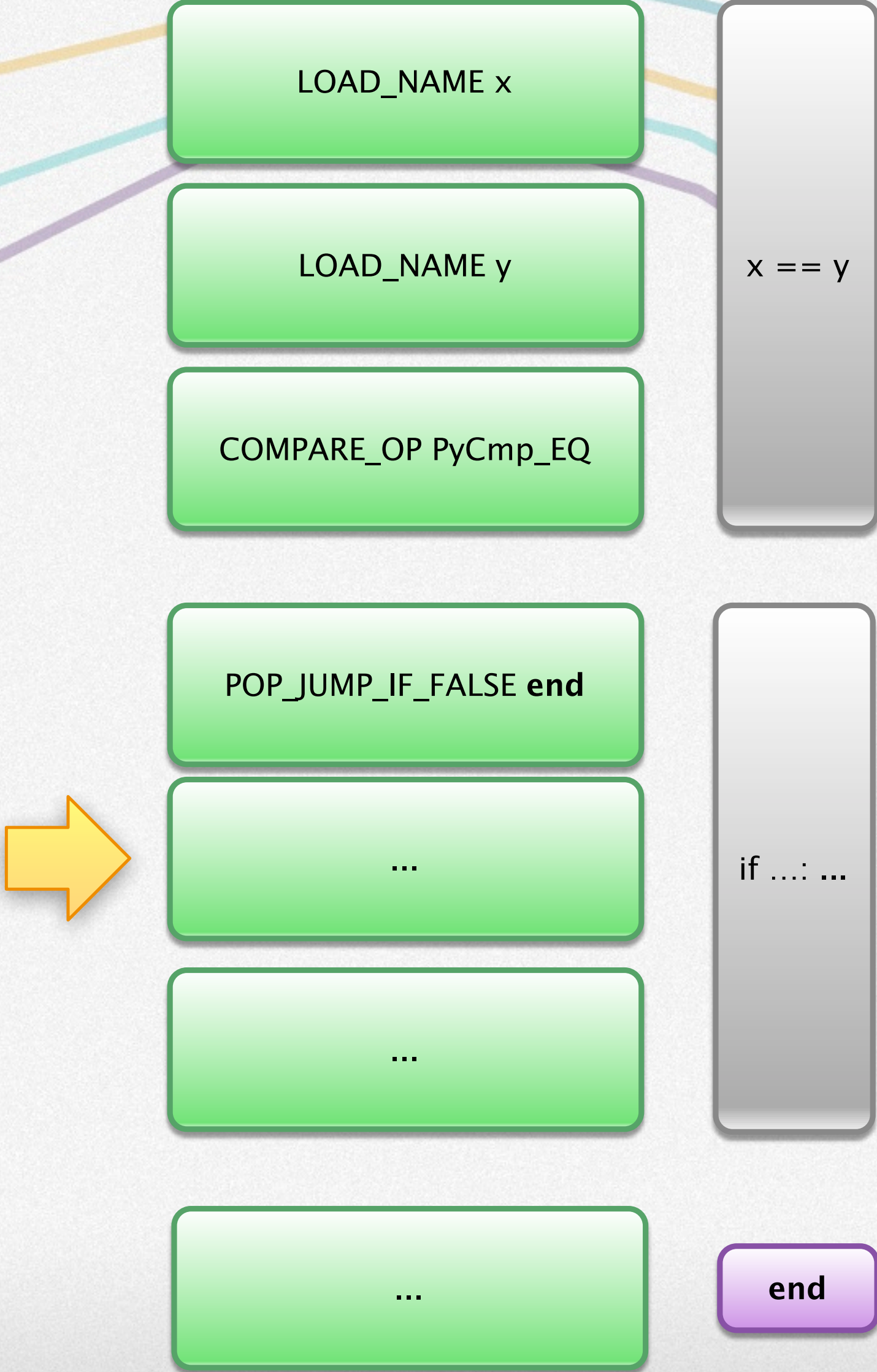


# Bytecode for “if x == y: ...”



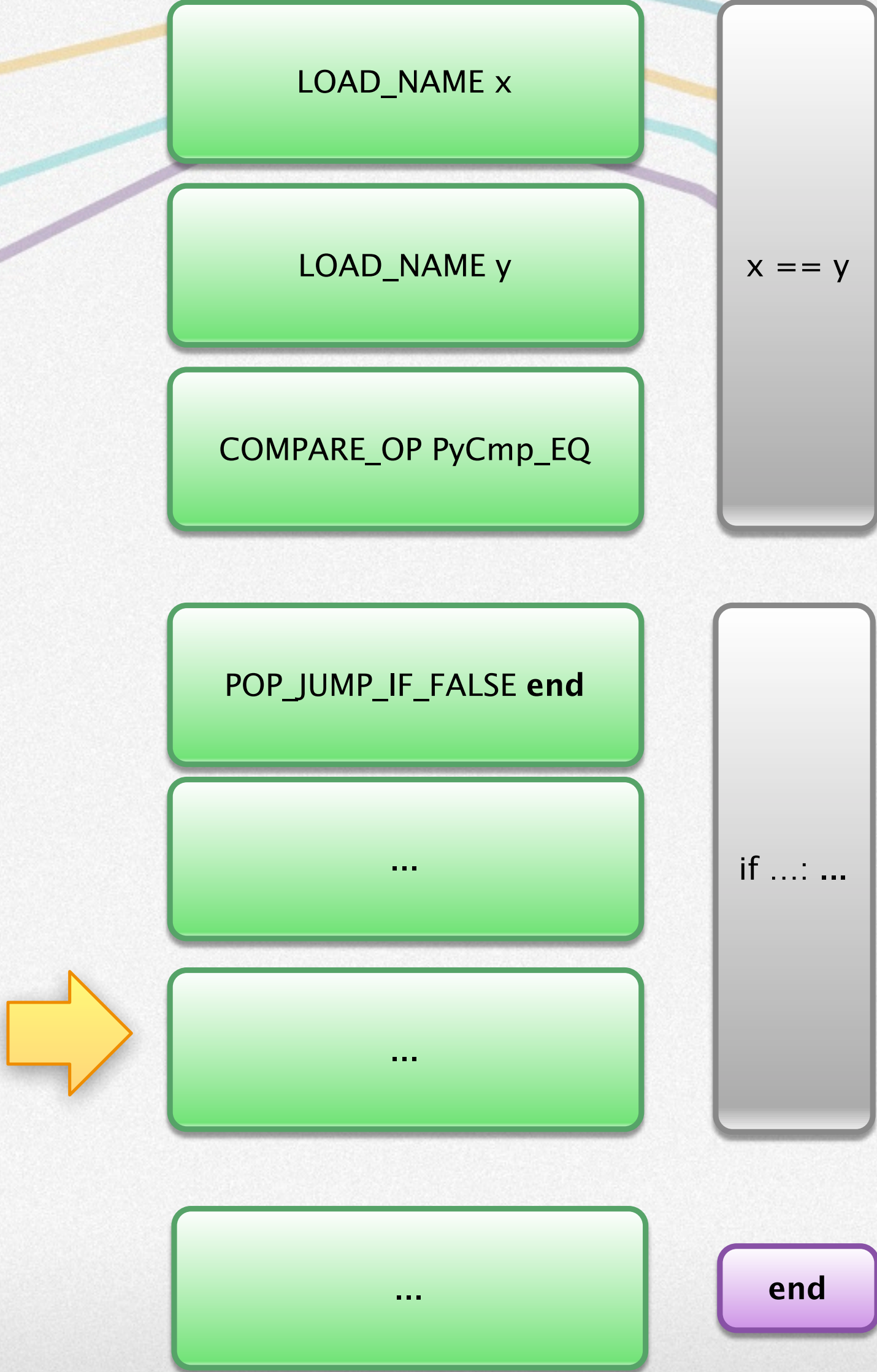


# Bytecode for “if x == y: ...”



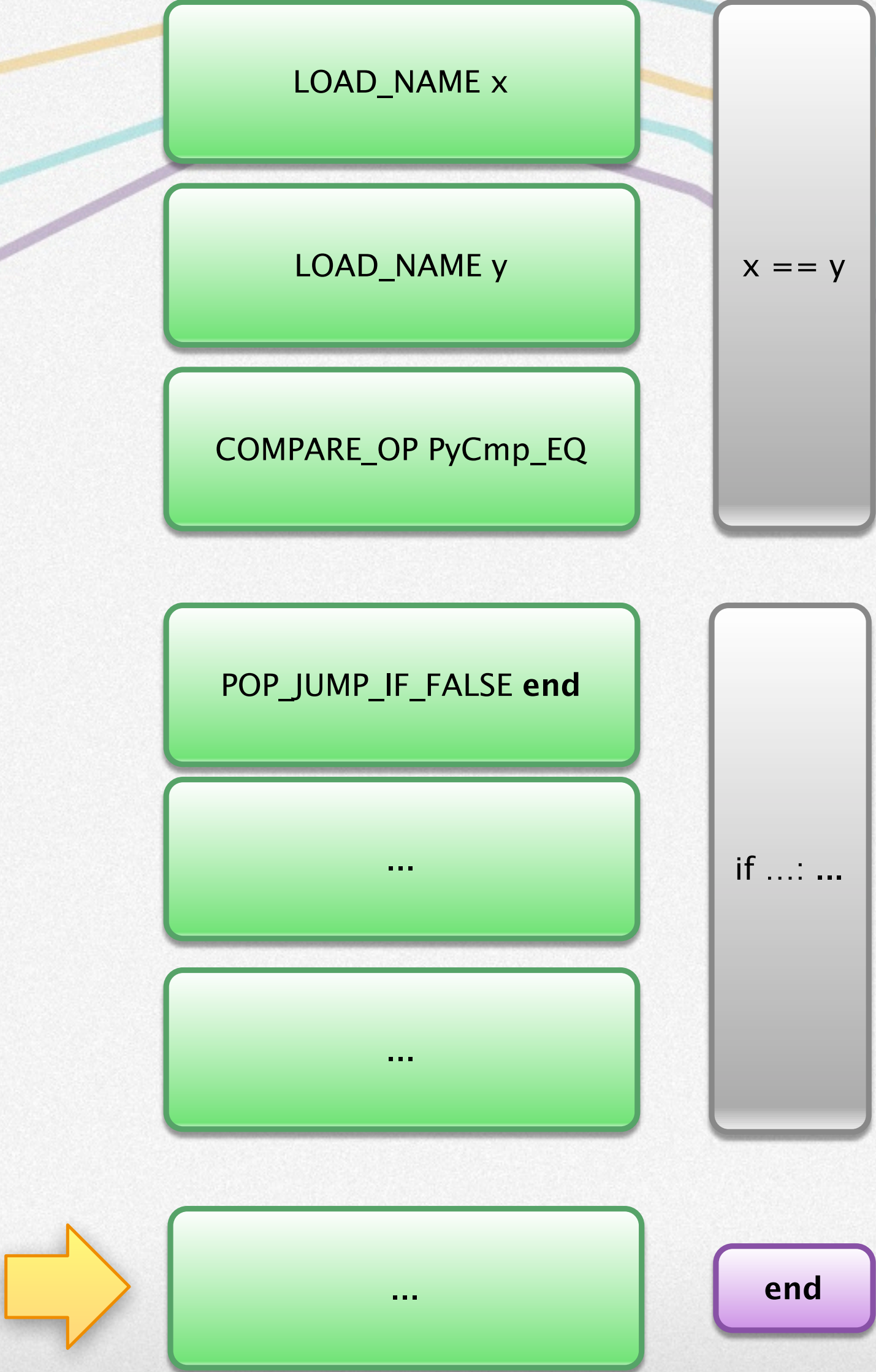


# Bytecode for “if x == y: ...”



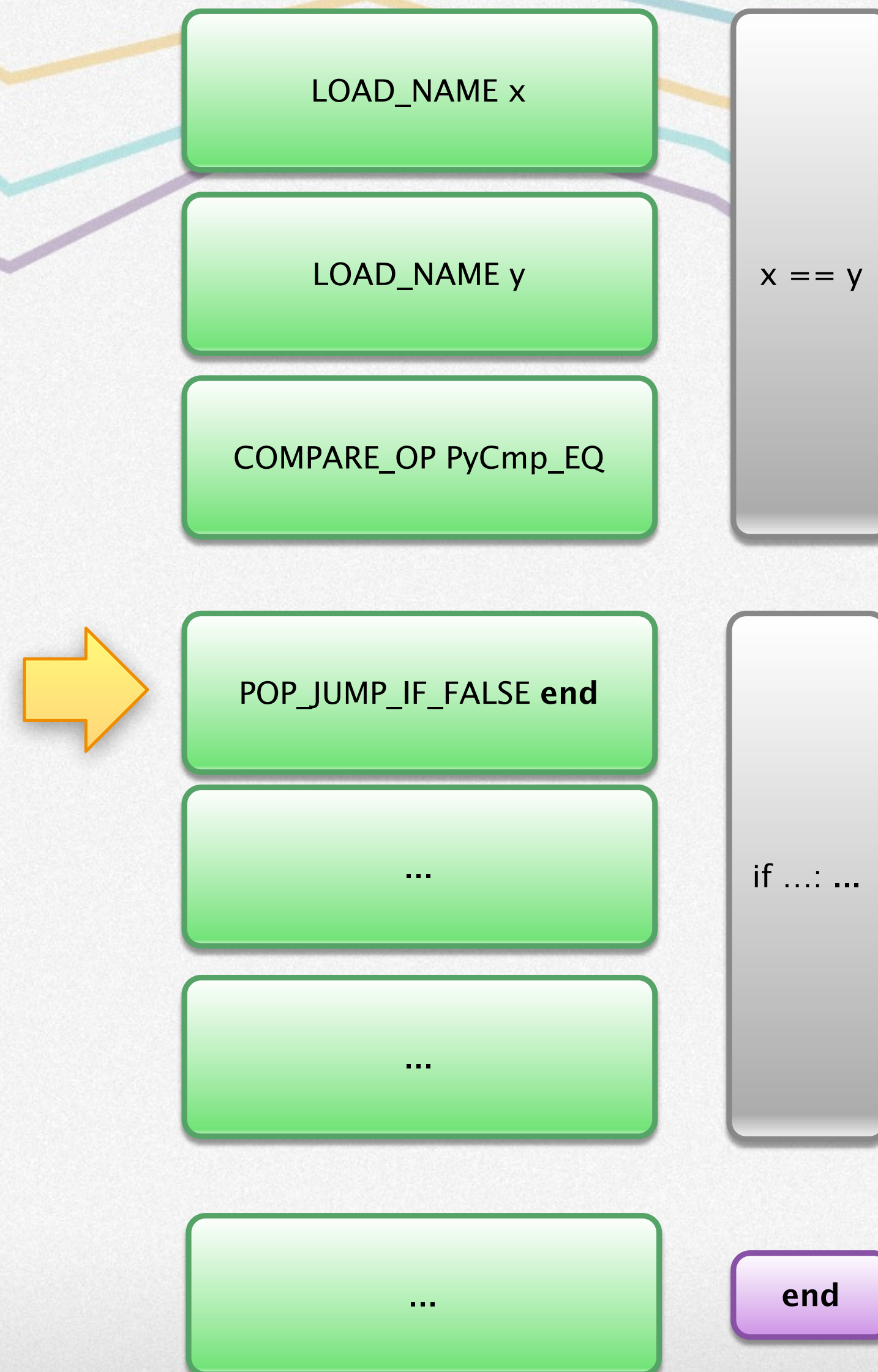


# Bytecode for “if x == y: ...”



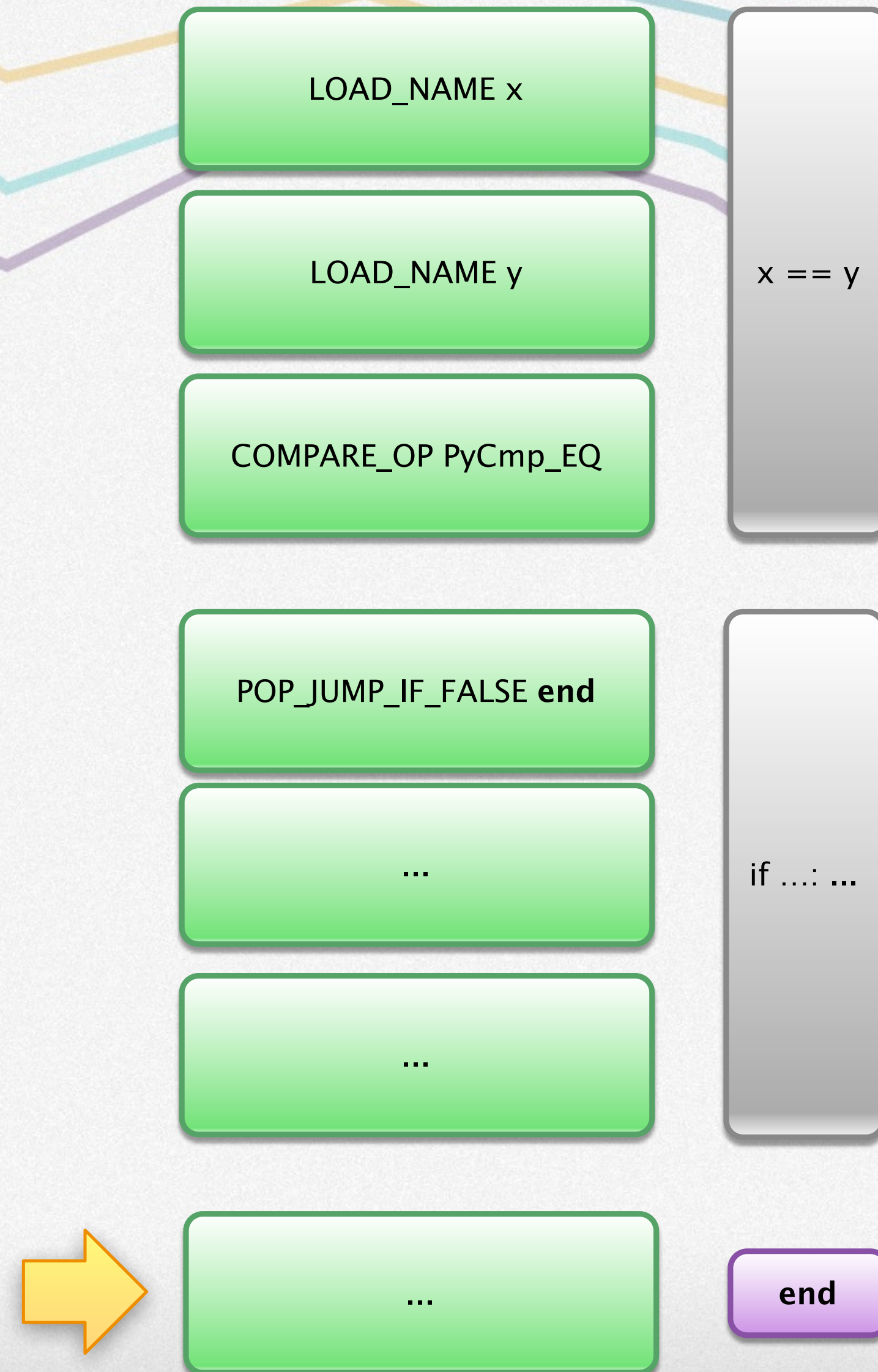


# Bytecode for “if x == y: ...”





# Bytecode for “if x == y: ...”





# 4. Emit bytecode for the new AST node (cont.)



- **Python/compile.c**

- Look for **<AST>\_kind** to track down existing impls for AST nodes -- e.g. **ClassDef\_kind** for classes
- Once again: how is **if** implemented?
  - Look for **If\_kind**
- How can we modify this to implement **unless**?



# Demo!

- The usual **./configure && make** dance on Linux/OSX.
- With a little luck, magic happens and you get a binary in the root directory.
- ... So, uh, does it work?



# And exhale.

- Lots to take in, right?
  - In my experience, this stuff is best learned bit-by-bit through practice.
- Ask questions!
  - Google
  - Python-Dev
  - Or hey, ask me...

🐳 \_ 🐳

**DON'T PANIC**





# Thanks!



[oscon@tomlee.co](mailto:oscon@tomlee.co)

[@tglee](https://twitter.com/tglee)

[http://github.com/thomaslee](https://github.com/thomaslee)

<http://newrelic.com>