Fun with compilers: exploring languages one Python at a time

Peter McCormick
@pdmccormick

github.com/pdmccormick/pyconca2019-fun-with-compilers

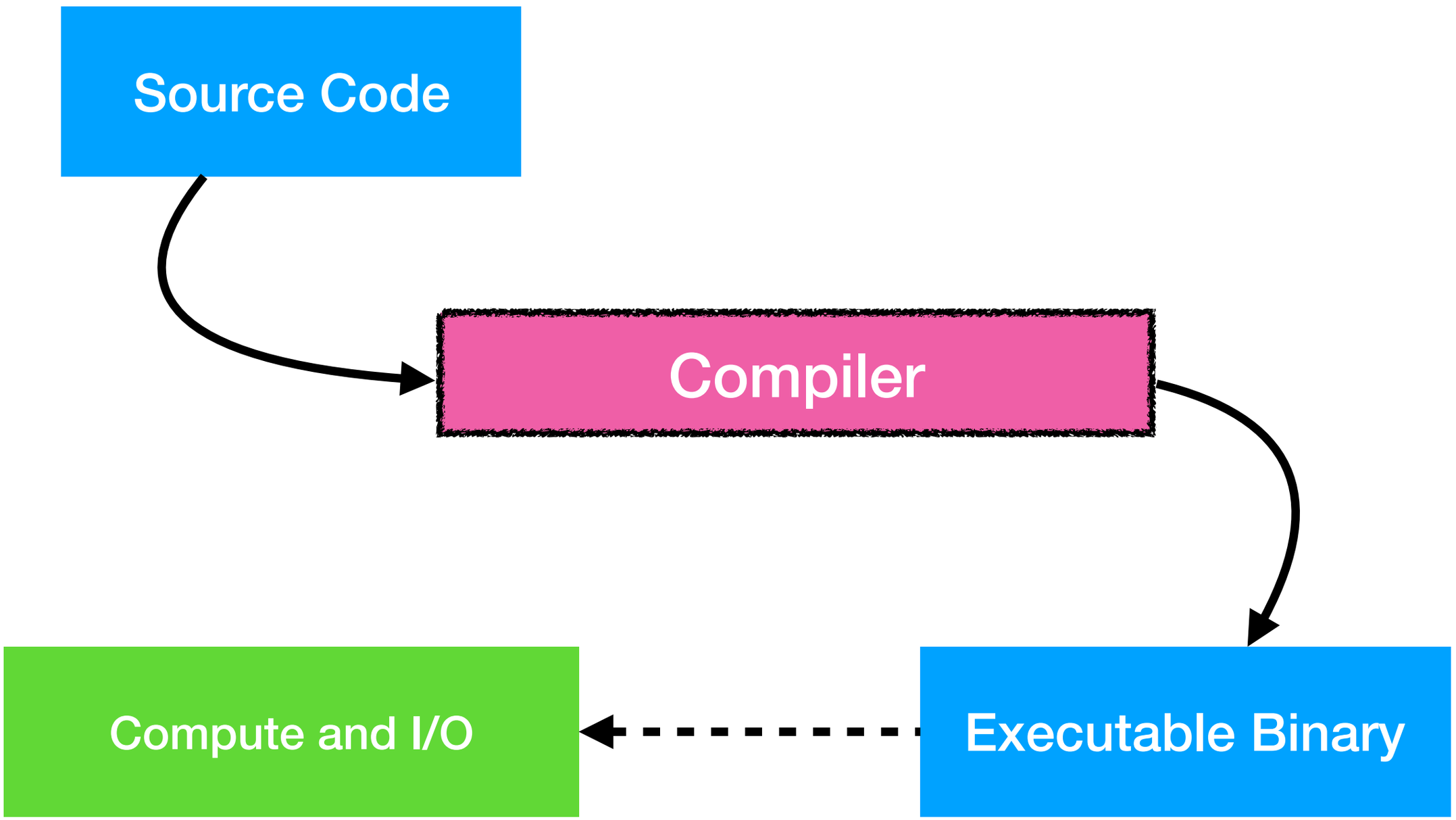
What *is* a compiler?

A **compiler** *transforms a*
human-meaningful *program*
into a
machine-meaningful *form*

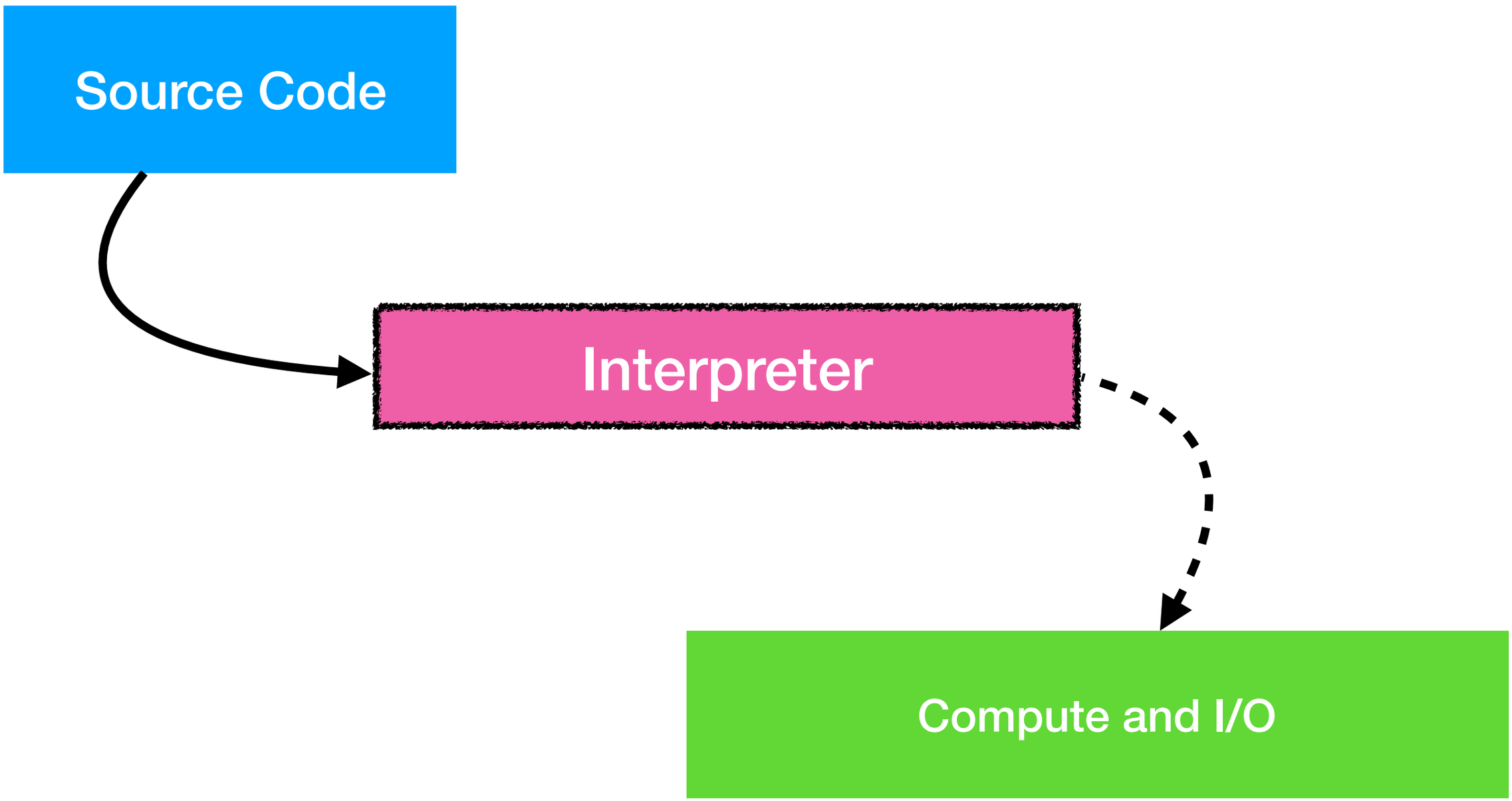
Is Python

compiled or *interpreted*?

Classical compiler



Classical interpreter



Python is neither
interpreted nor compiled

Python is just a language

An *implementation* of a Python engine could be a classical compiler or classical interpreter or a combination of the two

CPython is an implementation of a Python engine

CPython …

- … *compiles* Python into a machine-independent *byte code* representation
- … and *interprets* that byte code using a virtual machine

CPython also…

- Implements a large supporting language runtime
- Includes the *Standard Library*
- Other tools and documentation

Other Python implementations

- Jython: Python on the JVM
- IronPython: Python on .NET
- PyPy: JIT compiler, performance-focused
- Stackless Python: CPython branch support specialized concurrency constructs

Why are compilers *fun*?

1. Data structures & algorithms

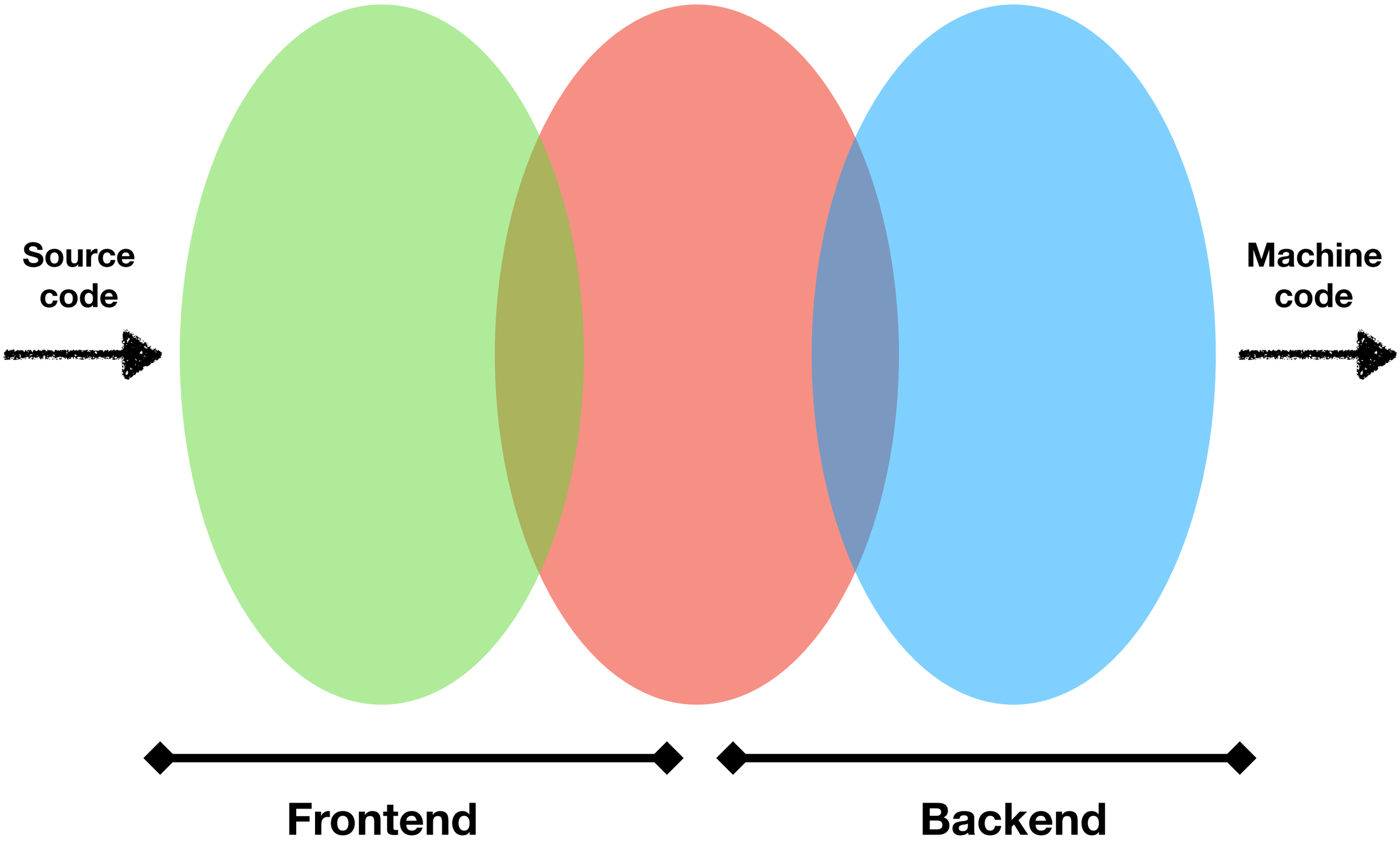
2. Software architecture & design

3. Language design

How are compilers built?

A pipeline of representation transformations

Recognize Analyze Transform



Recognize the language

- Lexical structure:
  - Alphabet and punctuation
- Syntactic structure:
  - Grammar and sentence construction
  - Distinctions like *statements* vs *expressions*
- Language specific!

From *characters*

to *tokens*

```
if x < y: v = 1
```

```
if x < y: v = 1
```

reserved keyword if

identifier x

operator <

identifier y

colon

identifier v

operator =

integer 1

Tokenizer

or

Lexer

or

Scanner

Lexical error are inputs that don't follow the rules of correct token formatting

- Identifiers starting with a number: **1plusX**
- Operators that don't exist: **++** **@@** *******

In Python…

is all whitespace *meaningful?*

```
x = 1 + 2 * 3
```

```
x = 1   +   2   *   3
```

```
x = 1    +      2          *      3
# Hello PyCon Canada!
```

Whitespace on the *left* is meaningful

```
if x == 0:
    zero = True
else:
    zero = False
```

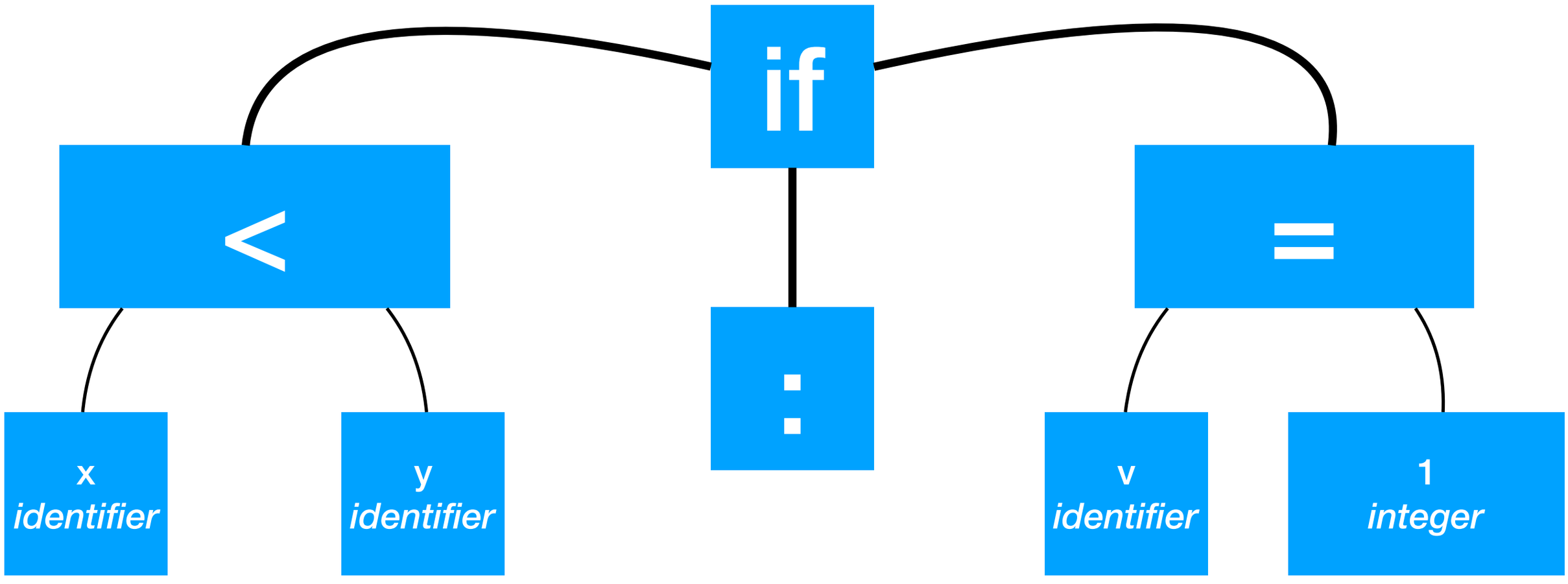
```
if ID == NUM : NEWLINE
INDENT ID = ID NEWLINE
DEDENT else : NEWLINE
INDENT ID EQ ID NEWLINE
DEDENT
```

From *tokens*

to *parse trees*

Parse trees, derivations, concrete syntax trees (CST)

```
if x < y: v = 1
```



Structural shape
according to some language
grammar

Syntax errors are inputs that don't *fit* into the correct shape

A grammar constructions cannot be *recognized*

Different classifications of language grammars

LL(1) ***LR(1)*** ***LALR(1)***

Python's syntax has been thoughtfully designed to be *LL(1)*

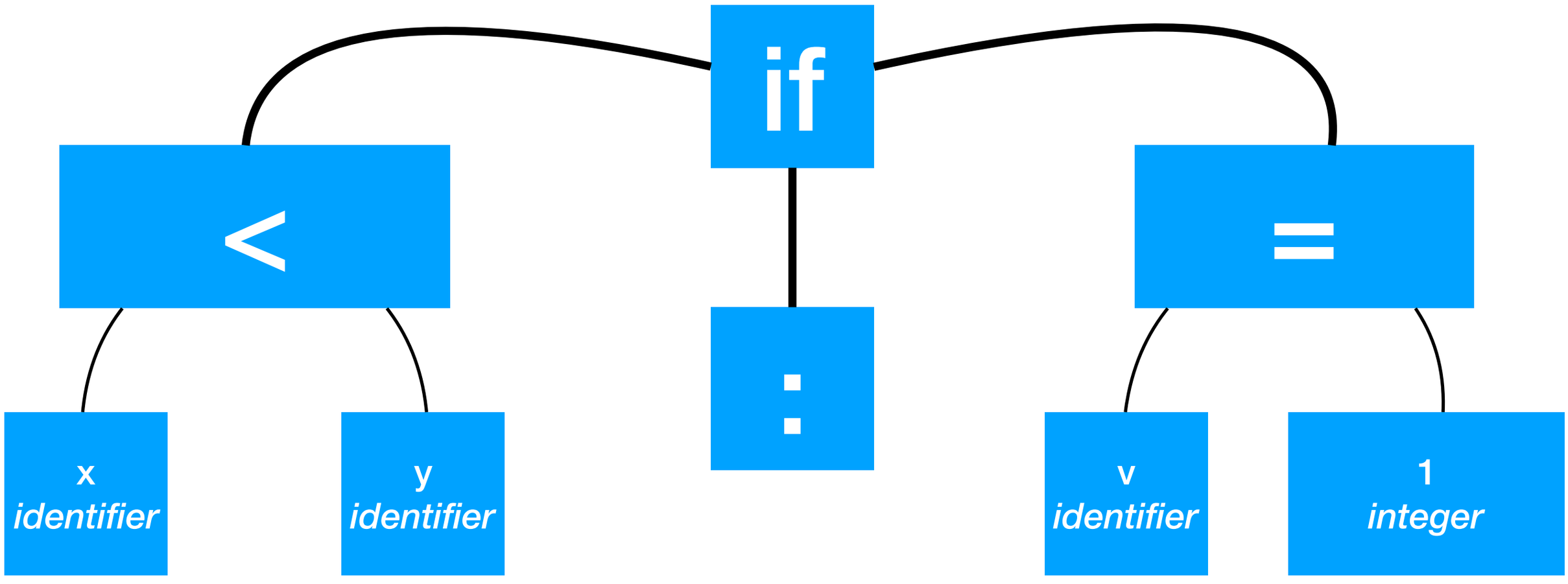
This means it can be easily parsed using a technique called *recursive descent*

More on this later…

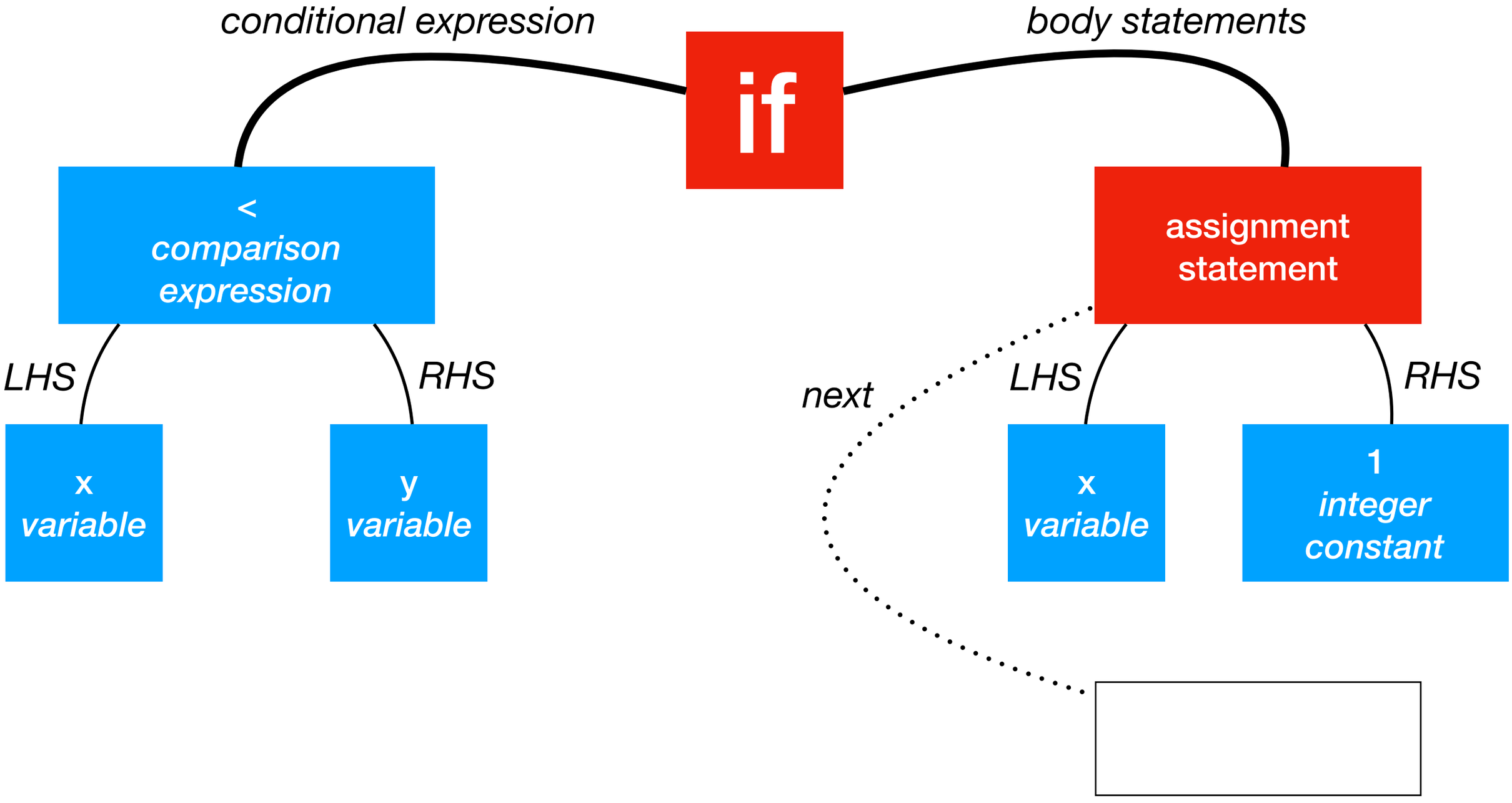
From *parse trees*

to *abstract syntax trees* (AST)

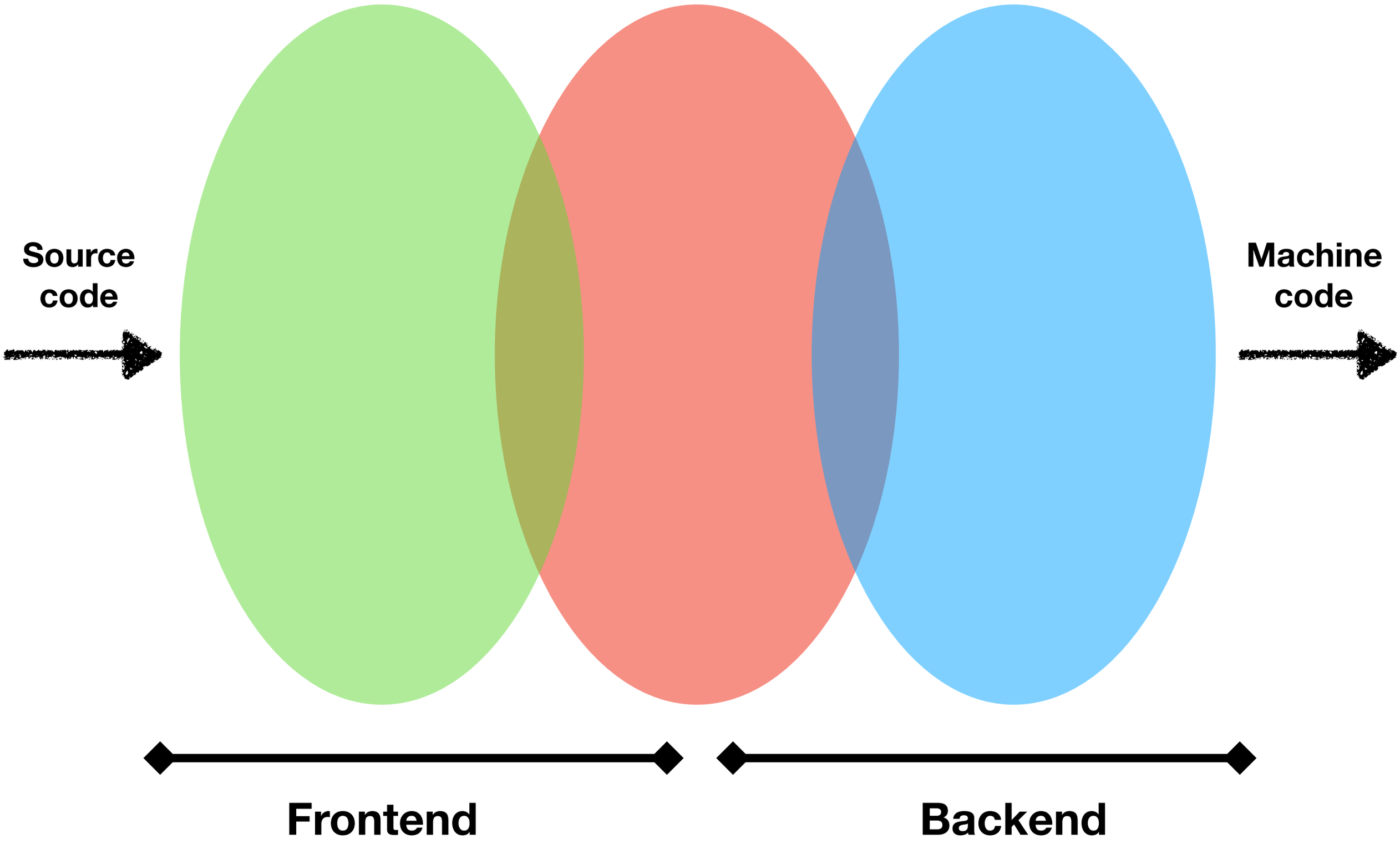
```
if x < y: v = 1
```



```
if x < y { v = 1 }
```


conditional expression
if
body statements
<
comparison
expression
LHS
RHS
x
variable
y
variable
assignment
statement
next
LHS
RHS
x
variable
1
integer
constant

Recognize Analyze Transform



Analyze structure & meaning

- Analyze AST for invalid language constructions
- Add extra annotations to the AST structure
  - Link names together through *symbol tables*
- Checking for correct usage
- Possibly transform between different flavours of AST
  - Untyped vs typed

Analysis: invalid constructions

- `break` or `continue` without an enclosing loop

```
if True:
    break
```

Where's the loop?!

Analysis: what’s in a name?

```
def add(x, y):
    return x + y
```

Analysis: what's in a name? (2)

```
int year = 2019;
print(year);

{
    int year = 2019;
    print(year);
}
```

Same *name*
but referring to two
different things!

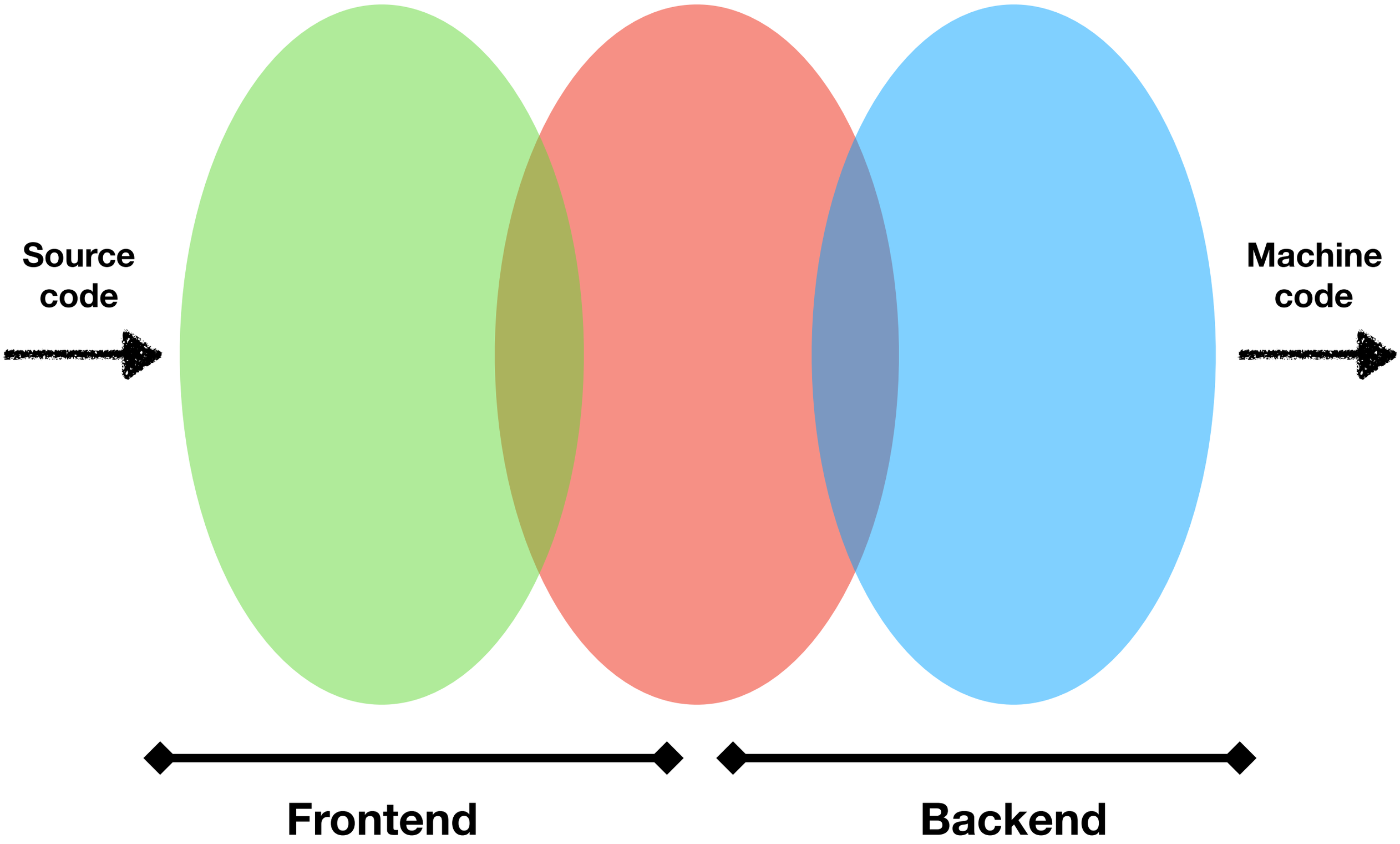
Python defers some things to *run-time* that other languages can detect at *compile-time*

```
if random() < 0.5:
    year = 2019

print(year)
```

NameError: name 'year' is not defined

Recognize Analyze Transform



Transform into final form

- Generate machine code!
- Take machine target specifics into account
- Make choices about memory, code organization, instruction selection
- Perform *optimizations* in space and time dimensions (can be done at AST level or later or both)

From AST’s to other tree and graph structures

Intermediate Representations (IR)

IR's tend to resemble machine code assembly languages

```
if x < y { v = 1 }
```

```
  load r1, x
  load r2, y
  lt r1, r2, r3
  jneq _end
  const r1, 1
  loadaddr r2, v
  store r2, r1
_end:
```

CPython represents compiled Python in the form of *bytecode* based instruction set

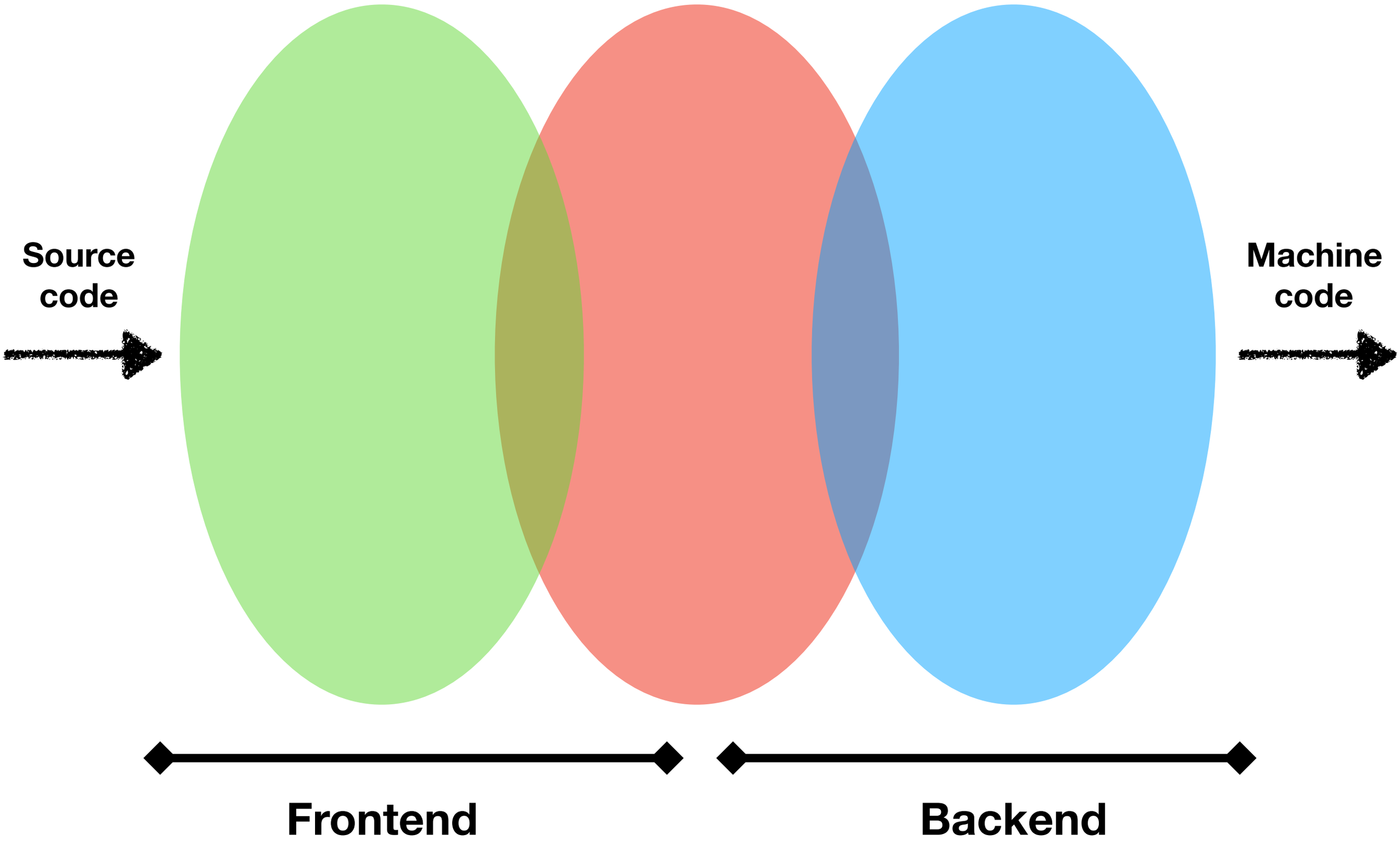
Bytecode is designed to be easy to *interpret* by a *virtual machine*

(i.e. an emulated processor, not a physical one)

.pyc files in *__pycache__* directories

"Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler"

Recognize Analyze Transform



Fun *with* compilers

CPython let's you programmatically compile Python!

- `compile`, `exec` and `eval` functions
- Many modules in the *Python Language Services*:
  - `tokenize` `parser` `ast` `dis` `symtable` `symbol`
  - `importlib`
  - More on PyPI: `astor` `bytecode`

What about playing *with* languages?

Let’s write a compiler!

Design a new language, or implement for an existing language?

Lots of moving parts

Lots of fun,

but also a lot of work

Let’s modify a compiler!

CPython is written in C

Excellent code base, highly recommended for study

Lots of thoughtful technical and project governance around CPython

But can I play with something written in Python?

Can I play with Python using something written in Python?

PyPy is a Python engine written in Python

Extremely
sophisticated

Transpilers:
source to source translators

JavaScript *minifiers*

Source to source minifiers

Remove whitespace

Shorten identifiers

Remove unused code

Babel:

use the future today!

Source++ to source

Express new language constructs
in terms compatible with older versions

Source++ to AST++

AST++ to AST

AST to Source

We need a *partial frontend* of a Python compiler…

Ideally written in Python

We want to be able to…

- Modify a Python parser to *recognize* our Python++ syntax, generating a modified AST++ structure
- Add custom analysis passes in terms of AST++
- Convert AST++ back into CPython AST
- Convert CPython AST back into Python source
- Feed the source back into the CPython compiler

PyPython: a pure Python implementation of a Python language parser and AST

A port of the CPython parsing grammar and AST design

Easy* to modify and play with

** Difficulty may vary…*

Add new Python++ syntax recognition

Generate AST++ from modified parser

Transform from PyPython AST++ into CPython AST

Run using CPython!

What could we *do* with this?

End run around to adding new syntax to Python!

Playfully explore new additions to the Python language

Example: *for-while* loops

Live coding!

github.com/pdmccormick/pyconca2019-fun-with-compilers

Shameless plug!

CompilerCourse.com

I'm putting together a free online course all about compilers! To be released in 2020.

Visit *CompilerCourse.com* and sign-up to my announcement mailing list if you'd like to find out more!

Conclusion

Thank you to all those who have made Python what it is today!

Let's have fun *with these wonderful tools that we've been gifted with!*

May you go and playfully explore and perhaps discover the next language feature we will all be thankful to use

Thank you!

@pdmccormick

github.com/pdmccormick/pyconca2019-fun-with-compilers

CompilerCourse.com