# Logical Graphs

## Control Flow Operations in TensorFlow

# Hello!

## I AM SAM ABRAHAMS

Co-author of *TensorFlow for Machine Intelligence*

Teach “Deep Learning with TensorFlow” at Metis

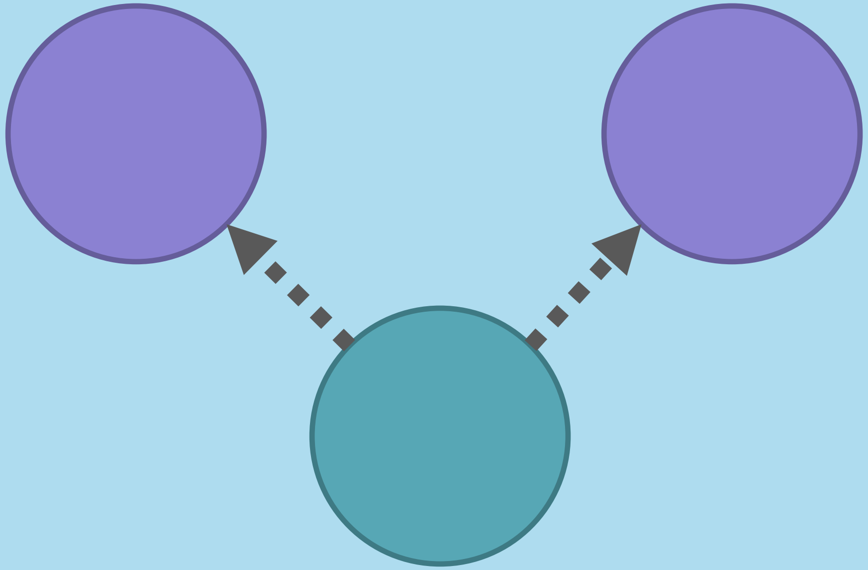
Long time TensorFlow contributor

The “TensorFlow on Raspberry Pi” guy (who isn’t Pete Warden)

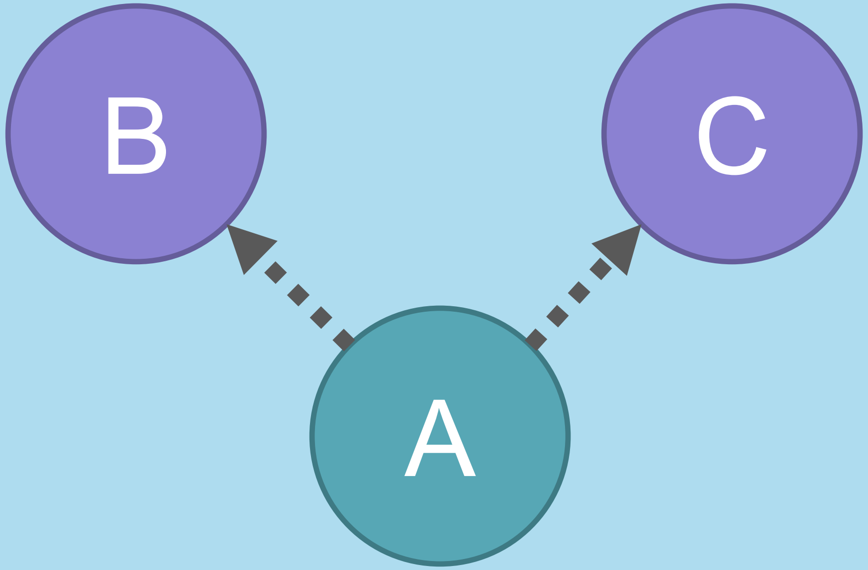
# Slides and code:

github.com/samjabrahams/talks/tree/master/tensorflow/control_flow

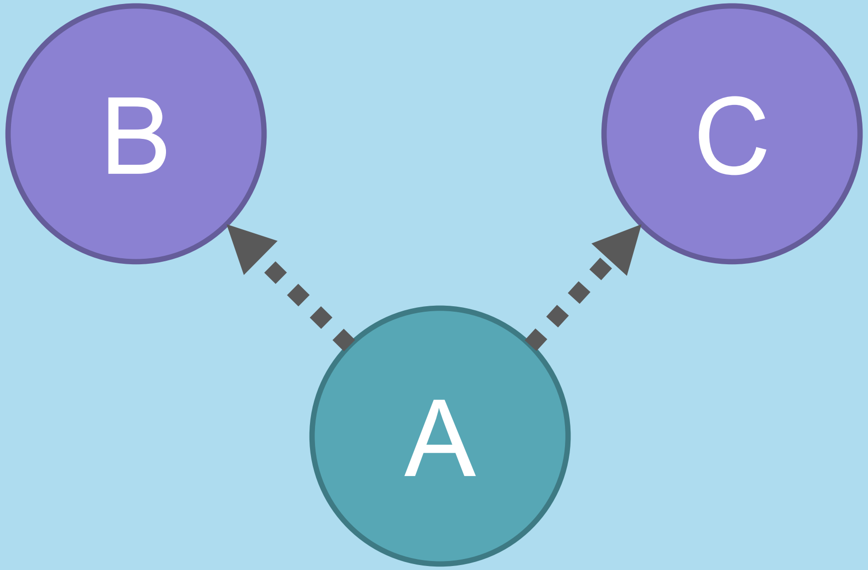
# Control flow: an example



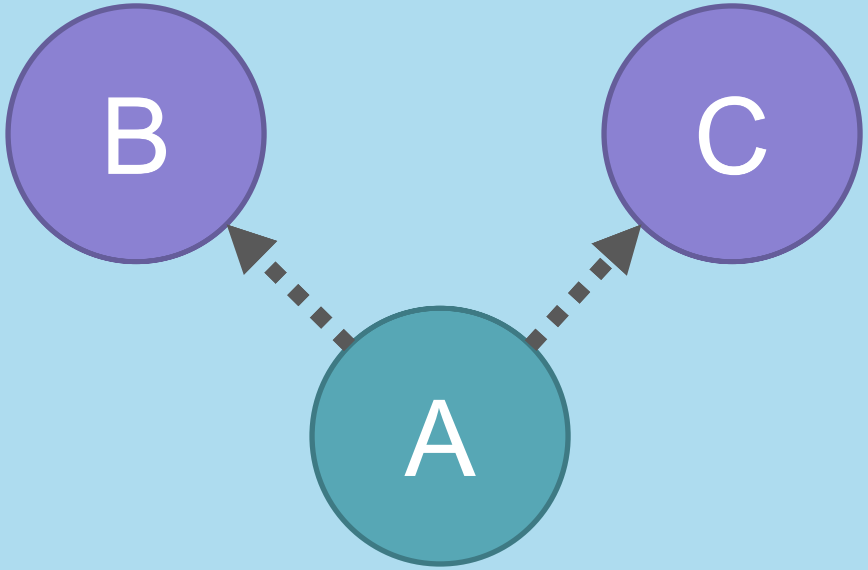
# We want to run either B or C, based on A’s value



# How might we do this?

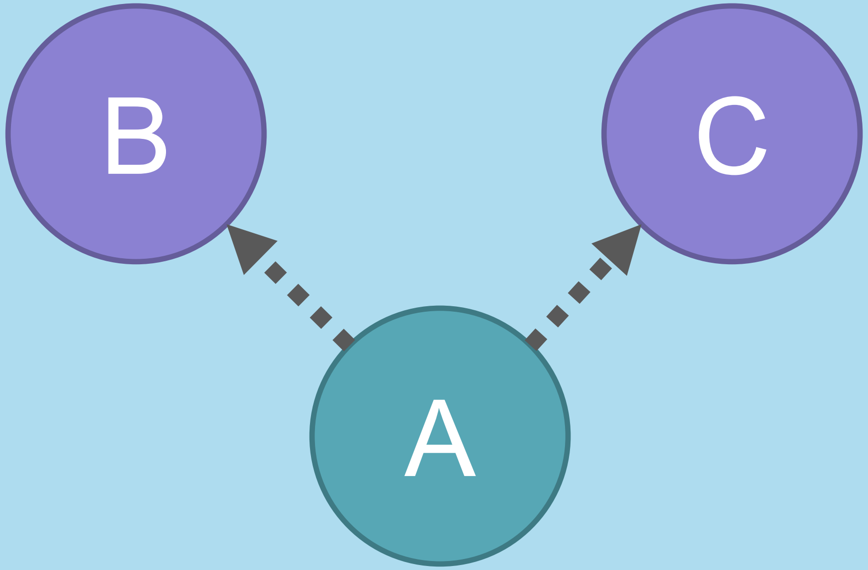


# Naively: use Python `if/else` and multiple runs,



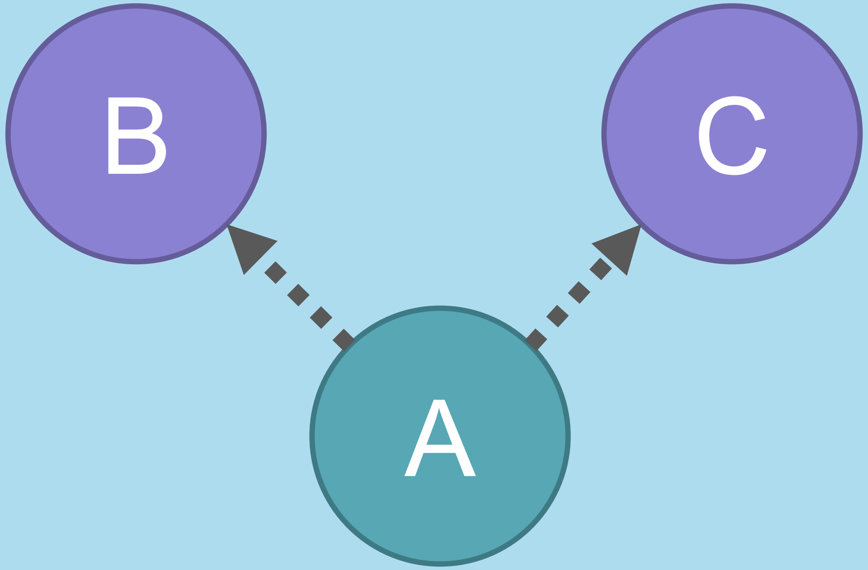

```
a = sess.run(a_op)
feed = {a_op: a}
if a > 0:
    sess.run(b_op, feed)
else:
    sess.run(c_op, feed)
```

# But this is awkward



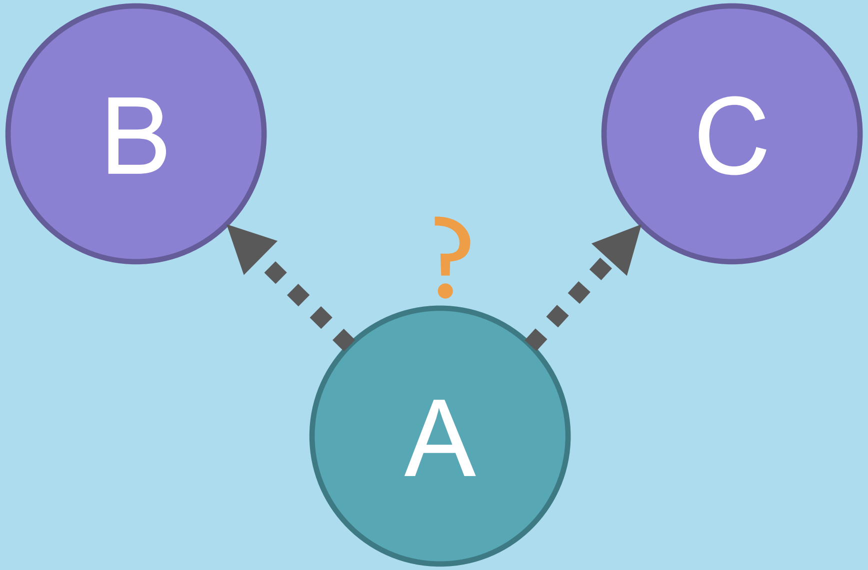

```
a = sess.run(a_op)
feed = {a_op: a}
if a > 0:
    sess.run(b_op, feed)
else:
    sess.run(c_op, feed)
```

# We fetch a value only to feed it back in



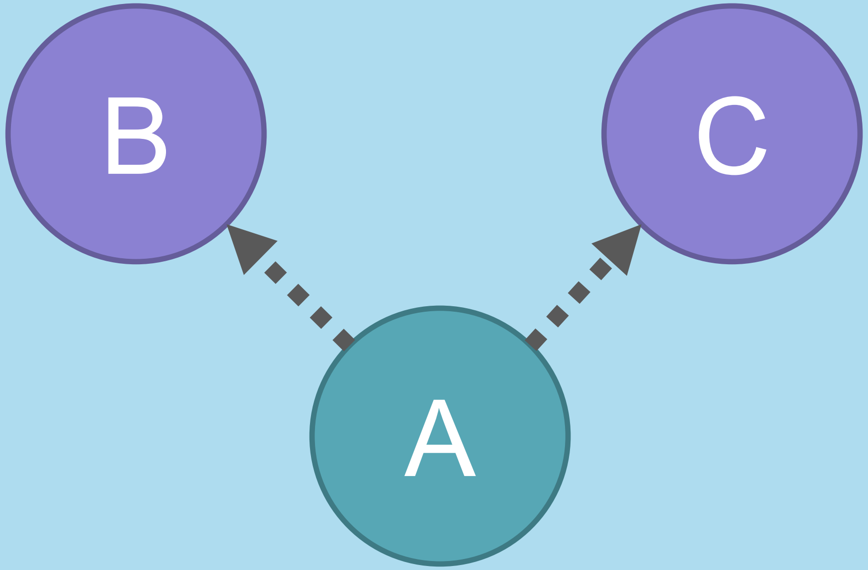

```
a = sess.run(a_op)
feed = {a_op: a}
if a > 0:
    sess.run(b_op, feed)
else:
    sess.run(c_op, feed)
```

# Python logic isn’t represented in the graph



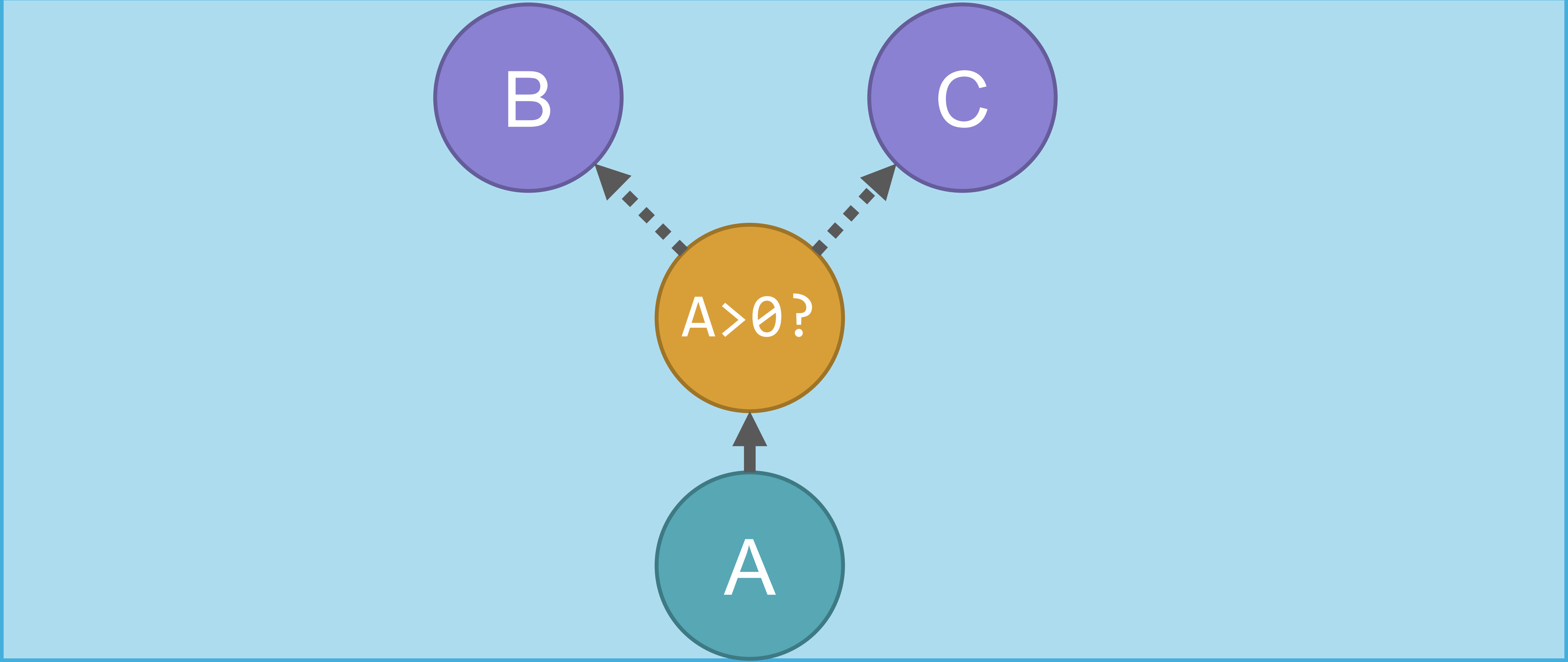

```
a = sess.run(a_op)
feed = {a_op: a}
if a > 0:
    sess.run(b_op, feed)
else:
    sess.run(c_op, feed)
```

# Also: one `sess.run` should represent an entire run




```
a = sess.run(a_op)
feed = {a_op: a}
if a > 0:
    sess.run(b_op, feed)
else:
    sess.run(c_op, feed)
```

# What we want: native logic gate



# Obvious follow up

TensorFlow has several operations for

**native control flow**

# Types of control available in TensorFlow

- Dependencies
  - `tf.control_dependencies`, `tf.group`, `tf.tuple`
- Conditional statements
  - `tf.cond`, `tf.case`
- Loops
  - `tf.while_loop`

# Why care about native control flow?

1. Efficiency
2. Flexibility
3. Compatibility

## Efficiency

- Passing data to/from the Python layer is slow
- Want to run graph end-to-end as much as we can
- Takes advantage of pipelining, such as queues

## Flexibility

- Empower static graphs with dynamic components
- Model logic kept in one place → better decoupling
- Graph can change without affecting training loop

## Compatibility

- Debug and inspect with TensorBoard
- Seamlessly deploy with TensorFlow Serving
- Auto-differentiation, queues, pipelining

# Note:

I’m bad with colors

Color change in

3

# Color change in

2

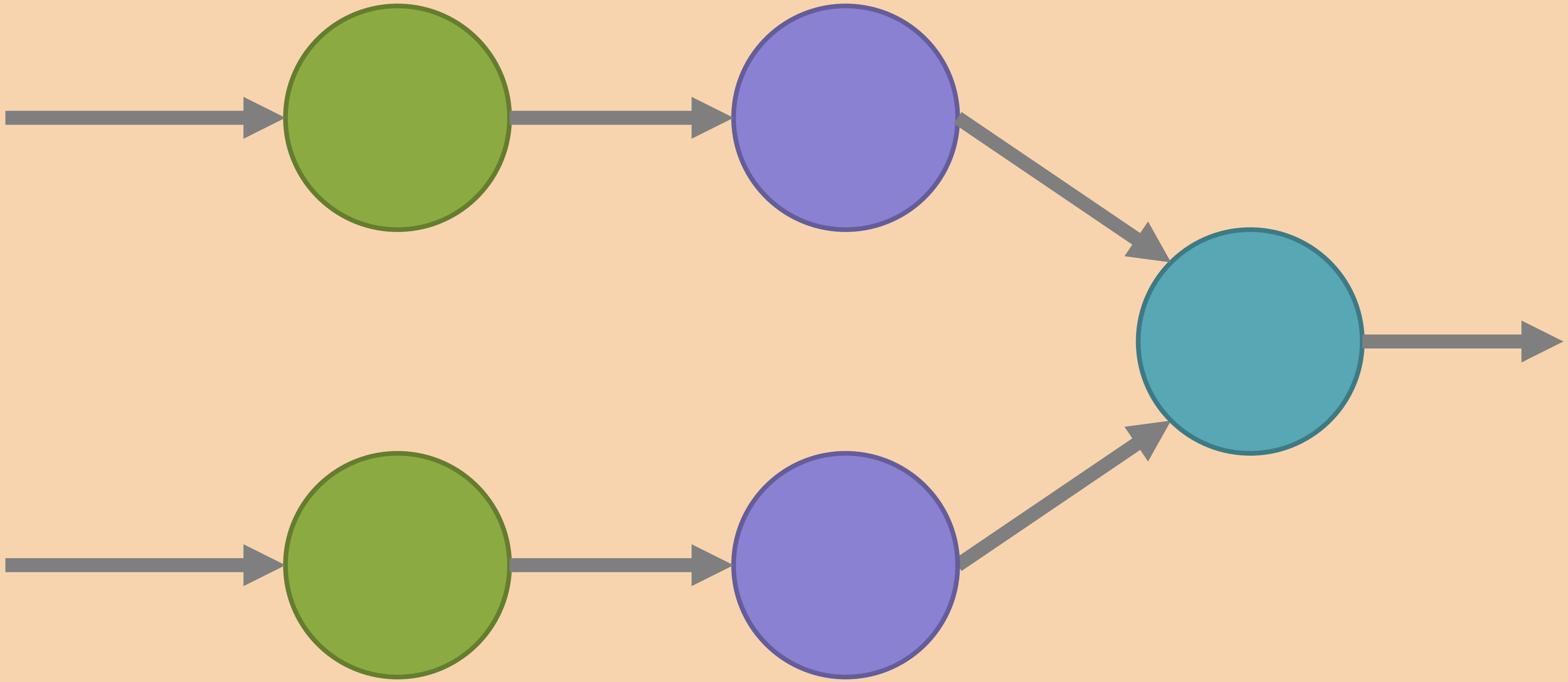
# Color change in

1

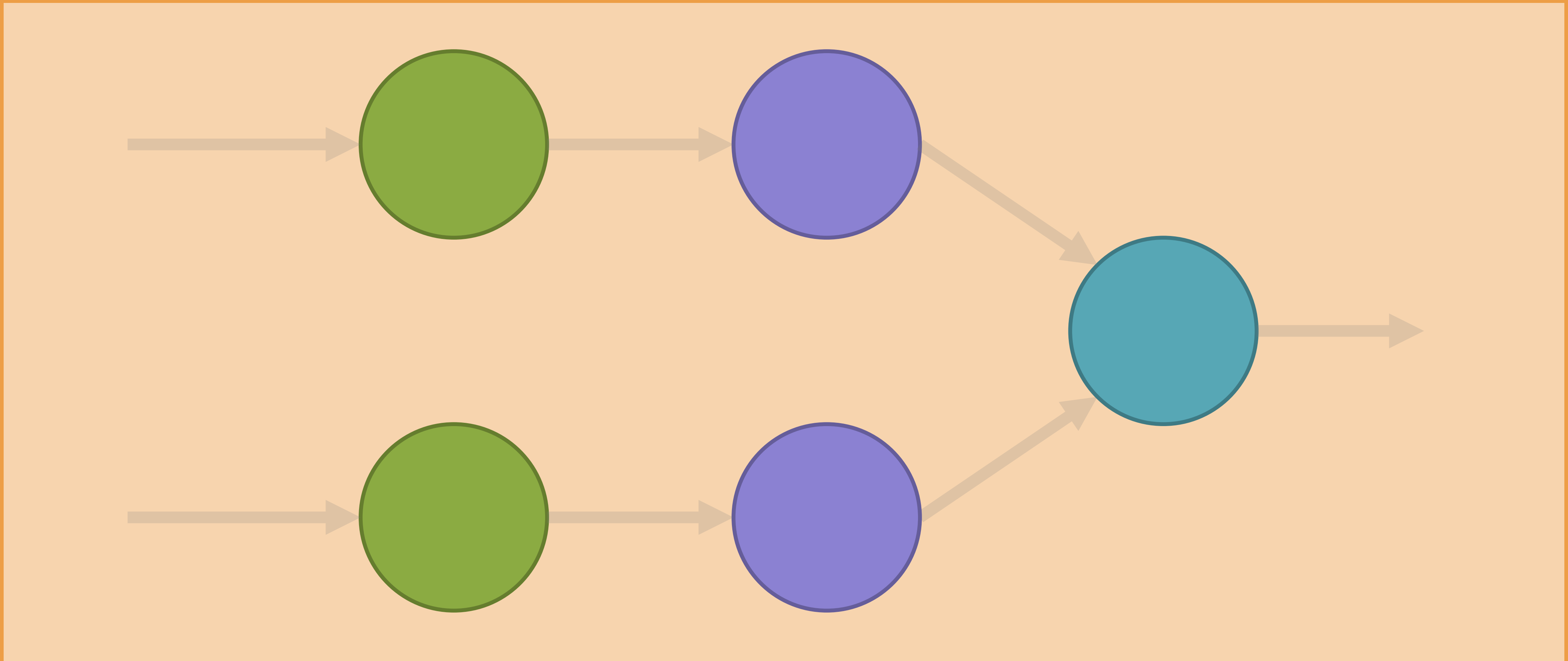
# CONTROL DEPENDENCIES

# Dependencies: quick recap

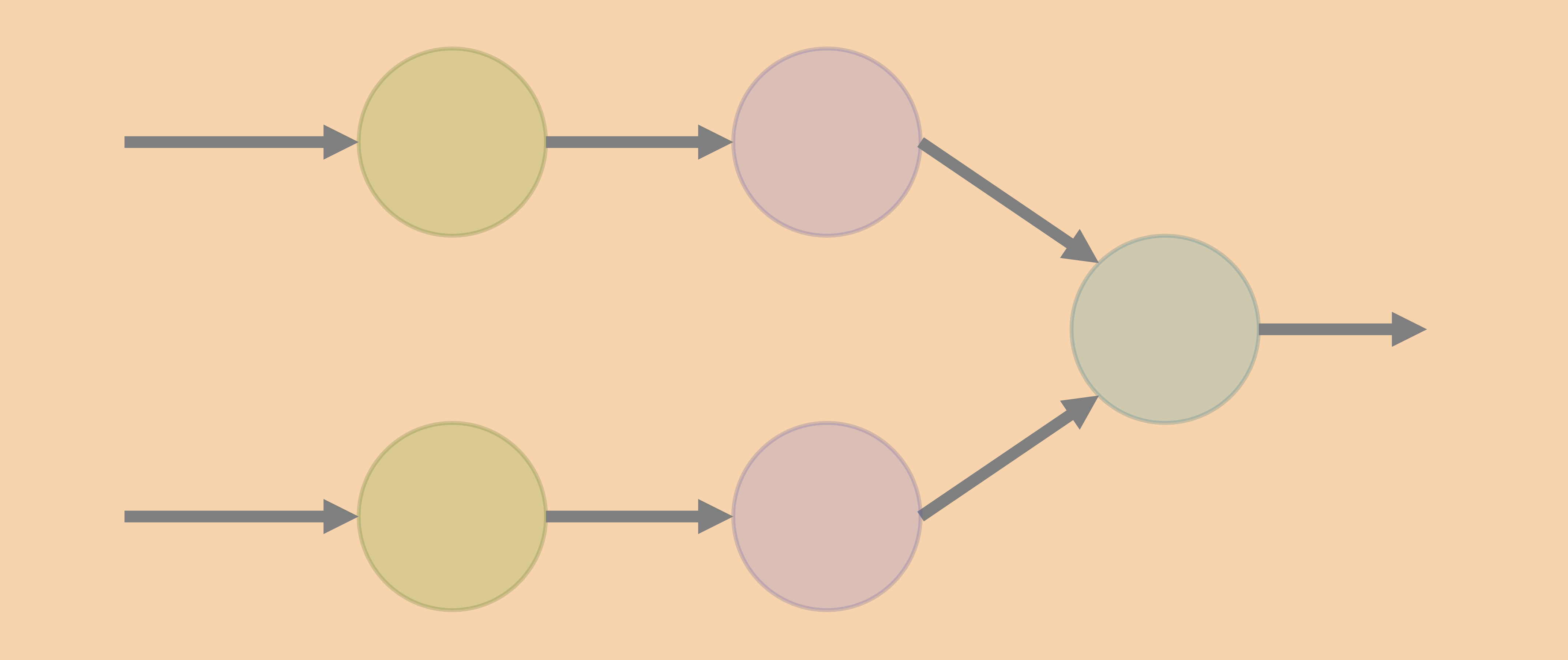
# Here’s a graph!



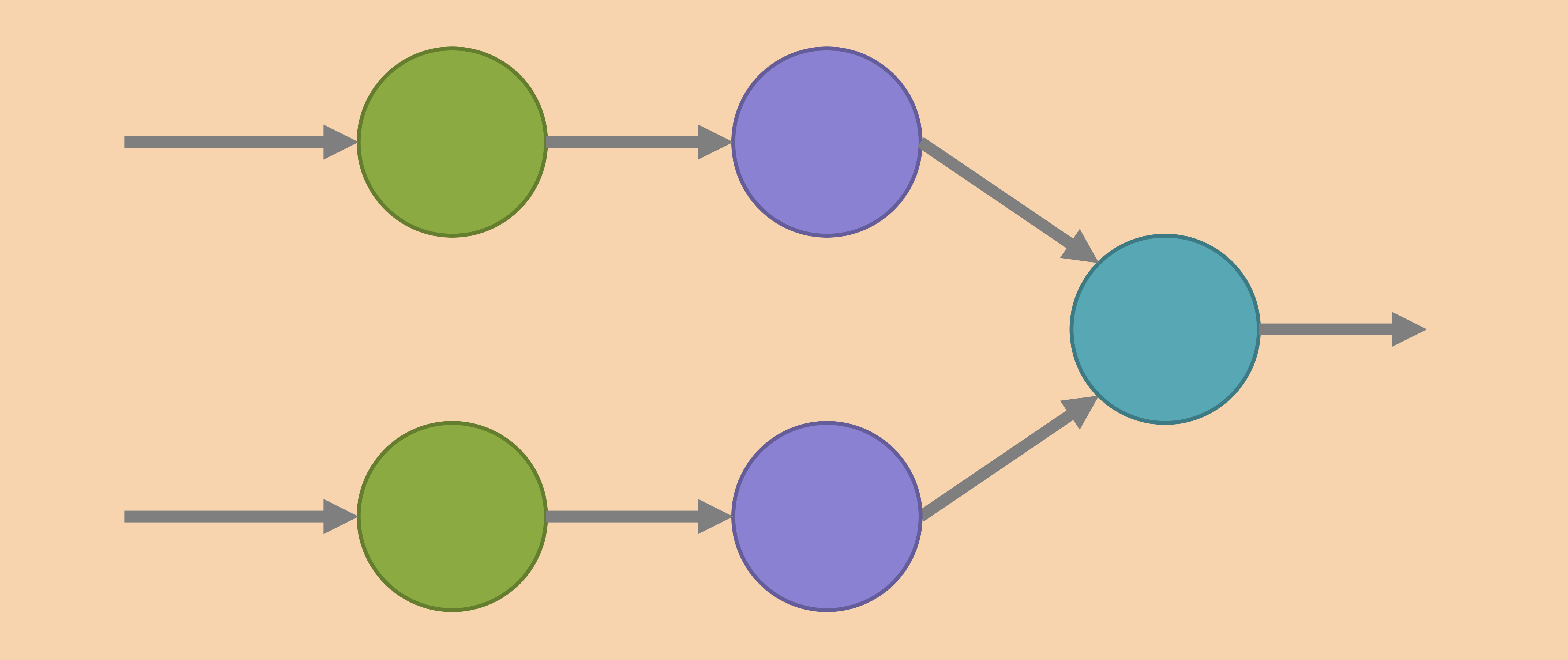
# Nodes (operations)



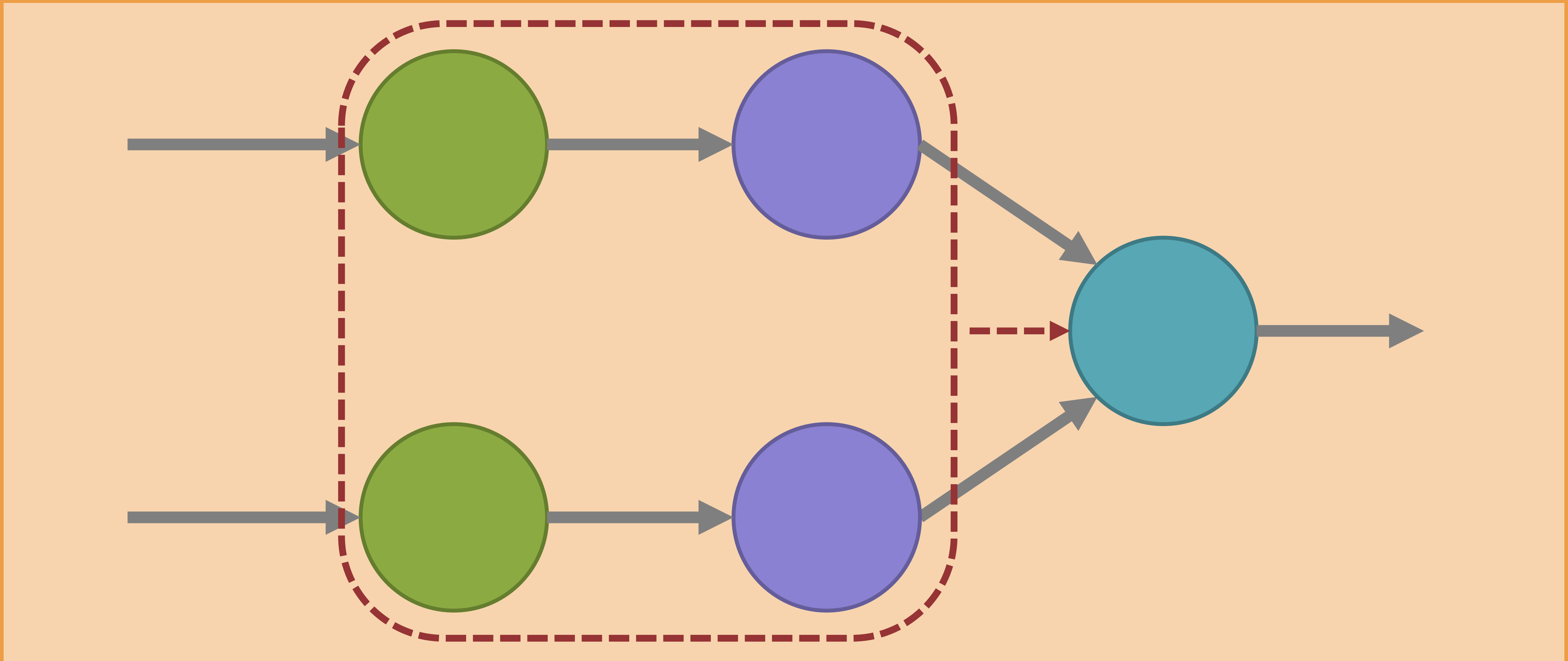
# Edges: (tensors)



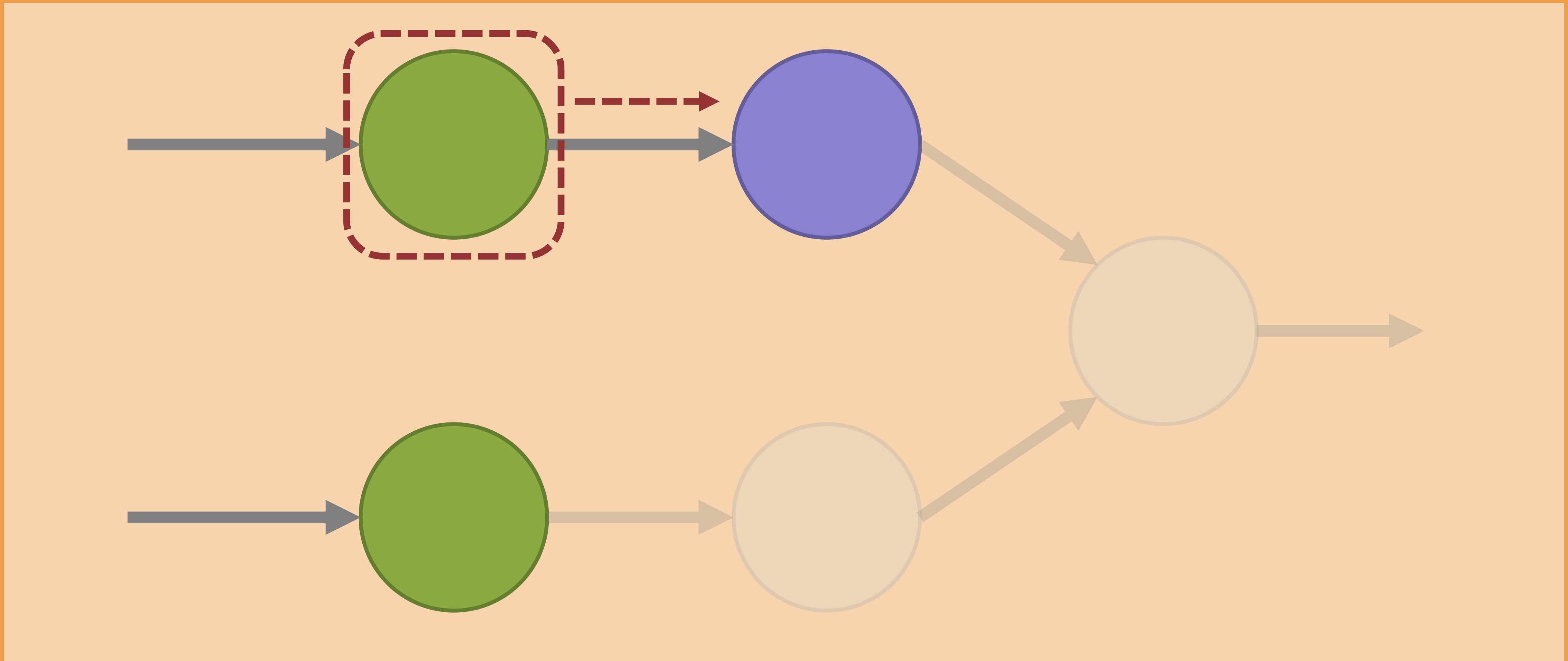
# Dependencies: nodes required to compute another node



# The dependencies of the last node



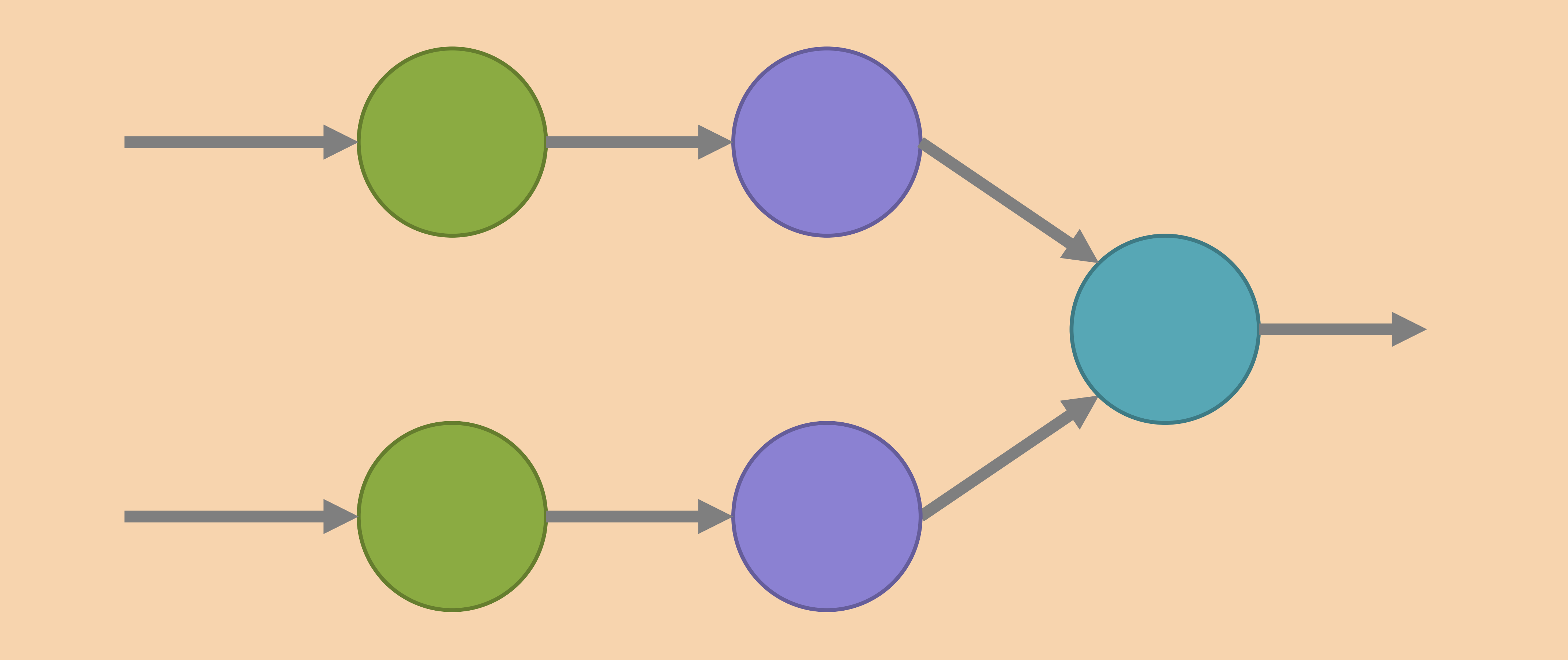
# Dependency of an earlier node



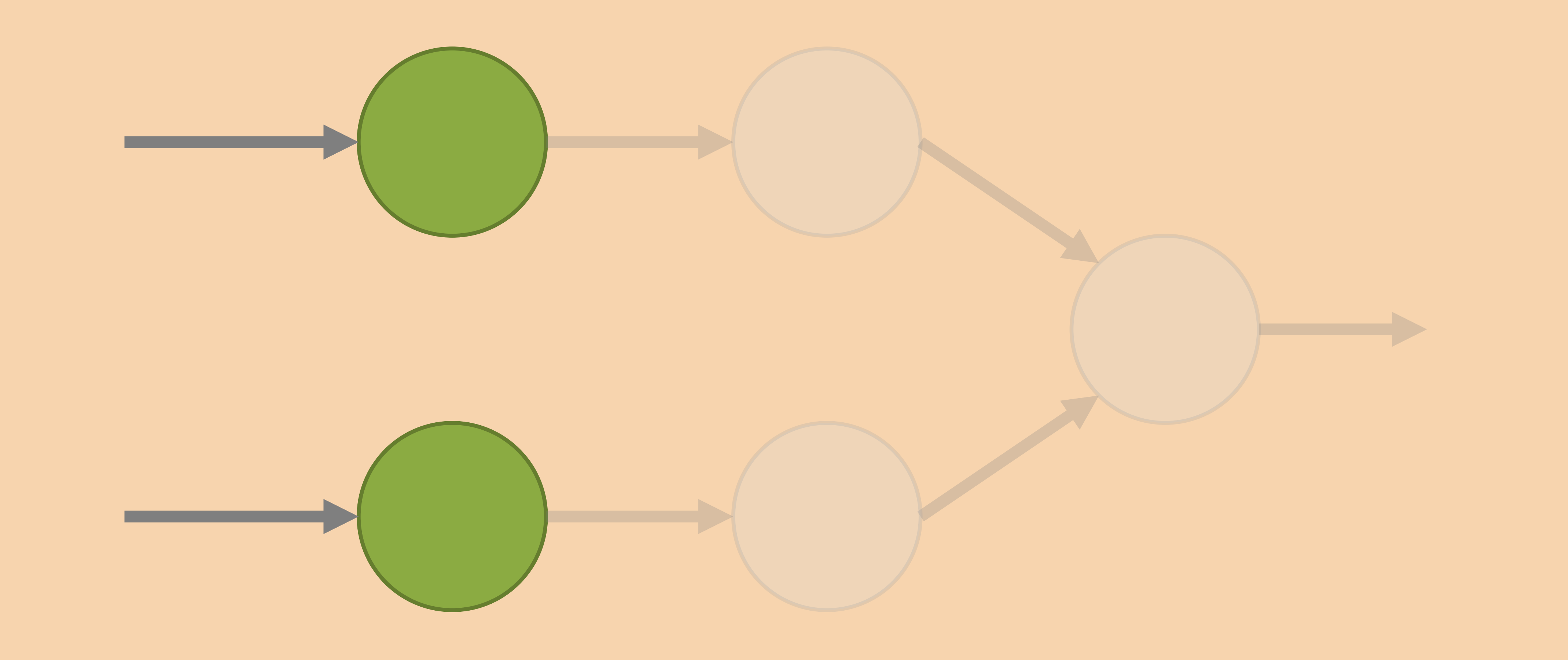
# Dependencies and execution order

- TF keeps track of every operation's dependencies
- Uses them to schedule computation
  - An op is eligible to run once its dependencies have finished
- Two eligible ops can execute in any order

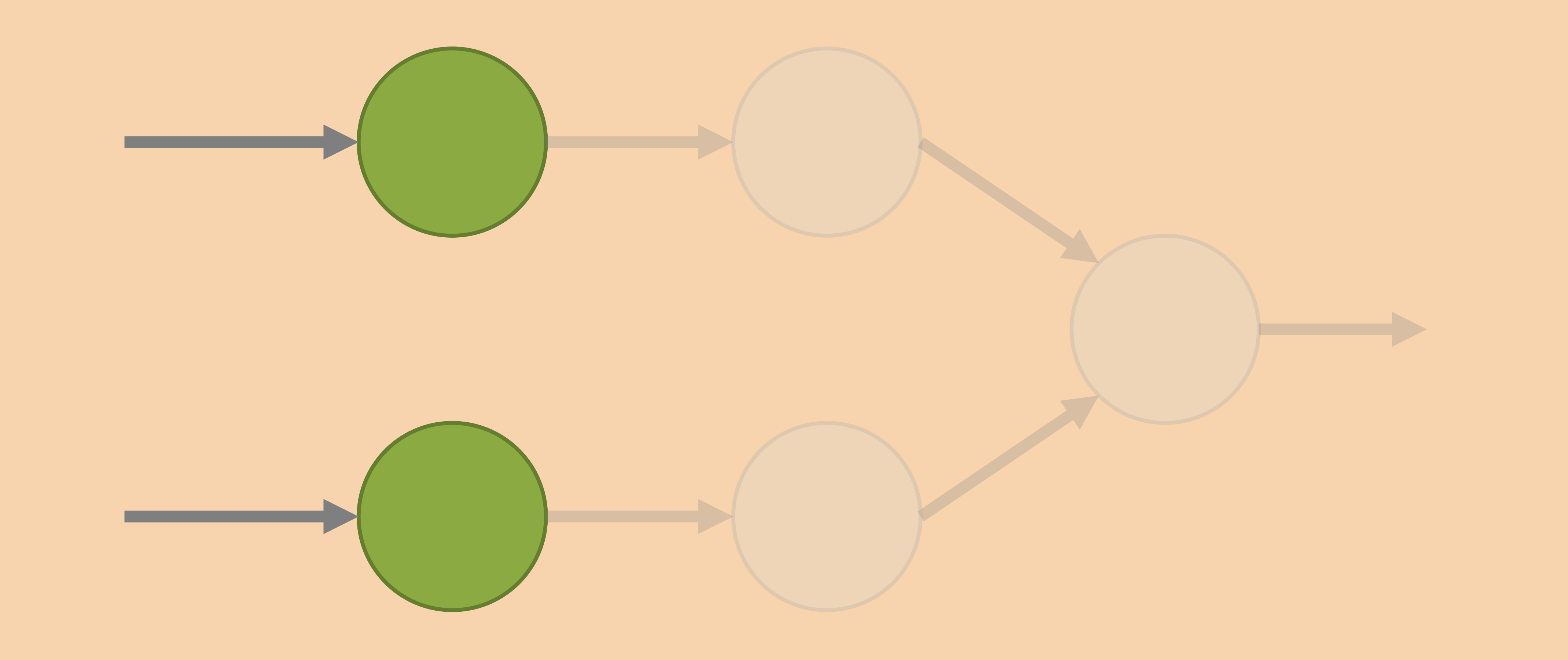
# Back to our graph



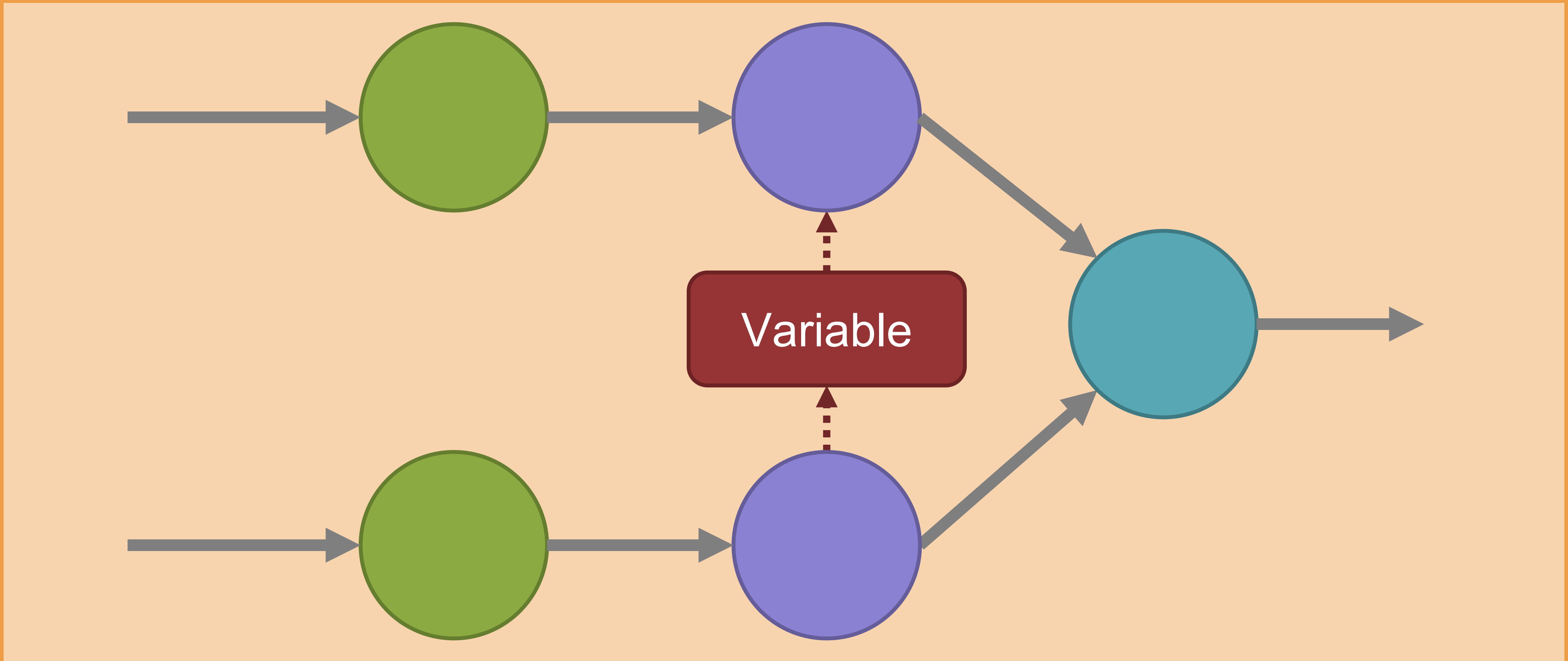
# These two nodes have no dependencies



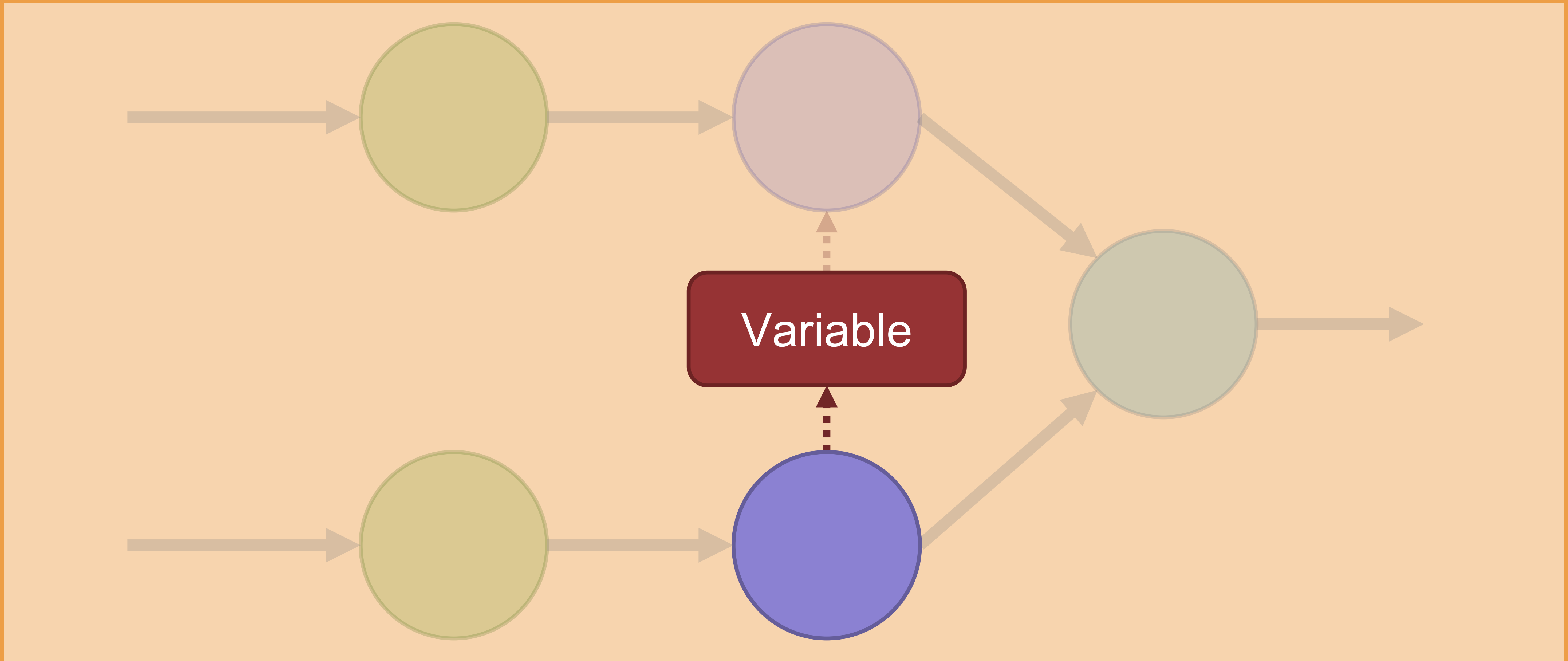
# This means they can run in any order



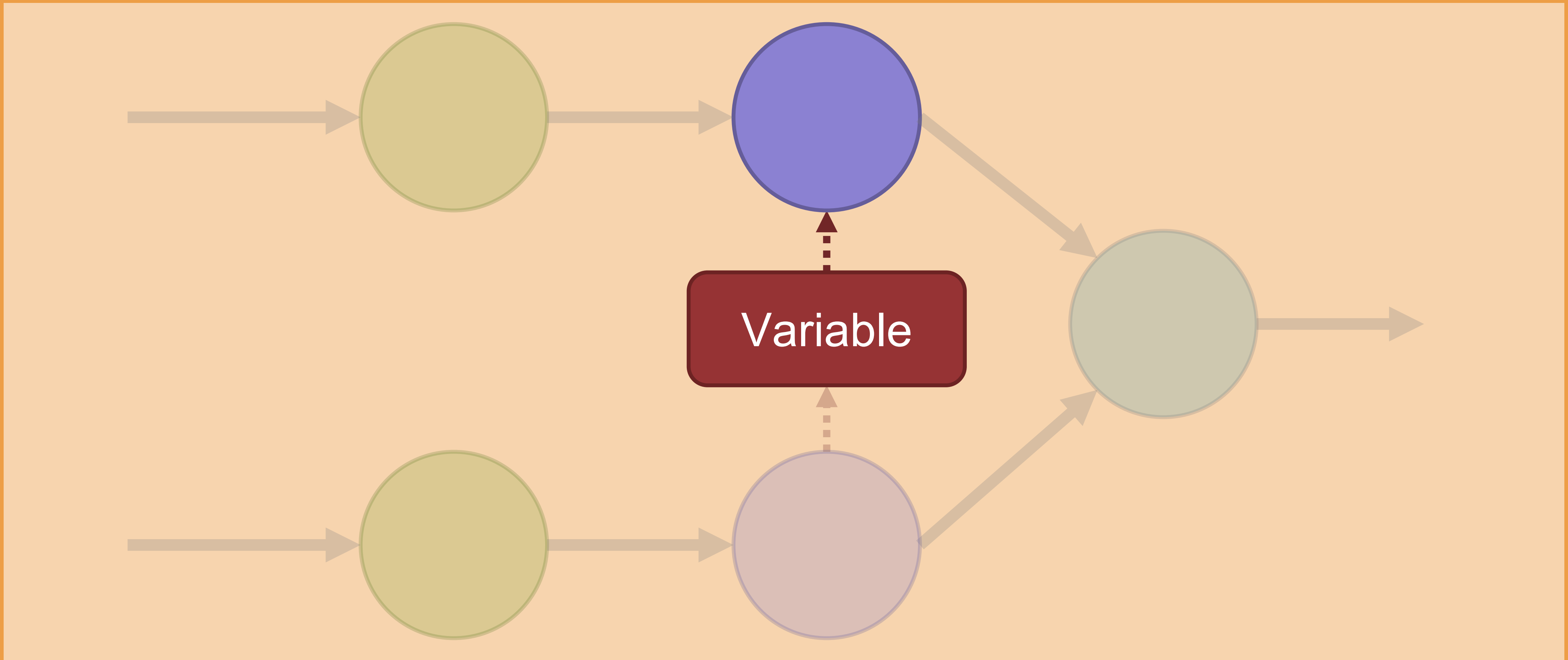
# Scenario: race-condition



# One operation changes a variable



# The other reads from that variable

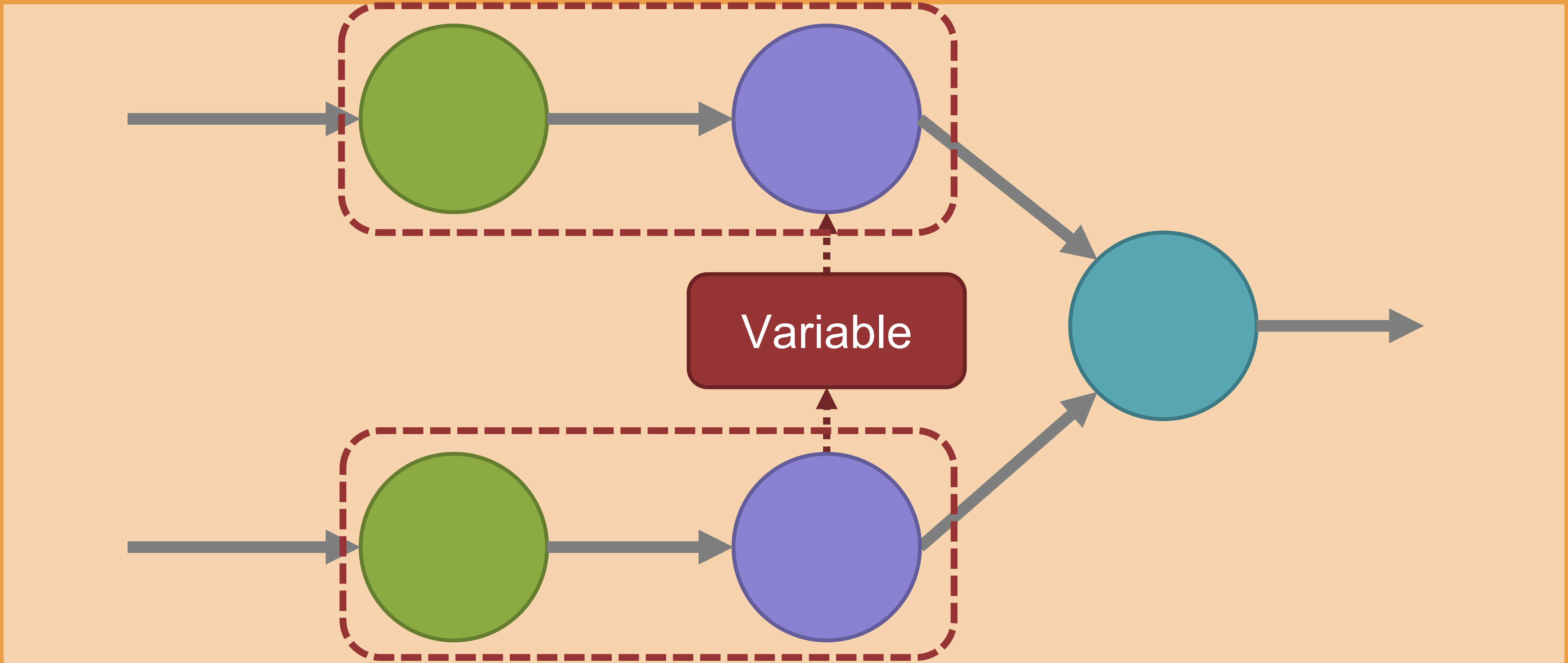


# Code might look like this…

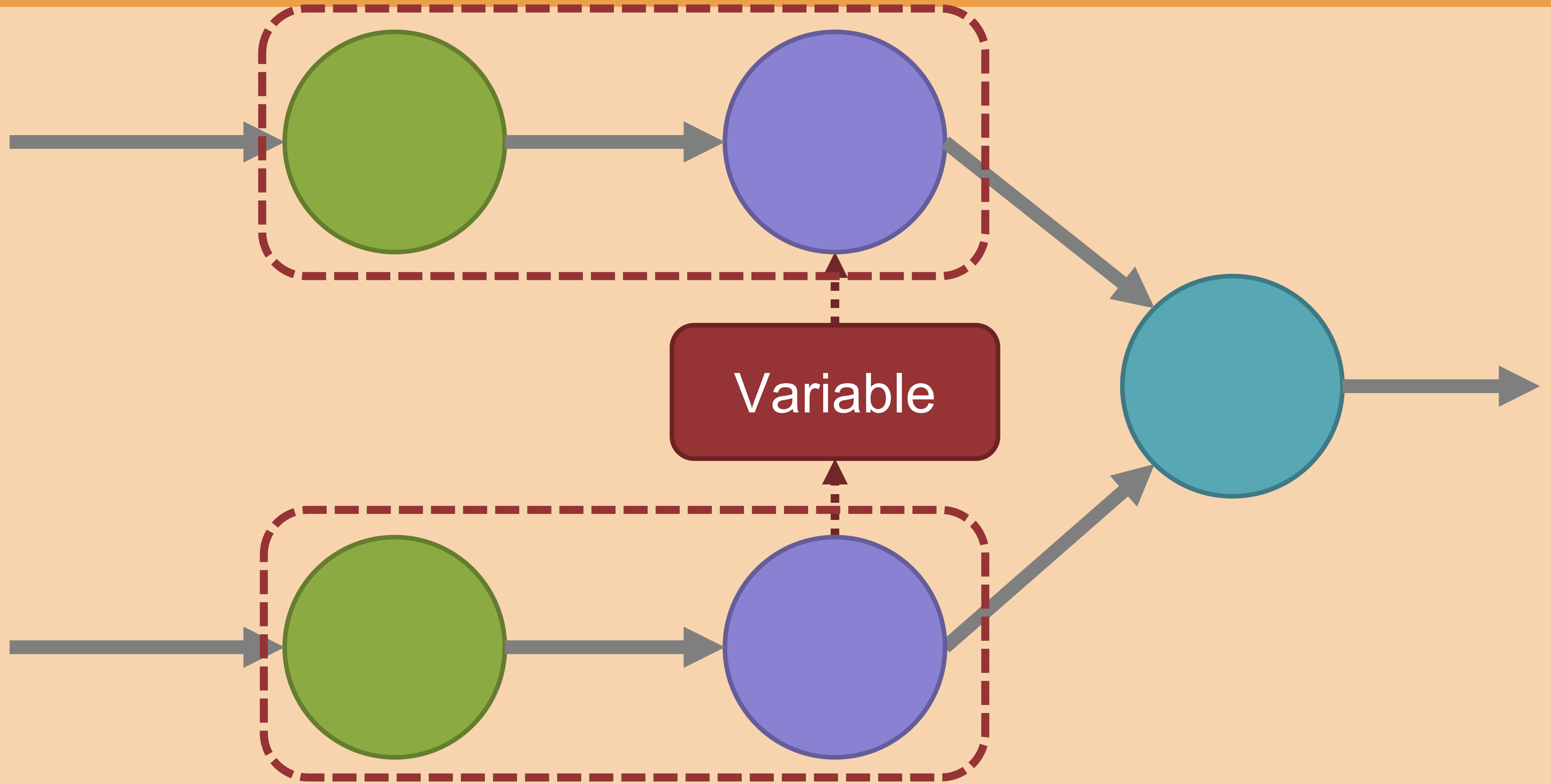
```
var = tf.Variable(…)
top = var * 2
bot = var.assign_add(2)
out = top + bot
```

Note: `assign_add()` returns value of Variable after being adjusted.

# Currently: execution order is non-deterministic



# Might lead to unexpected behavior!

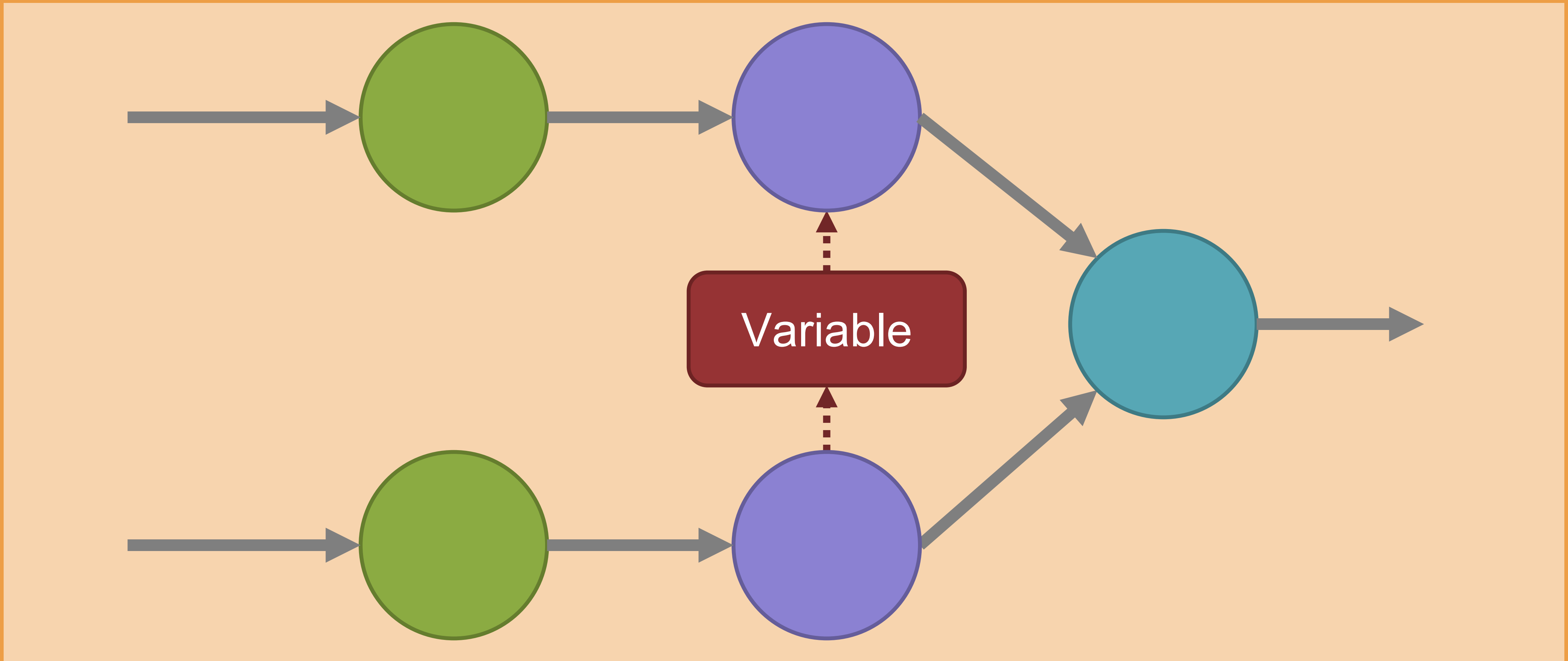


# How do we control this?

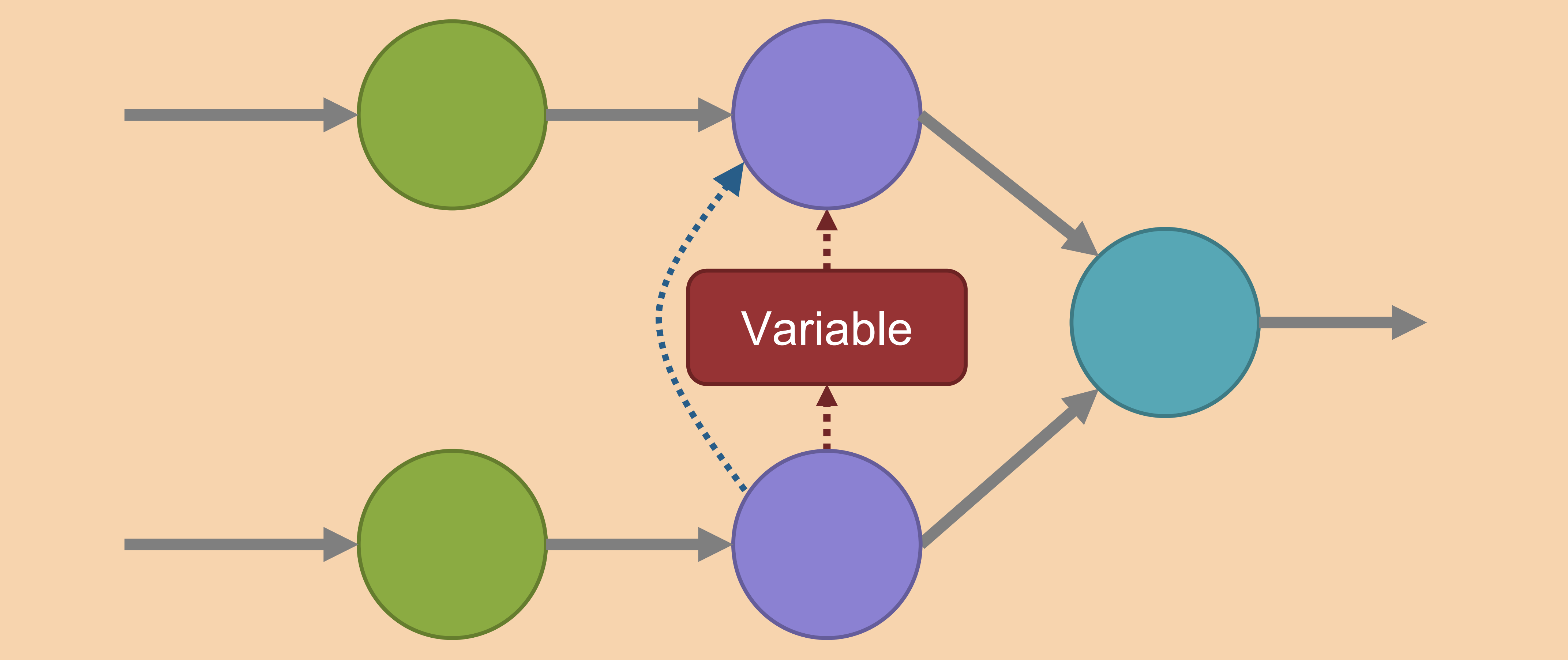
# Dependency management

- TensorFlow automatically determines dependencies
  - Basically, all of an Op’s inputs
- User can define additional dependencies
  - Forces specified operations to complete first

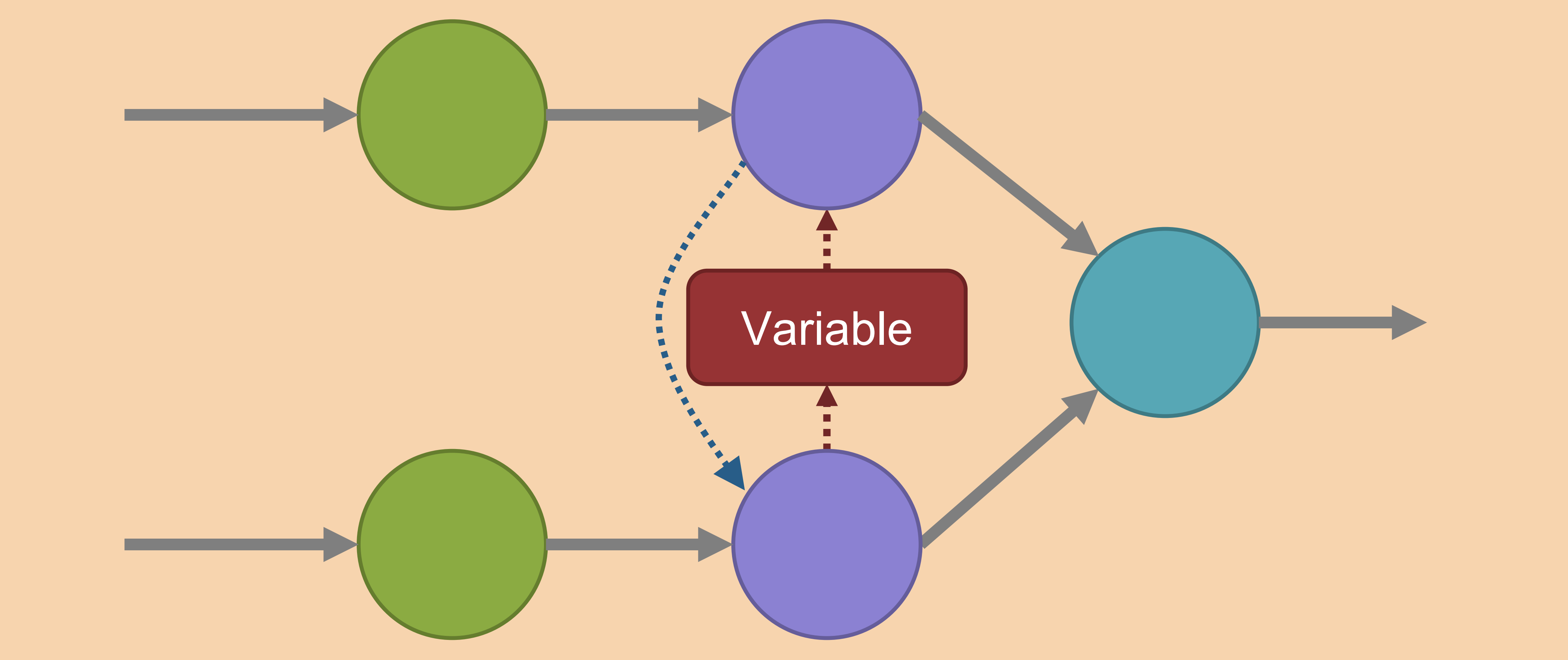
# We can control the order depending on needs



## If we want the variable to change and then read from it, make the top depend on the bottom



# If we want the read the variable before it changes, make the bottom depend on the top



# We do this with `tf.control_dependencies`

```
# Before the changes
var = tf.Variable(…)
top = var * 2
bot = var.assign_add(2)
out = top + bot
```

# We do this with `tf.control_dependencies`

```
# Force bot to wait for top
var = tf.Variable(…)
top = var * 2
with tf.control_dependencies([top]):
    bot = var.assign_add(2)
out = top + bot
```

## `tf.control_dependencies(control_inputs)`

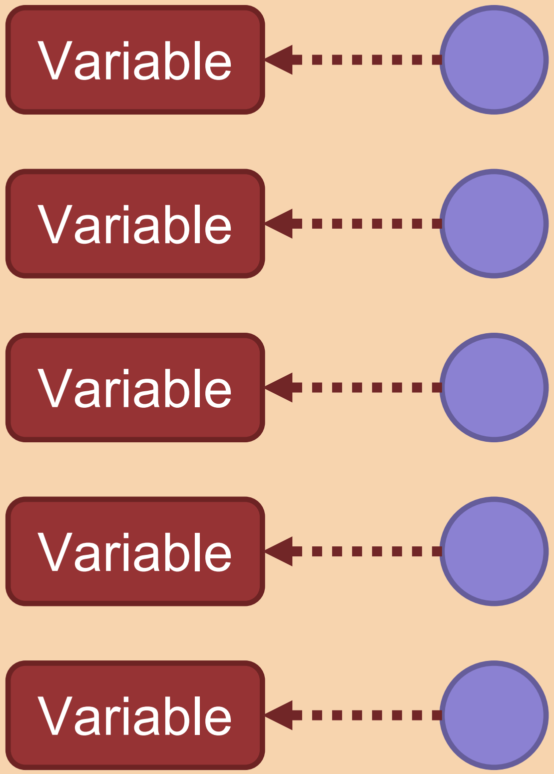
1. Put list of desired dependencies as `control_inputs`

2. Ops defined in the `with` block gain those dependencies

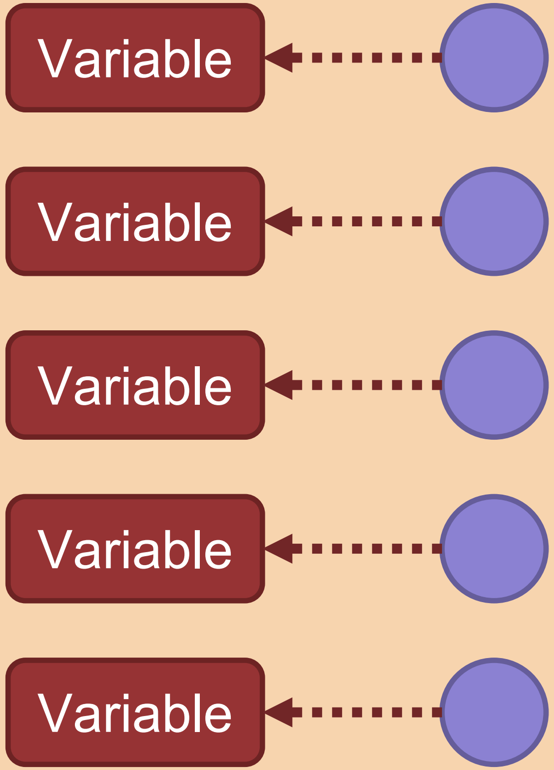
# Use cases

- Enforcing execution order
  - As shown previously
- Grouping operations
  - Run many operations with one handle
- Adding assertion statements
  - Build exceptions into your graph

# Grouping

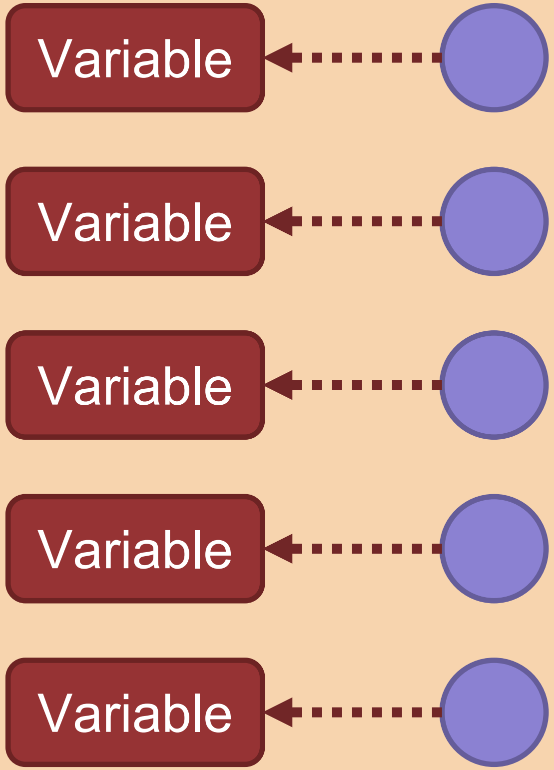


# Many variables are updated with separate ops

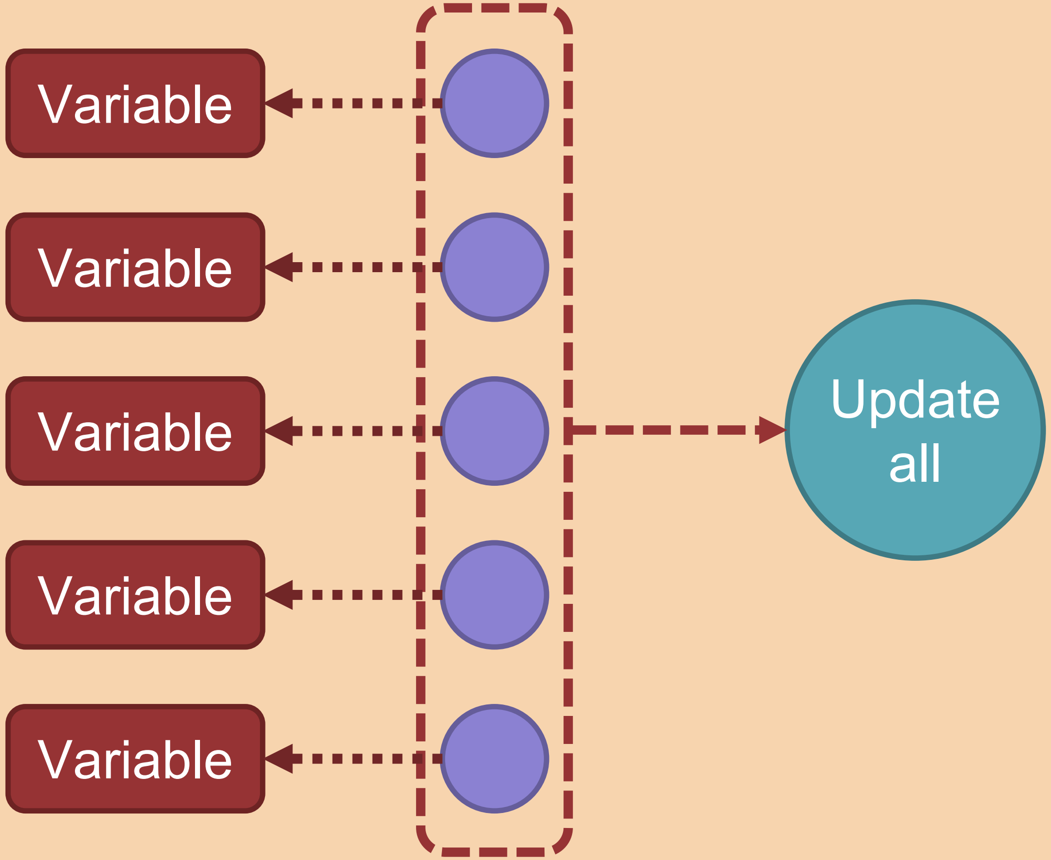


# Running each update op separately is a pain

```
val1, val2, … = sess.run([update1, update2…])
```

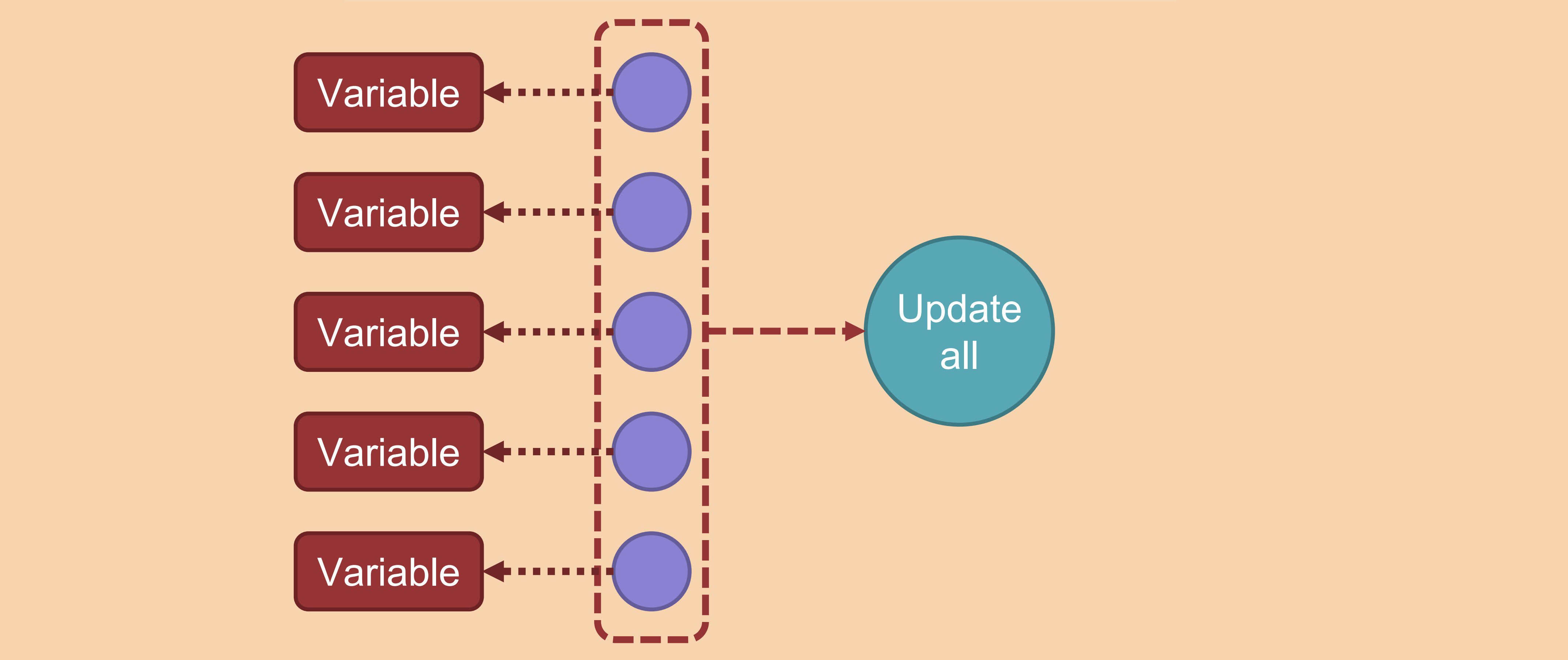


# Fix: use dummy op that depends on all updates



# This gives us simpler (and more semantic) code

```
_ = sess.run(update_all)
```



# Grouping Operations “raw”

```
updates = [update1, update2…]
with tf.control_dependencies(updates):
    update_all = tf.no_op()
```

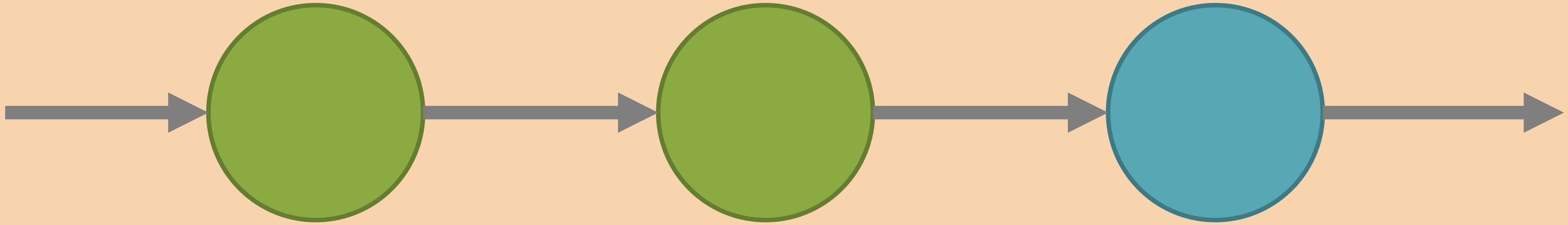
TensorFlow has a built-in helper to make this cleaner

# tf.group

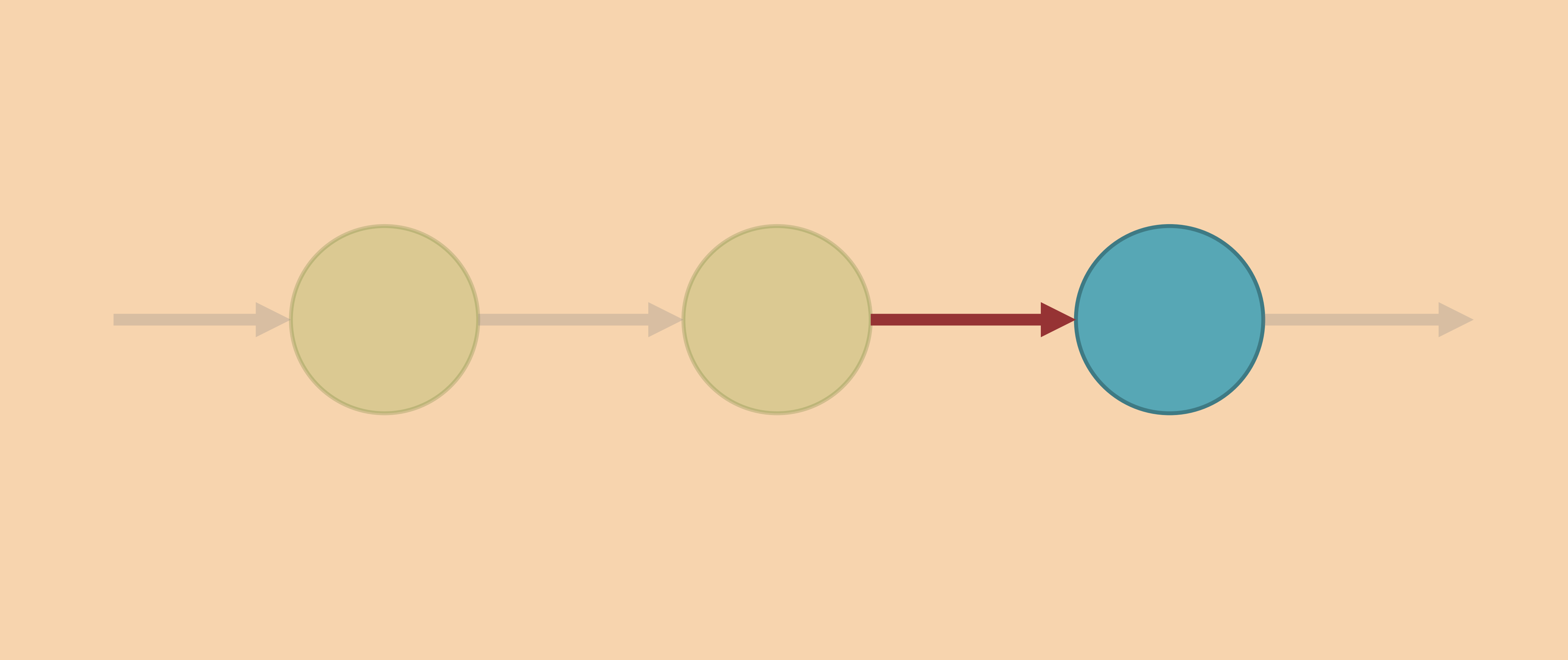
```
updates = [update1, update2…]
update_all = tf.group(*updates)
```

- Uses `tf.control_dependencies` under the hood
- Has extra built-in functionality
  - Automatically groups operations by device (CPU, GPU1, GPU2, etc)

# Assertions



# Need to validate tensor going into this op



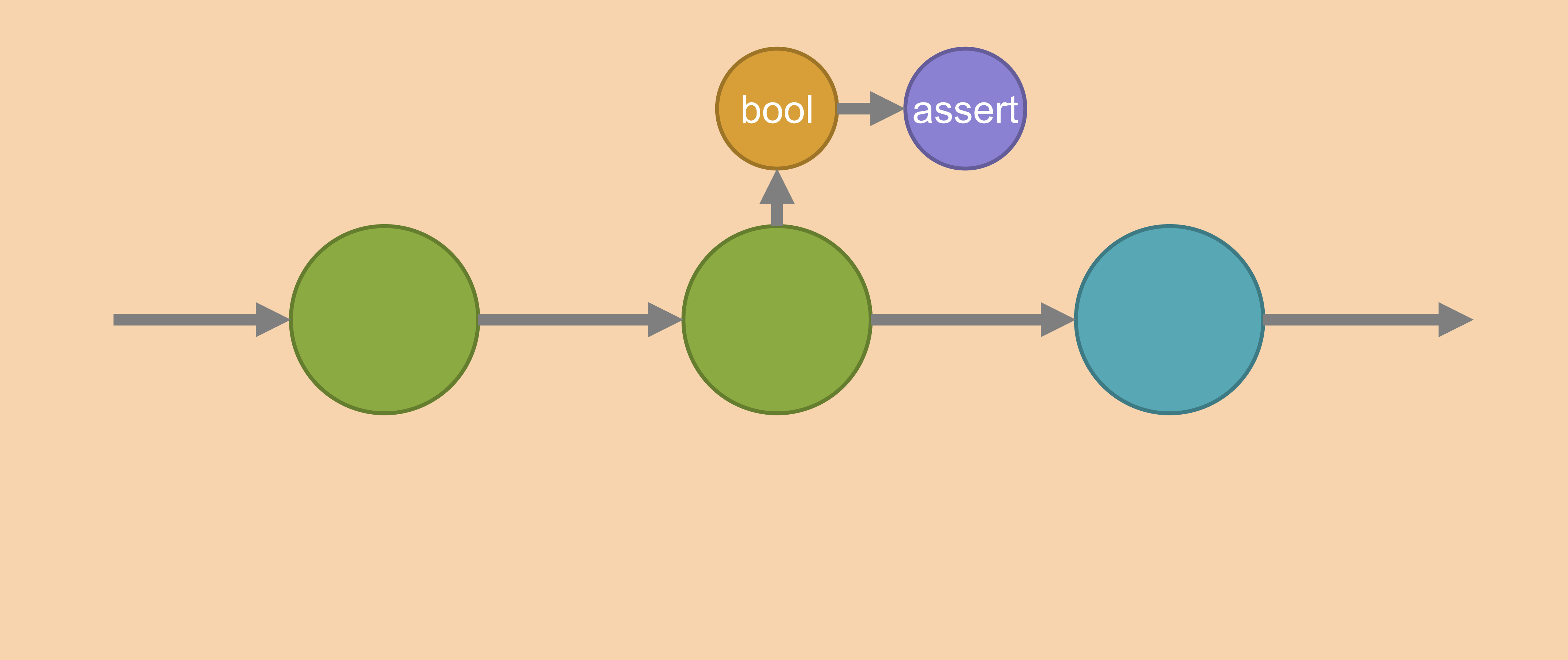
# Pain in the ass version:

```
_, check_me = sess.run([train, check_op])
if not validate(check_me):
    raise ValueError(…)
```

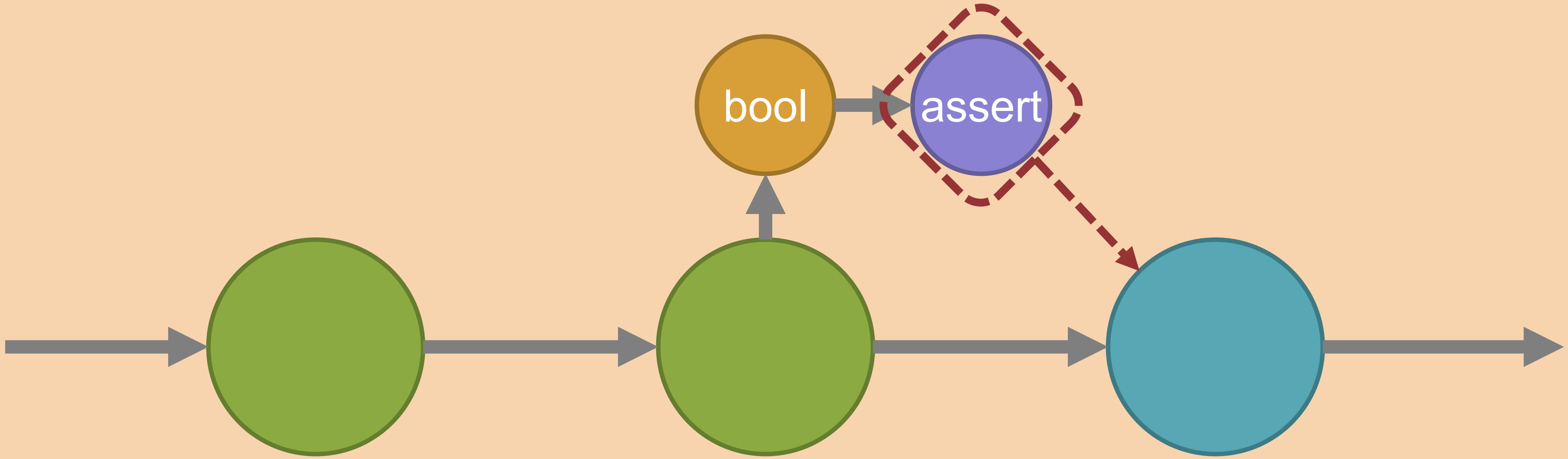
Gets worse the more checks you need to make

Better: validate as the graph runs

# `tf.assert` raises exception if check value is `False`



# Make it a dependency to run before critical op



# Simple Assertion

```
check_me = tf.multiply(…)
assert_op = tf.assert(check_me != 0, check_me)
with tf.control_dependencies([assert_op]):
    next_op = tf.divide(10, check_me)
```

Required arguments for tf.assert:

1. Boolean check value
2. A tensor to print in the error message

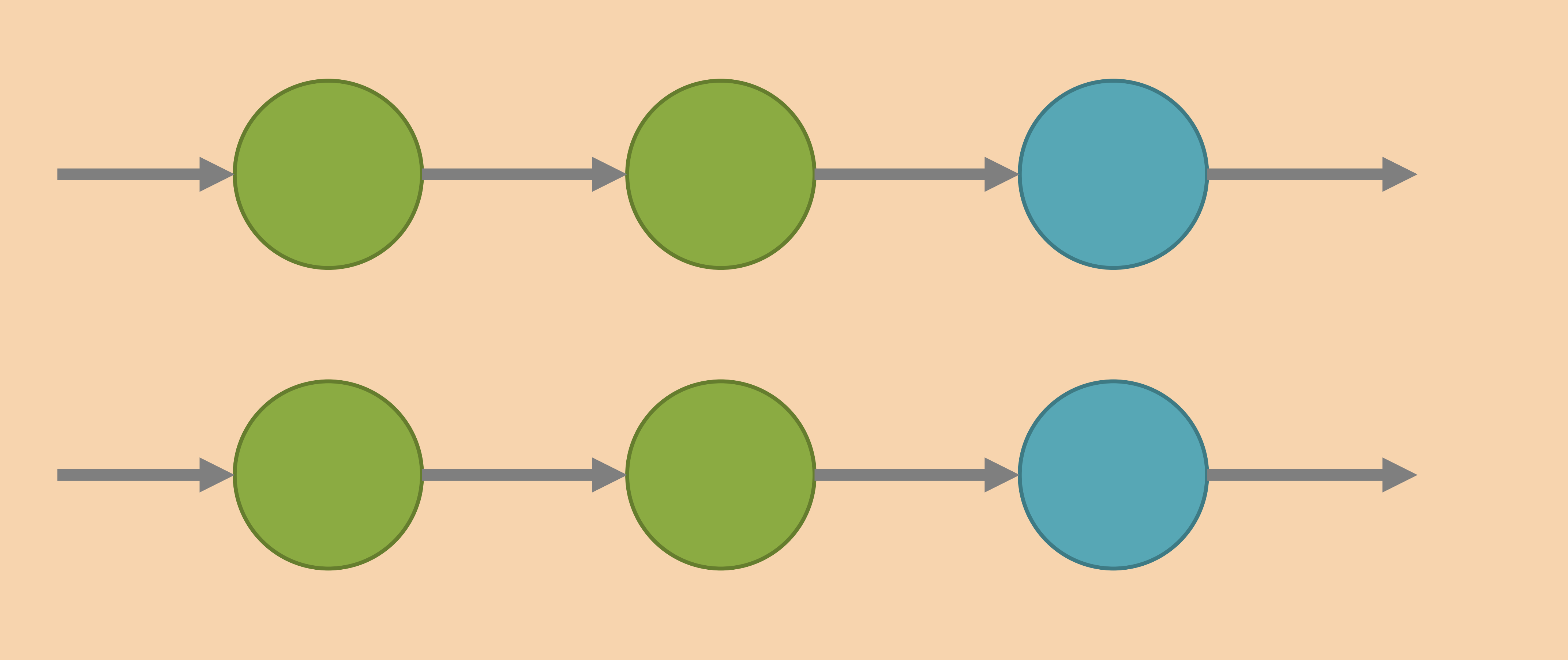
# Common scenario: check for `NaN` or `Inf`

```
check_me = tf.matmul(…)
assert_op = tf.check_numerics(check_me, ‘It broke!’)
with tf.control_dependencies([assert_op]):
    next_op = …
```

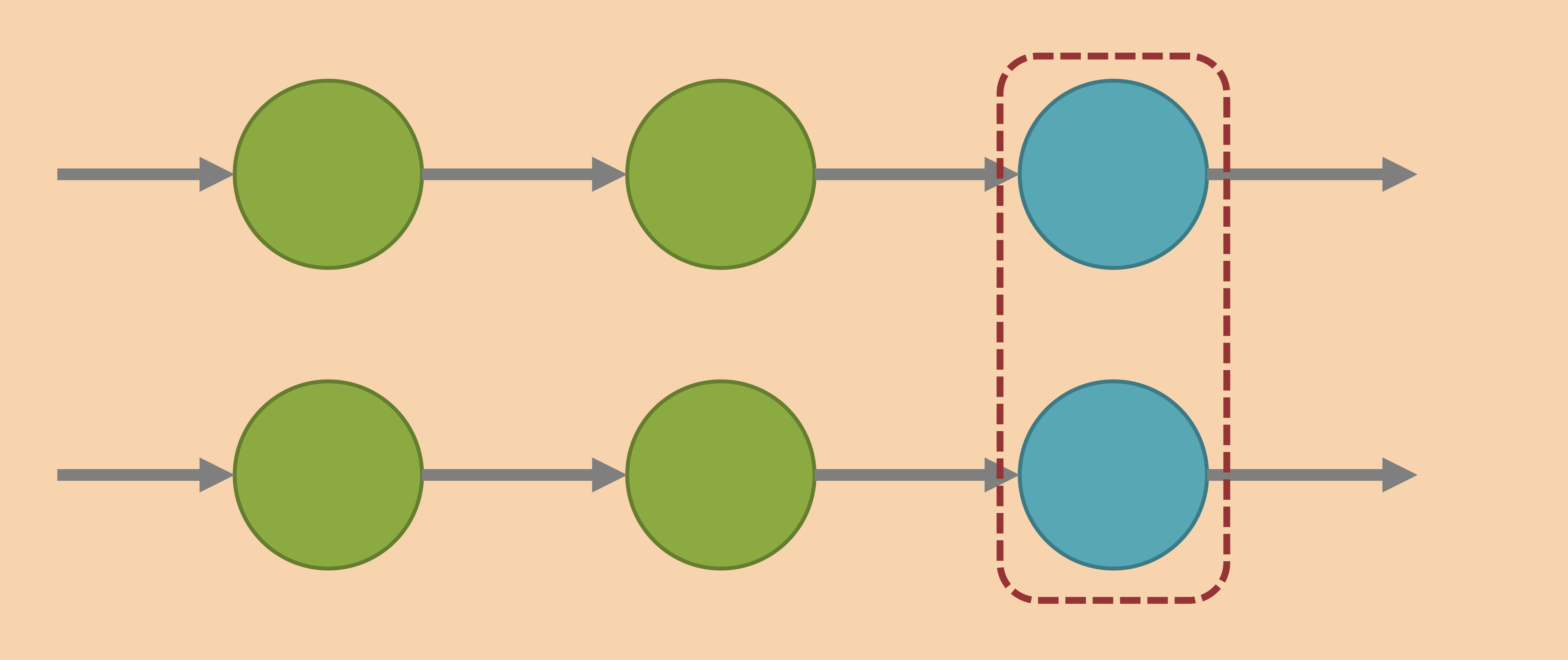
Many built-in assertion helpers:

1. https://www.tensorflow.org/api_guides/python/check_ops
2. https://www.tensorflow.org/api_guides/python/control_flow_ops#Debugging_Operations

# One last example: synchronization



# Want to wait for *both* to finish before moving on



# tf.tuple

```
wait_1 = tf.some_op(…)
wait_2 = tf.another_op(…)
sync_1, sync_2 = tf.tuple([wait_1, wait_2])
```

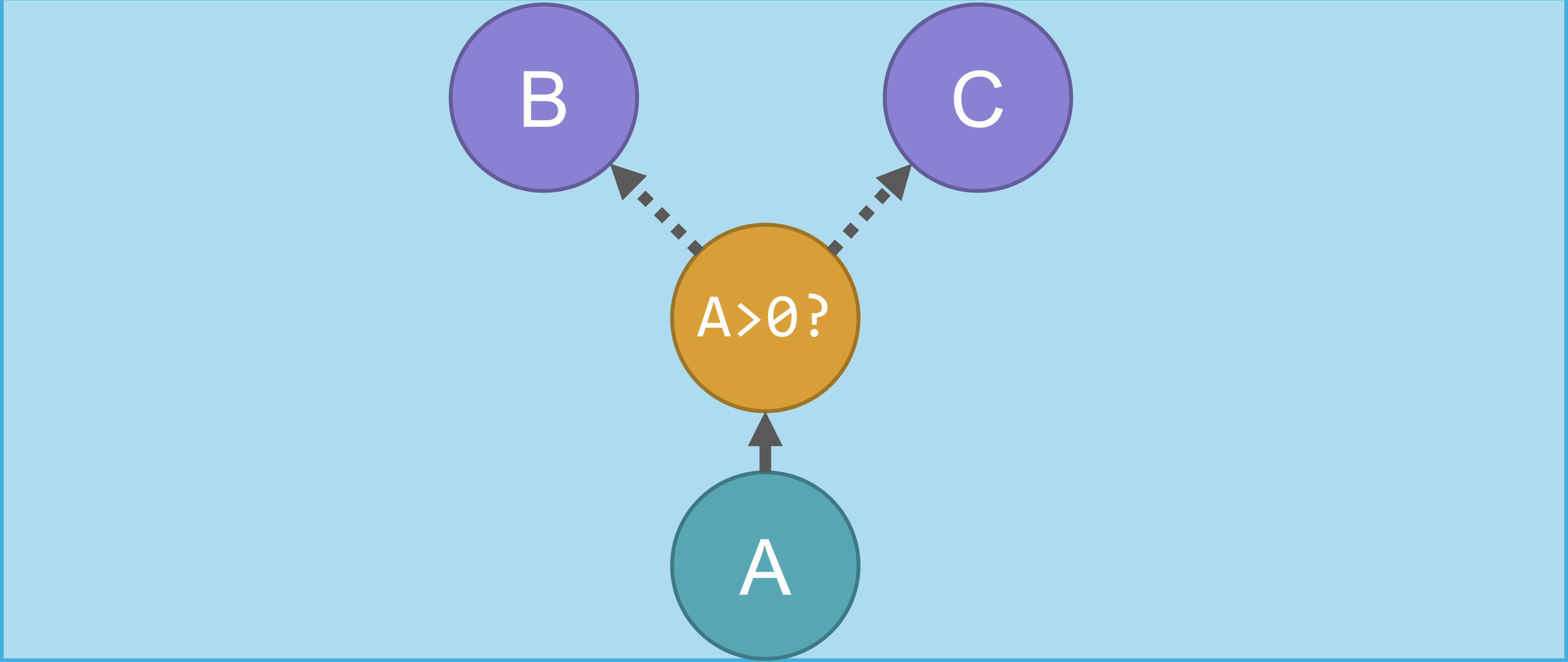
Note: TensorFlow already waits for dependencies

`tf.tuple` is generally reserved for unique requirements

# CONDITIONAL LOGIC

# Idea: different ops based on intermediate results

# Like our example from the intro



# TensorFlow offers two Ops for conditionals

- `tf.cond`
  - Like an `if/else` statement
- `tf.case`
  - Like a `case` statement

# Using `tf.cond`

```
tf.cond(pred, run_if_true, run_if_false)
```

`tf.cond` takes three required arguments

# Using `tf.cond`

```
tf.cond(pred, run_if_true, run_if_false)
```

- The first is a scalar boolean predicate
- Switch telling TensorFlow which branch to run

# Using `tf.cond`

```
tf.cond(pred, run_if_true, run_if_false)
```

- The second is a callable (function, lambda, etc)
- Should take no input, and return zero or more tensors
- Runs if the predicate is true

# Using `tf.cond`

```
tf.cond(pred, run_if_true, run_if_false)
```

- The last is also a callable
- Similar to previous input
- Runs if the predicate is false

# Example

```
pred = a < b
def run_if_true():
    return tf.add(3, 3)
def run_if_false():
    return tf.square(3)
out = tf.cond(pred, run_if_true, run_if_false)
```

# Define predicate

```
pred = a < b
def run_if_true():
    return tf.add(3, 3)
def run_if_false():
    return tf.square(3)
out = tf.cond(pred, run_if_true, run_if_false)
```

# “True” callable

```
pred = a < b
def run_if_true():
    return tf.add(3, 3)
def run_if_false():
    return tf.square(3)
out = tf.cond(pred, run_if_true, run_if_false)
```

# “False” callable

```
pred = a < b
def run_if_true():
    return tf.add(3, 3)
def run_if_false():
    return tf.square(3)
out = tf.cond(pred, run_if_true, run_if_false)
```

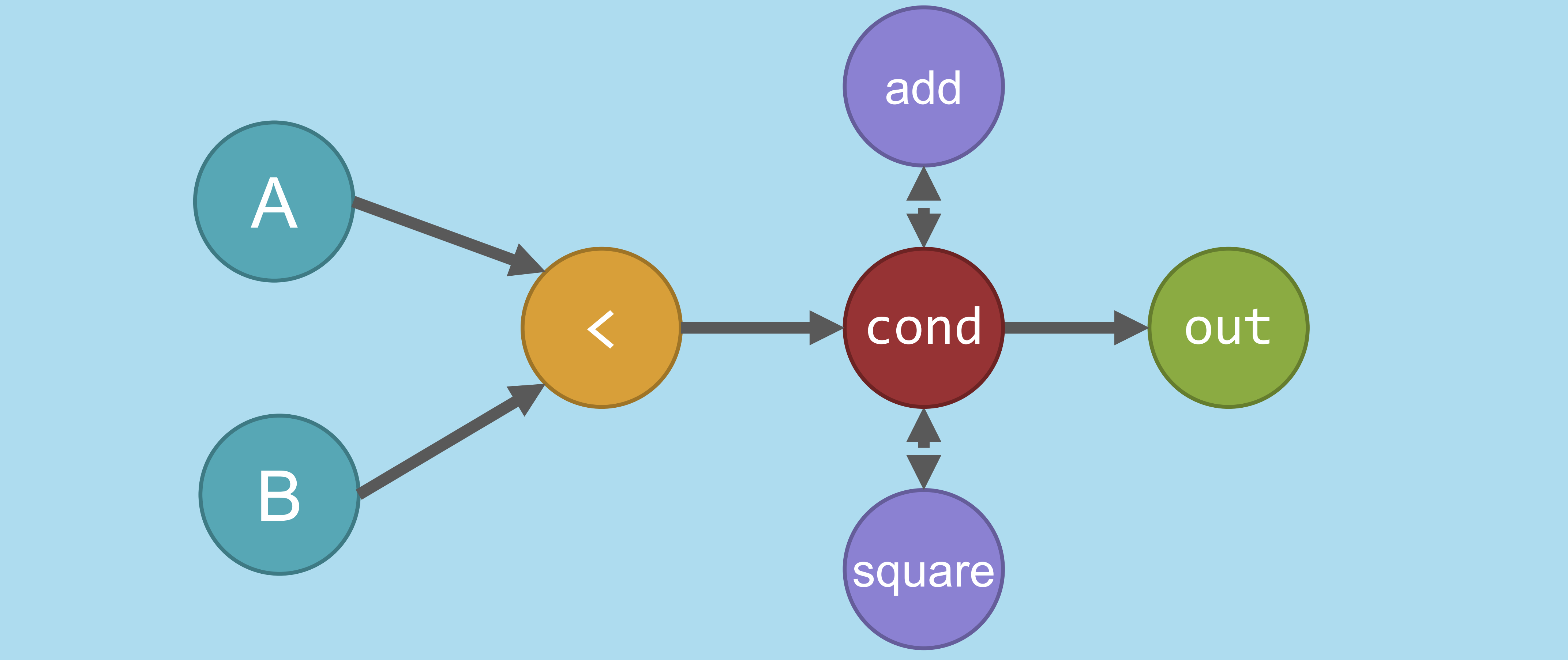
# Put it all together

```
pred = a < b
def run_if_true():
    return tf.add(3, 3)
def run_if_false():
    return tf.square(3)
out = tf.cond(pred, run_if_true, run_if_false)
```

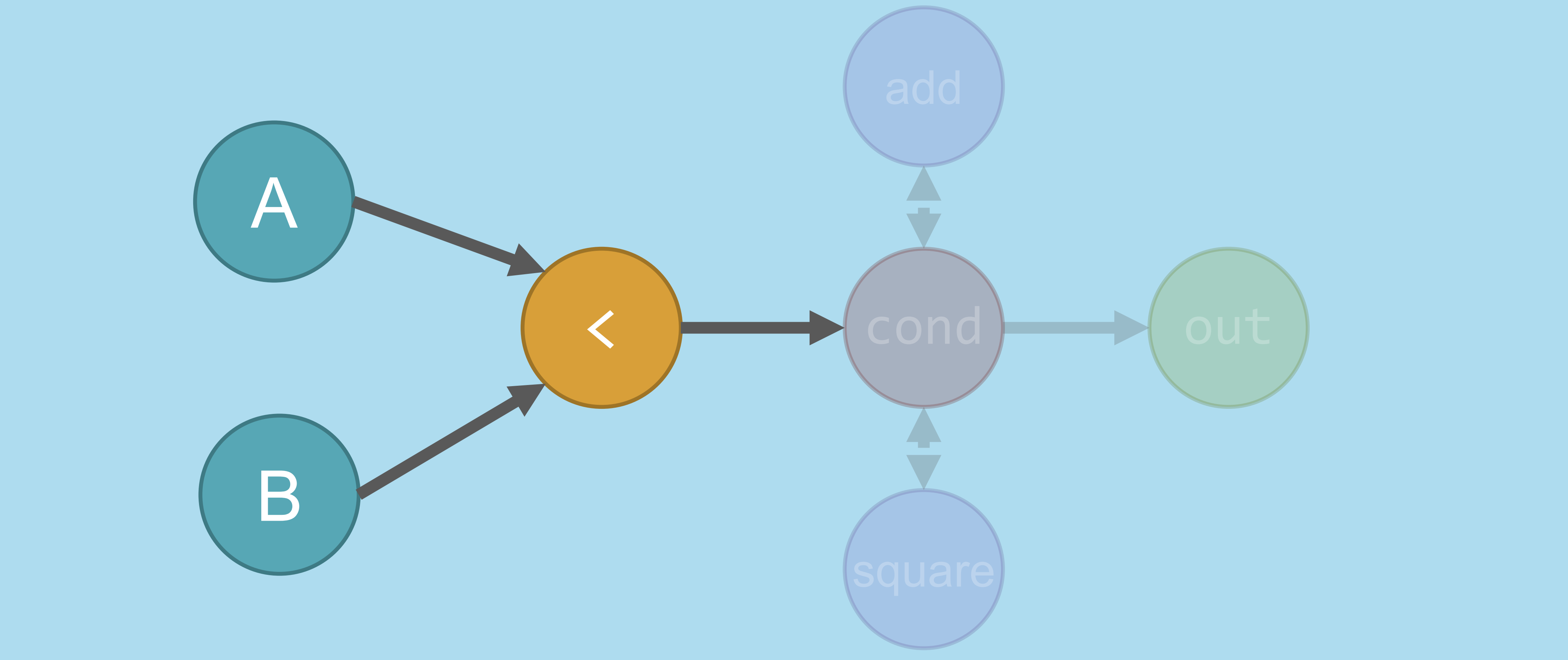
# You might one-liner it for simple uses

```
tf.cond(a < b, lambda: tf.add(3, 3), lambda: tf.sqaure(3))
```

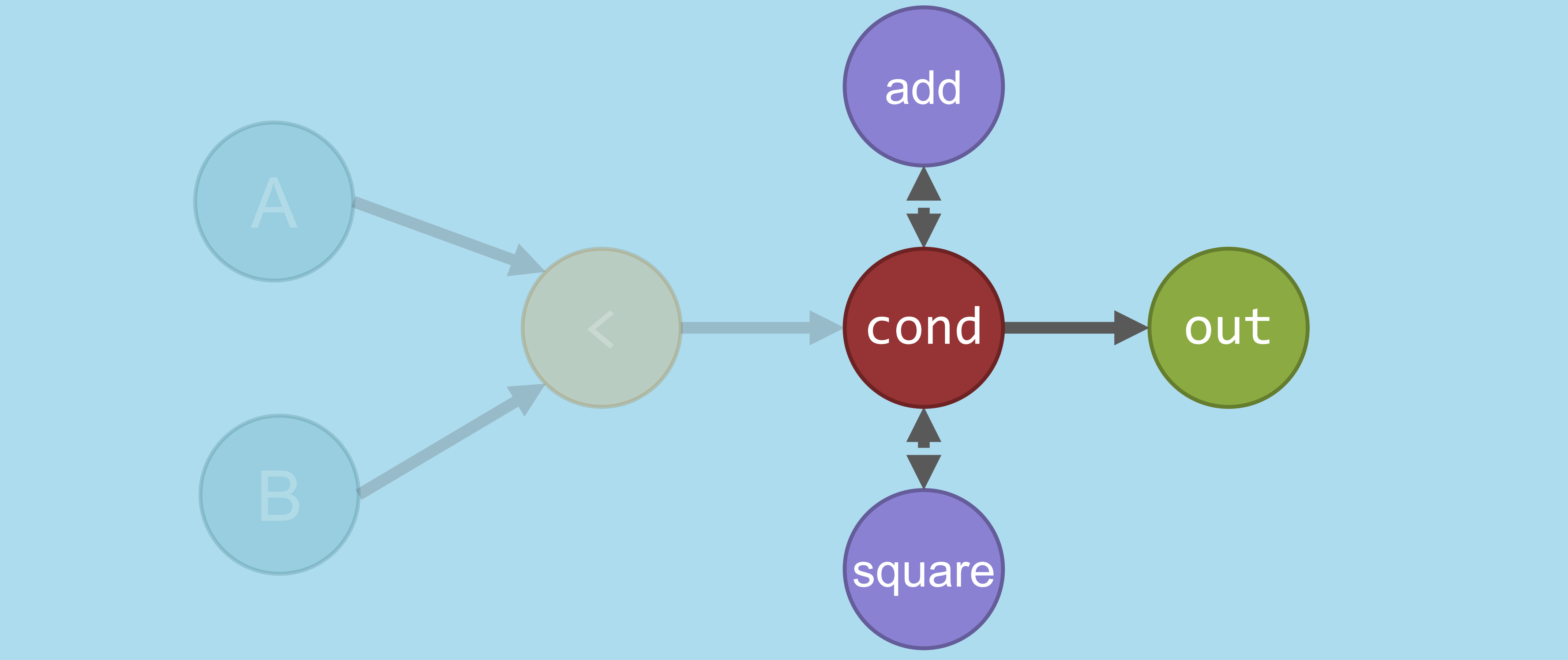
# The graph we just defined looks like this



# The less-than operation outputs a boolean



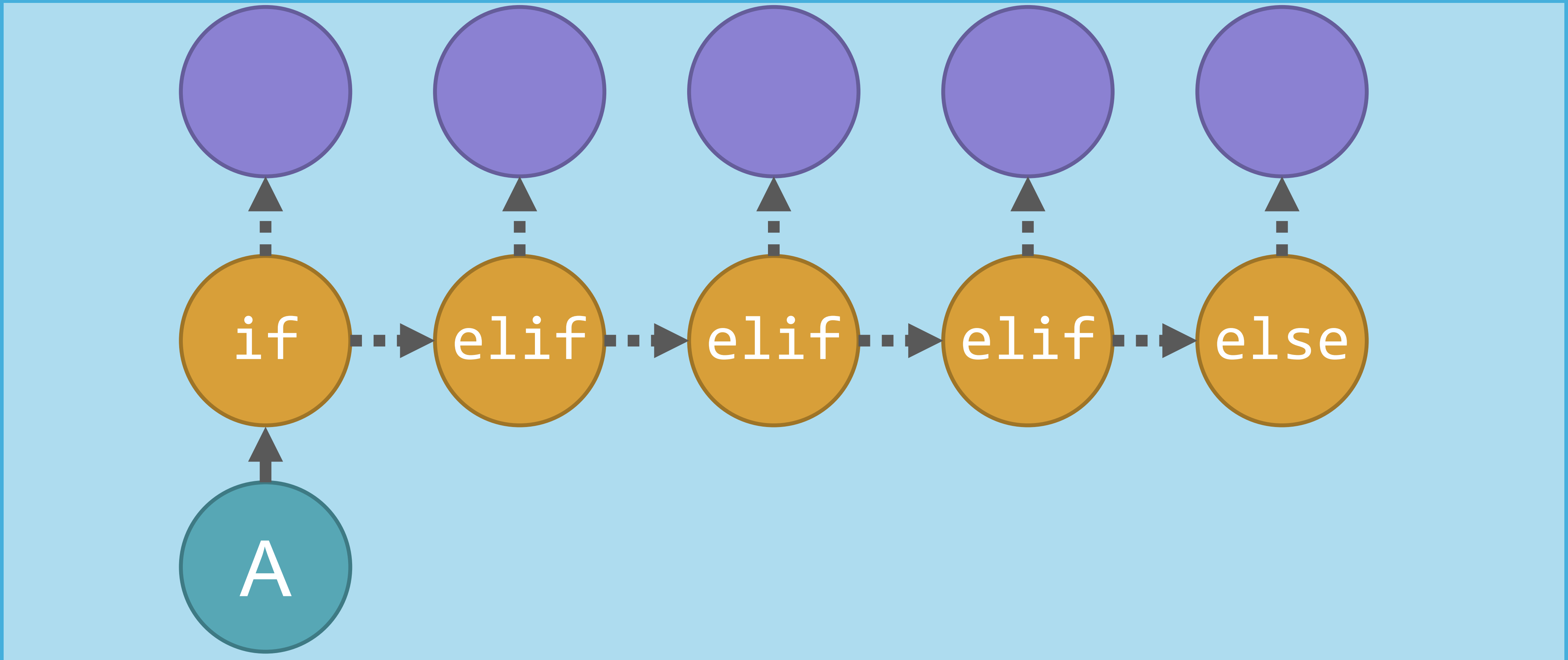
# `tf.cond` selects the right output to pass along



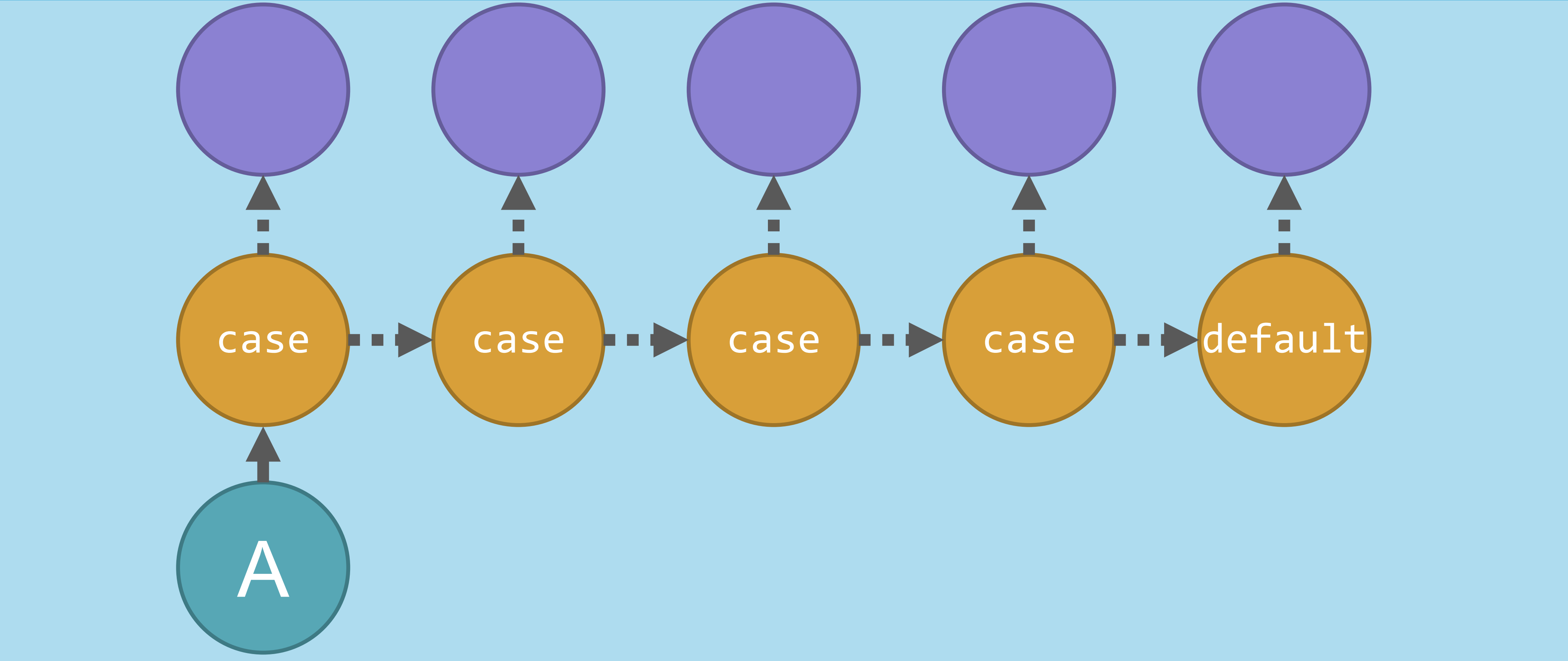
# `tf.cond` **notes:**

- Both callables' return signatures *must* match
  - Same number of tensors with the same type
- External ops needed for either branch always run
  - Place as many ops inside the callables as possible

# Now need choose from more than two actions



# Same graph, case statement syntax



Instead of chaining `tf.cond` over and over,

we can **use a single `tf.case`**

# Using `tf.case`

```
tf.case(pred_fn_pairs, default)
```

`tf.case` takes two required arguments

# Using `tf.case`

```
tf.case(pred_fn_pairs, default)
```

- The first is a list of tuple pairs `(predicate, callable)`
- Maps boolean predicates to potential operations to run
- It can also be a dictionary: `{pred: callable}`

# Using `tf.case`

```
tf.case(pred_fn_pairs, default)
```

- The second is a callable, as we've seen before
- Runs if none of the predicates are true

# Basic example:

```
a = (prev < 0,  lambda: prev + 3)
b = (prev < 10, lambda: prev * 3)
c = (prev < 20, lambda: prev - 3)
pairs = [a, b, c]
default = lambda: prev / 3
out = tf.case(pairs, default)
```

# Define tuple pairs of predicates/callables

```
a = (prev < 0,  lambda: prev + 3)
b = (prev < 10, lambda: prev * 3)
c = (prev < 20, lambda: prev - 3)
pairs = [a, b, c]
default = lambda: prev / 3
out = tf.case(pairs, default)
```

# Define a default callable

```
a = (prev < 0,  lambda: prev + 3)
b = (prev < 10, lambda: prev * 3)
c = (prev < 20, lambda: prev - 3)
pairs = [a, b, c]
default = lambda: prev / 3
out = tf.case(pairs, default)
```

# Create the op

```
a = (prev < 0,  lambda: prev + 3)
b = (prev < 10, lambda: prev * 3)
c = (prev < 20, lambda: prev - 3)
pairs = [a, b, c]
default = lambda: prev / 3
out = tf.case(pairs, default)
```

## `tf.cond` **notes:**

- All callables' return signatures *must* match (like `tf.cond`)
  - Same number of tensors with the same type
- Only one callable will run
  - As if each `case` has a `break` statement
- Can also pass in attribute `exclusive` (defaults to `False`)
  - Makes op throw exception if more than one predicate is true

## General notes on conditional logic:

- Ops on non-selected branches are not run
  - Great if heavy computation only needs to happen sometimes
  - Example: stochastic depth
- TensorFlow differentiates through the selected path
- For TensorBoard: `cond` and `case` can get ugly
  - Use `tf.name_scope` or `tf.variable_scope` inside callables

# WHILE LOOPS

## Common uses of loops in TensorFlow

1. Feeding intermediate results back into graph

2. “Unrolling” a loop of operations

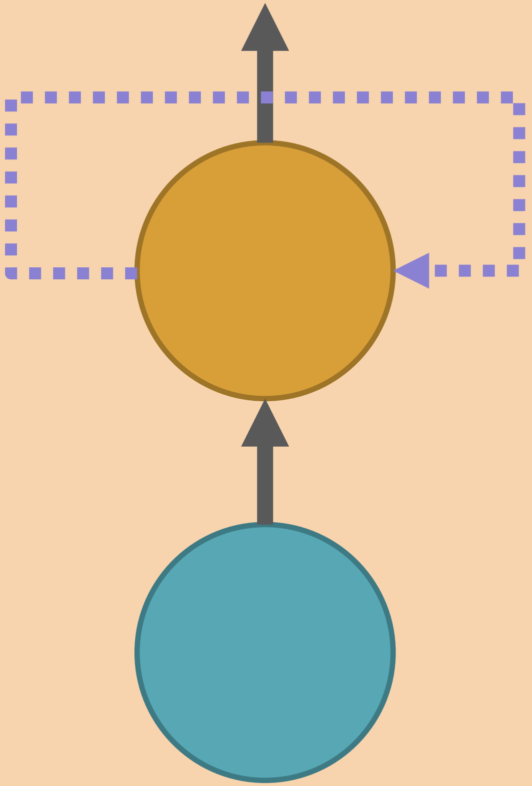
# Feeding results back into graph

```
my_op = tf.some_op(prev)
…
res = start_val
for i in range(…):
    feed_dict = {prev: res}
    res = sess.run(my_op, feed_dict)
```

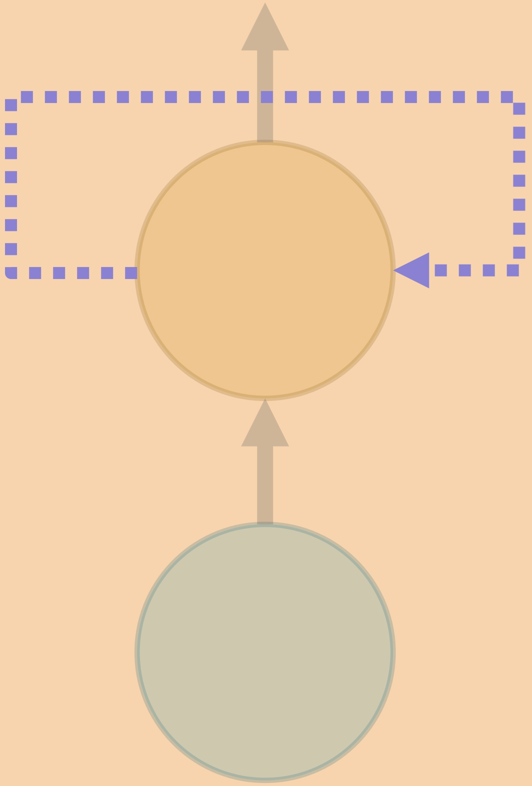
# Feeding results back into graph

```
my_op = tf.some_op(prev)
…
res = start_val
for i in range(…):
    feed_dict = {prev: res}
    res = sess.run(my_op, feed_dict)
```

# The graph looks like this



# This loop occurs in the Python layer

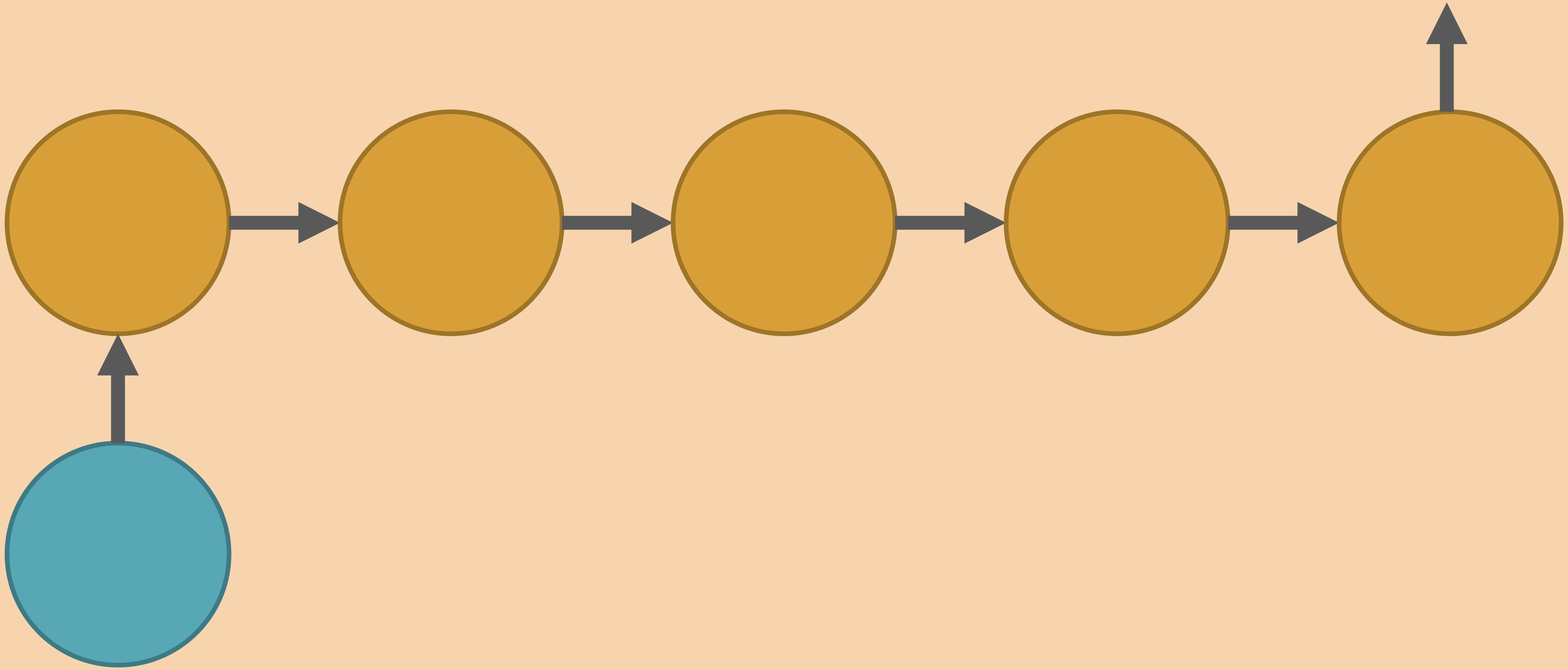


# “Unrolling” loops of operations

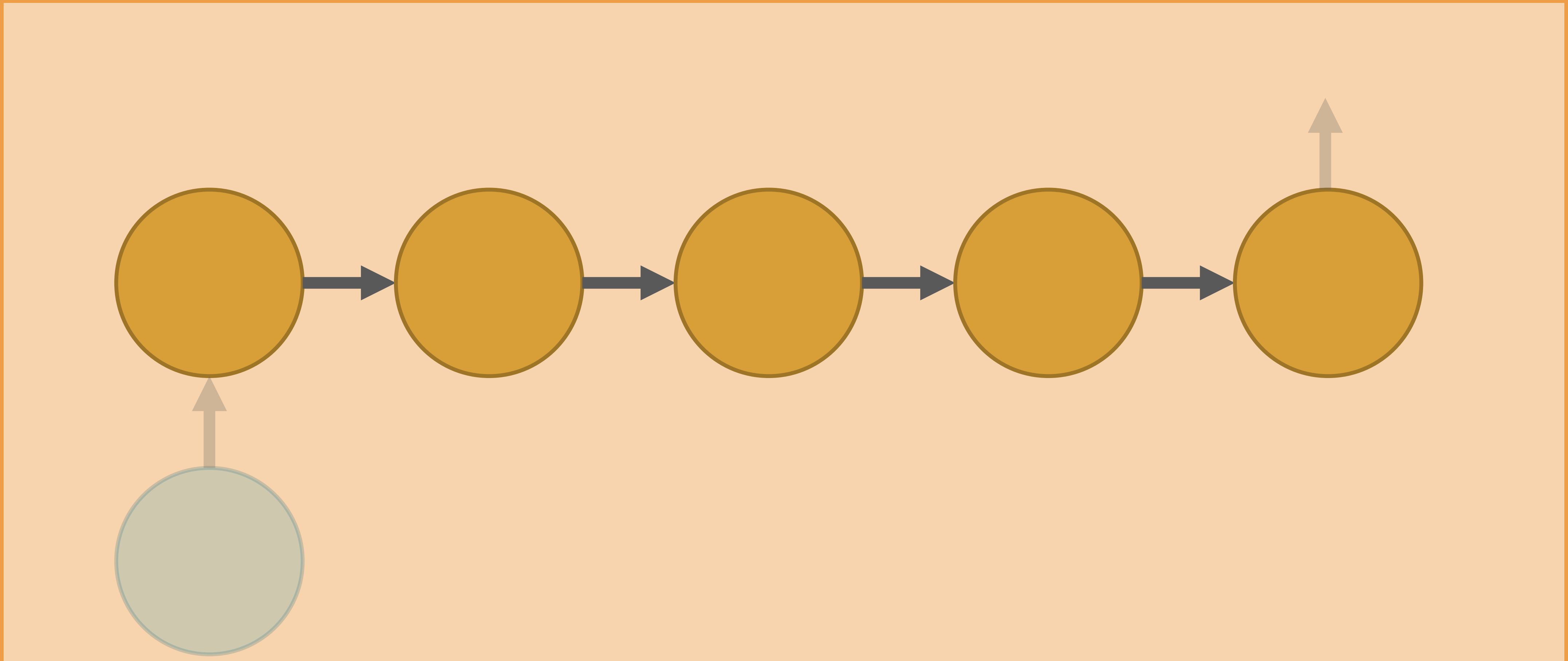
```
my_op = tf.some_op(prev)
for i in range(…):
    my_op = tf.some_op(my_op)
```

Basically, create a bunch of ops in the graph

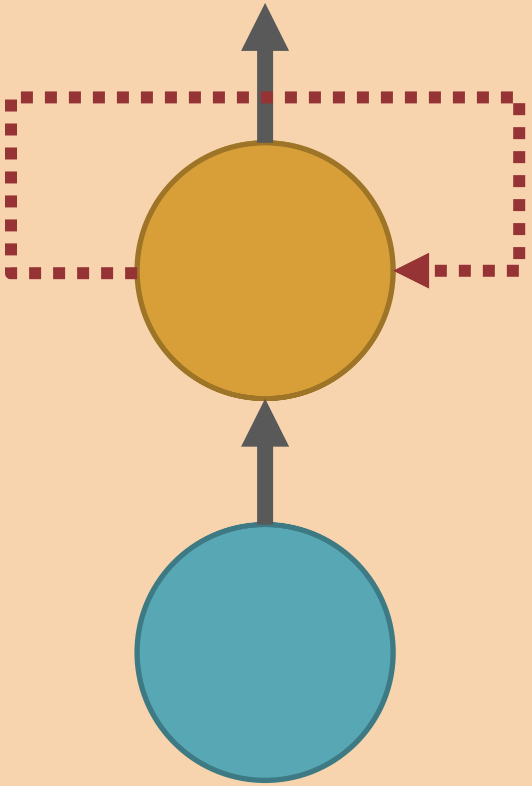
# The unrolled graph looks like this



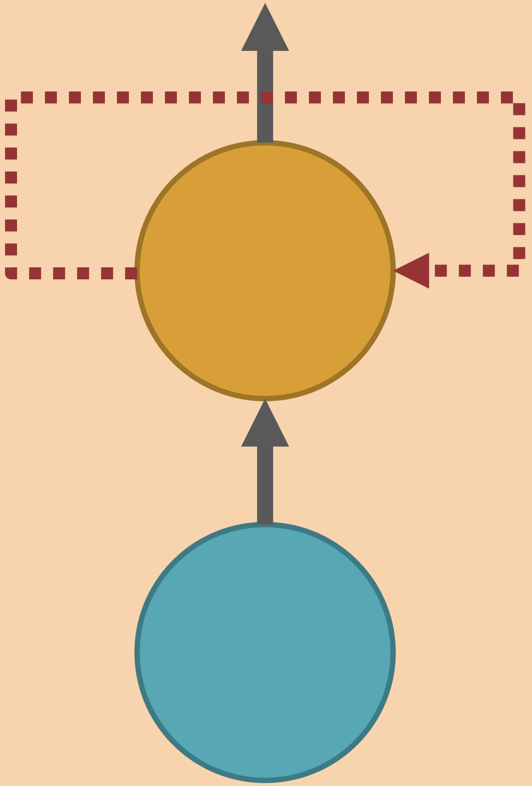
# Each additional op adds overhead



# Ideally: loop in C++ layer with minimal added ops



# `tf.while_loop` is what we're looking for!



# `tf.while_loop`

```
tf.while_loop(cond, body, loop_vars)
```

`tf.while_loop` takes three required arguments

# `tf.while_loop`

```
tf.while_loop(cond, body, loop_vars)
```

- Let's start with the last: `loop_vars`
- List/tuple of tensors used *in the first iteration* of the while loop
  - The documentation doesn't make this super clear
- These are passed to both the condition and body (up next)

# tf.while_loop

```
tf.while_loop(cond, body, loop_vars)
```

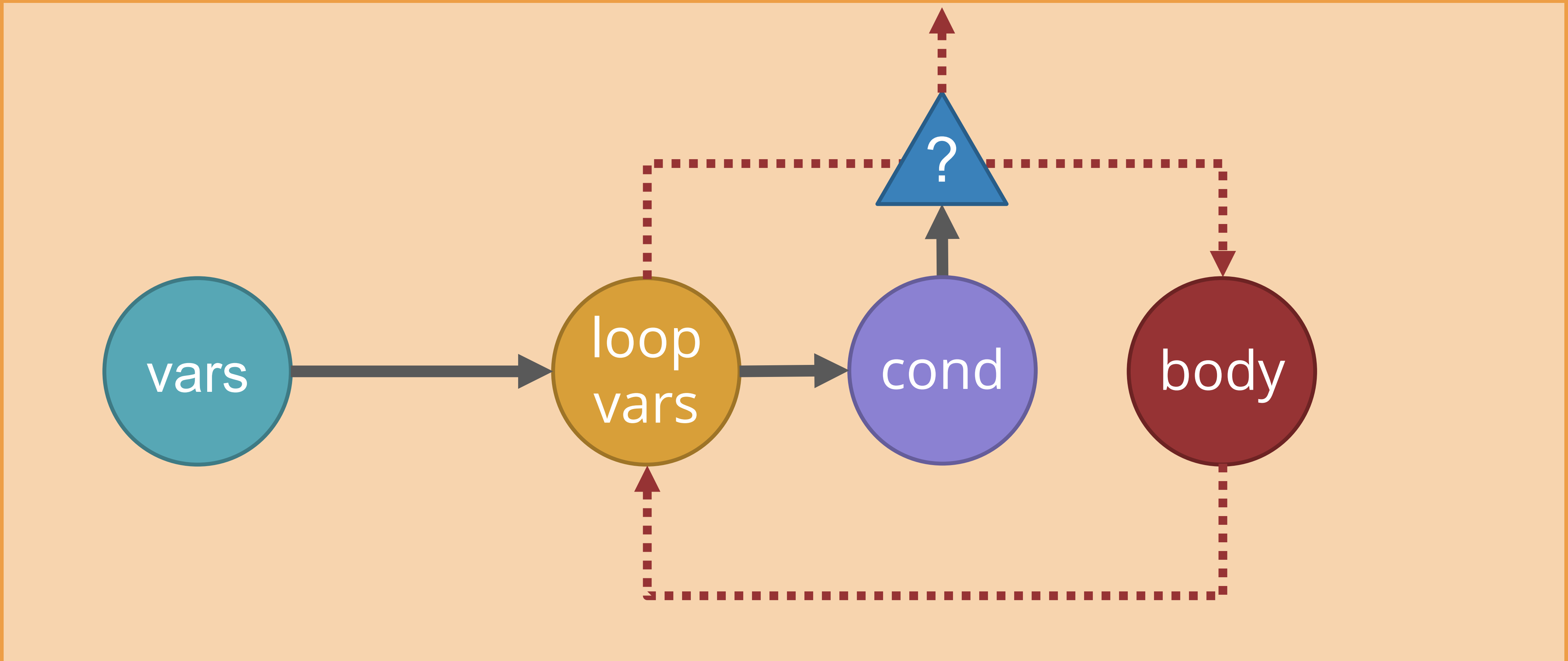
- Callable. Maps from `(*loop_vars)` → boolean scalar
- If it returns true, the body executes,
- Otherwise, we exit the loop

# `tf.while_loop`

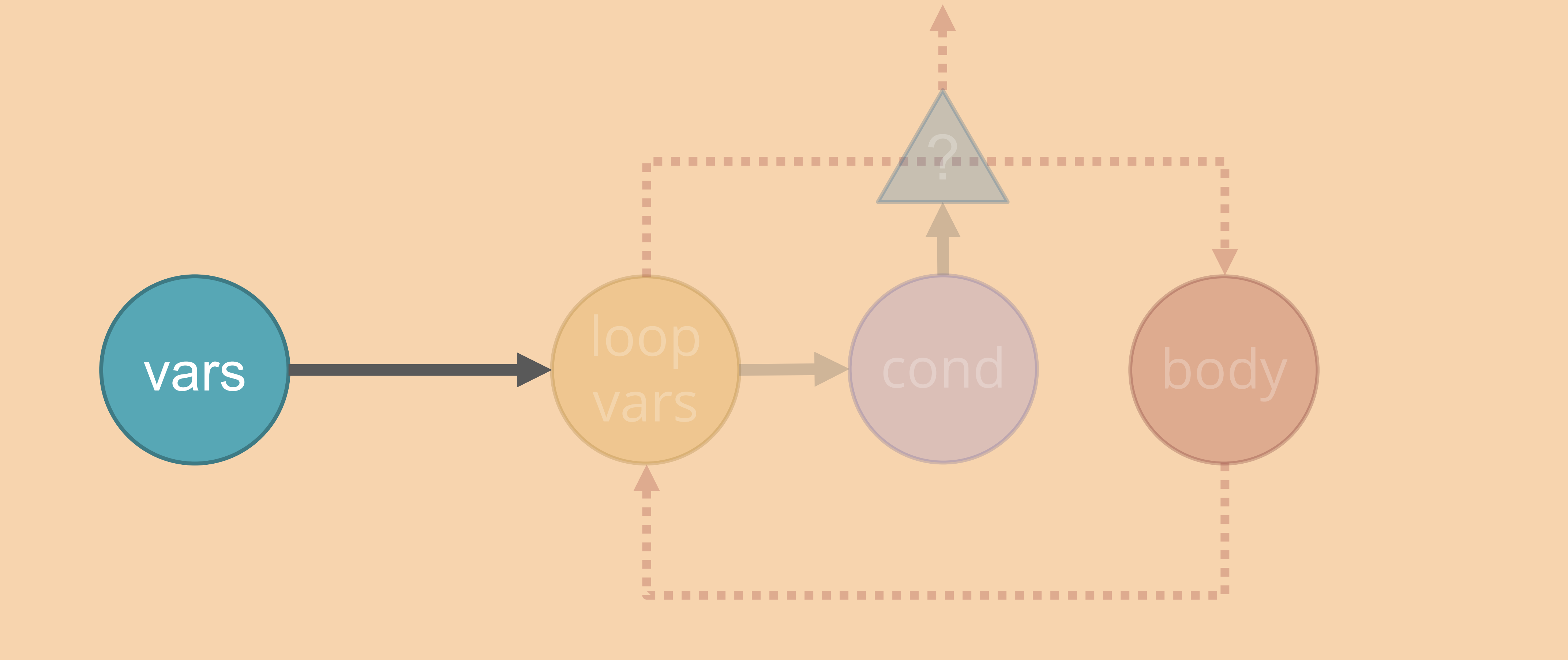
```
tf.while_loop(cond, body, loop_vars)
```

- Callable. Maps from `(*loop_vars)` → `(*next_loop_vars)`
- Main computation takes place here
- Also need to increment counter (if using one) here
- Output from this gets sent to next iteration `cond` and `body`

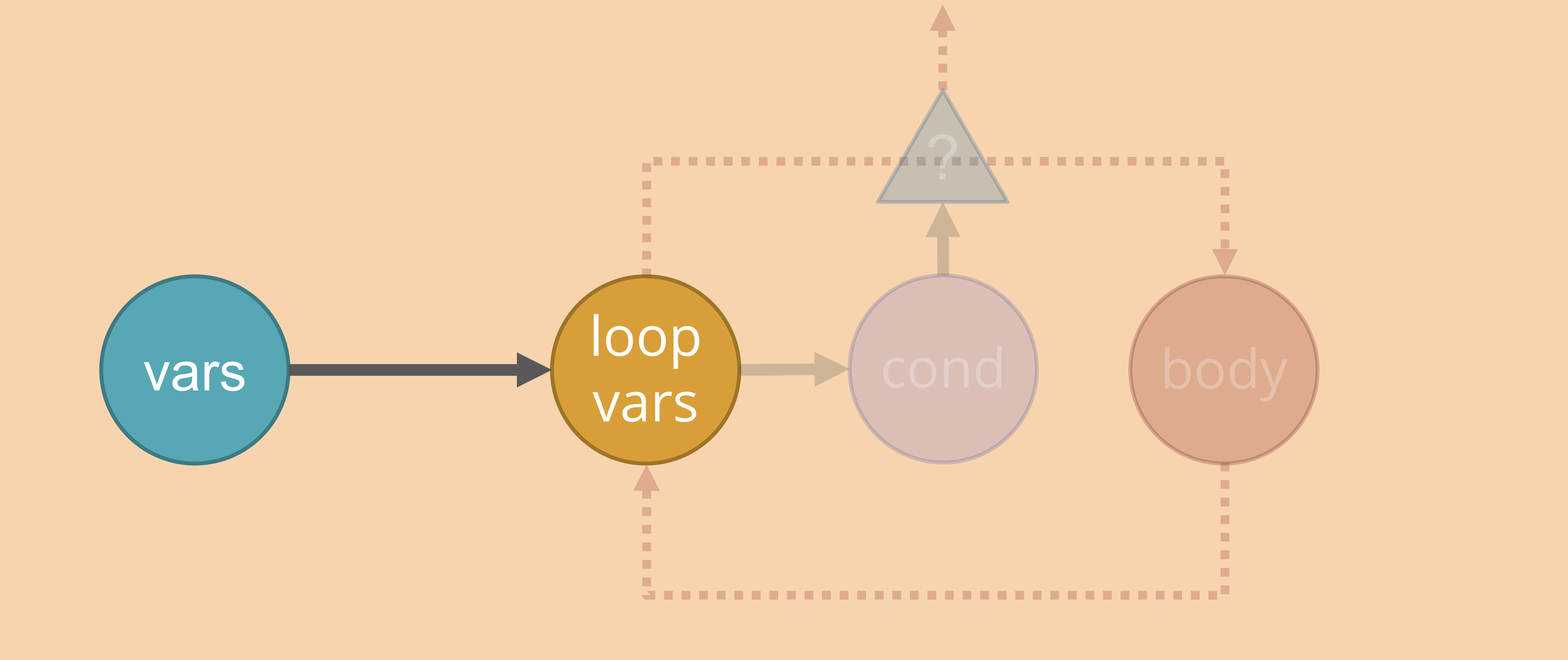
# Here is what the basic loop looks like



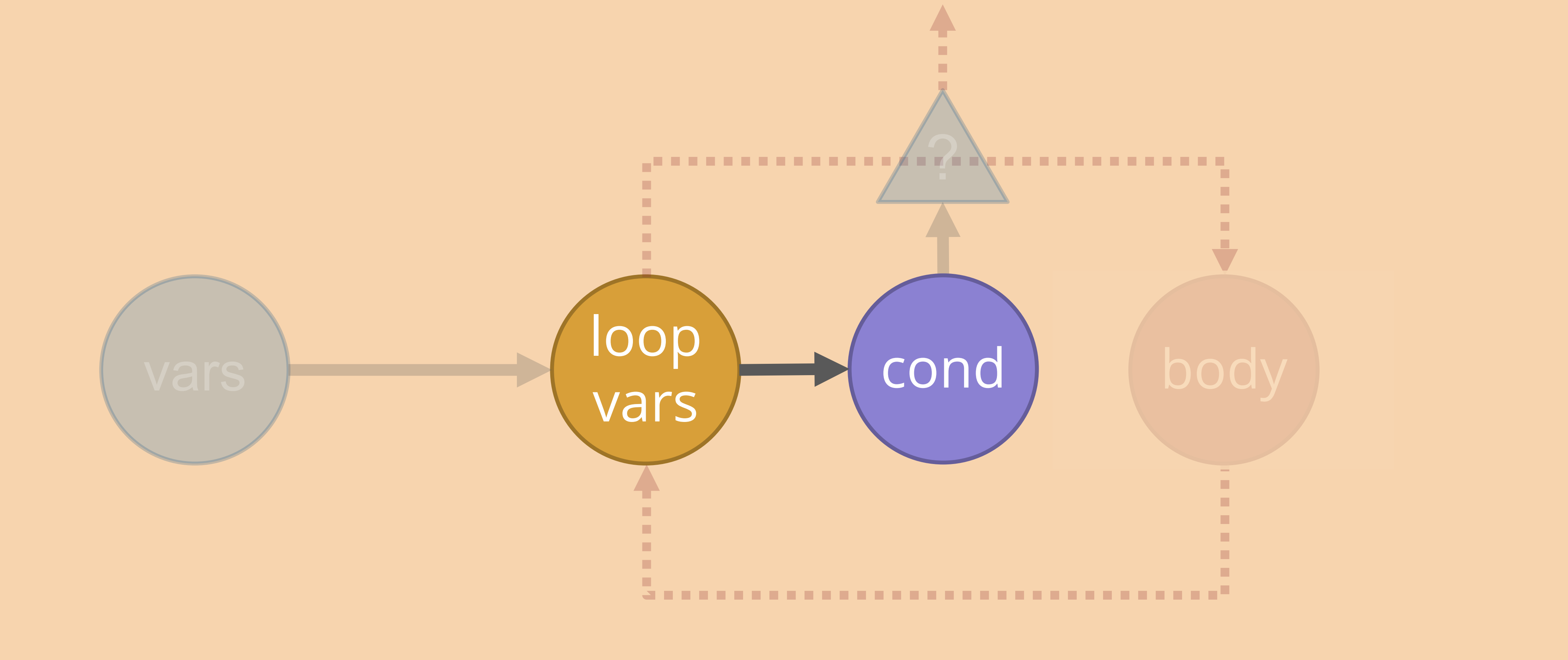
# We pass in our initial loop arguments



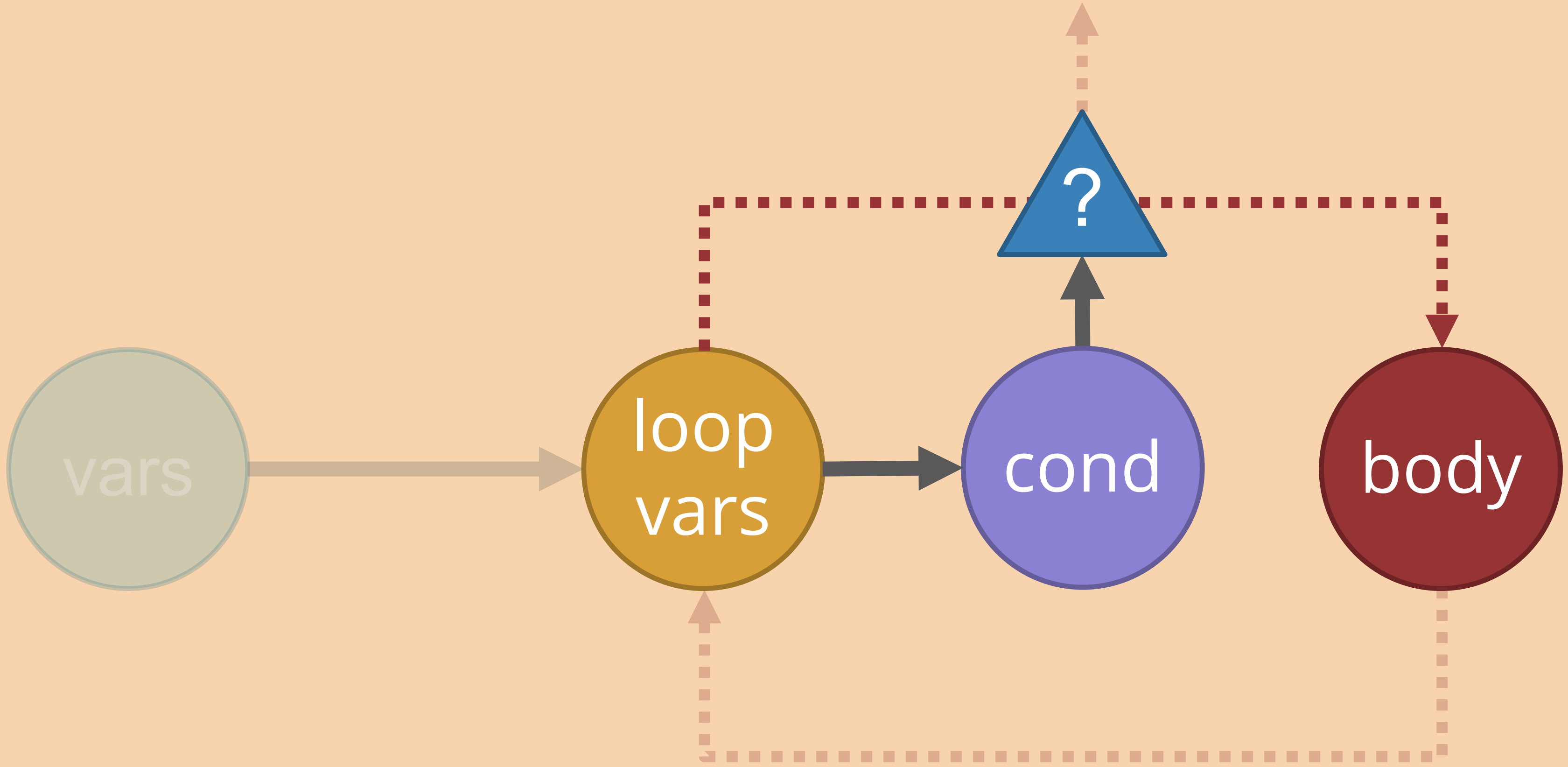
# Those are now the loop variables



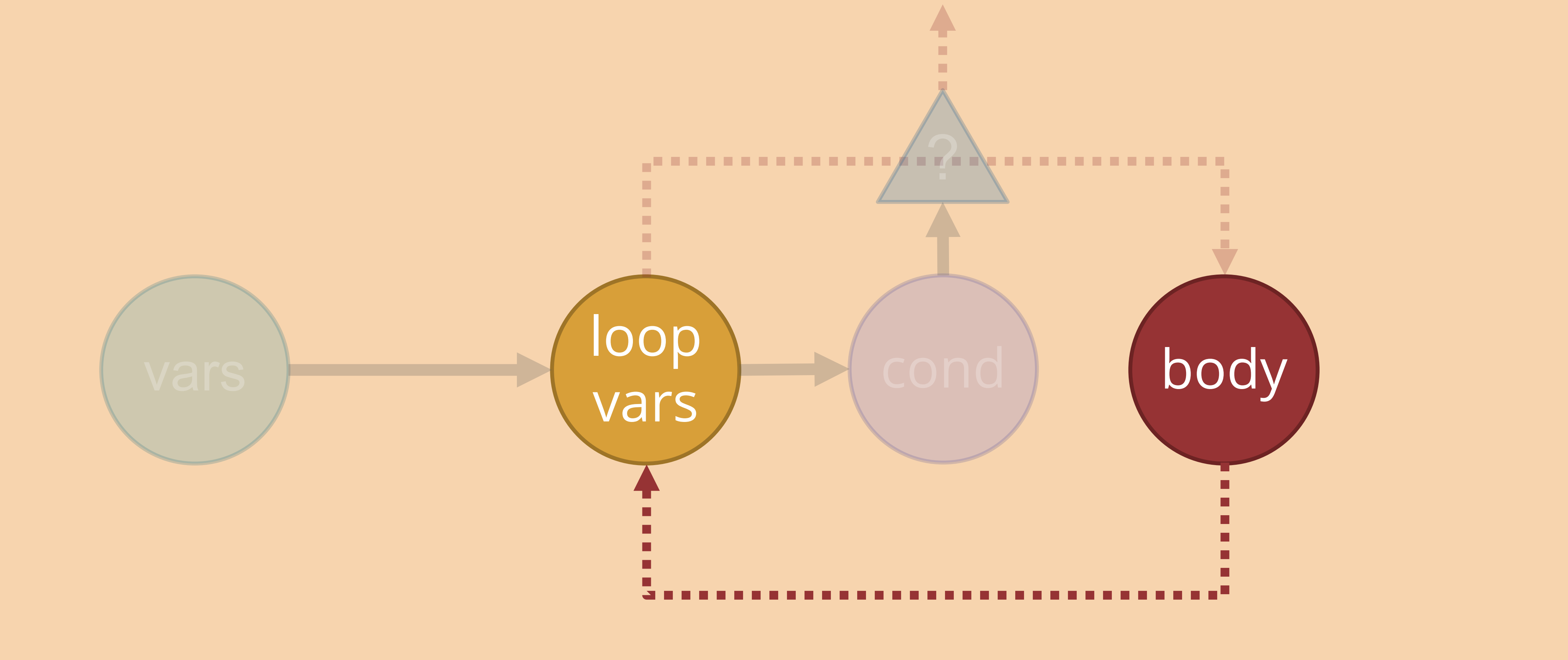
# The loop variables get sent to the cond function



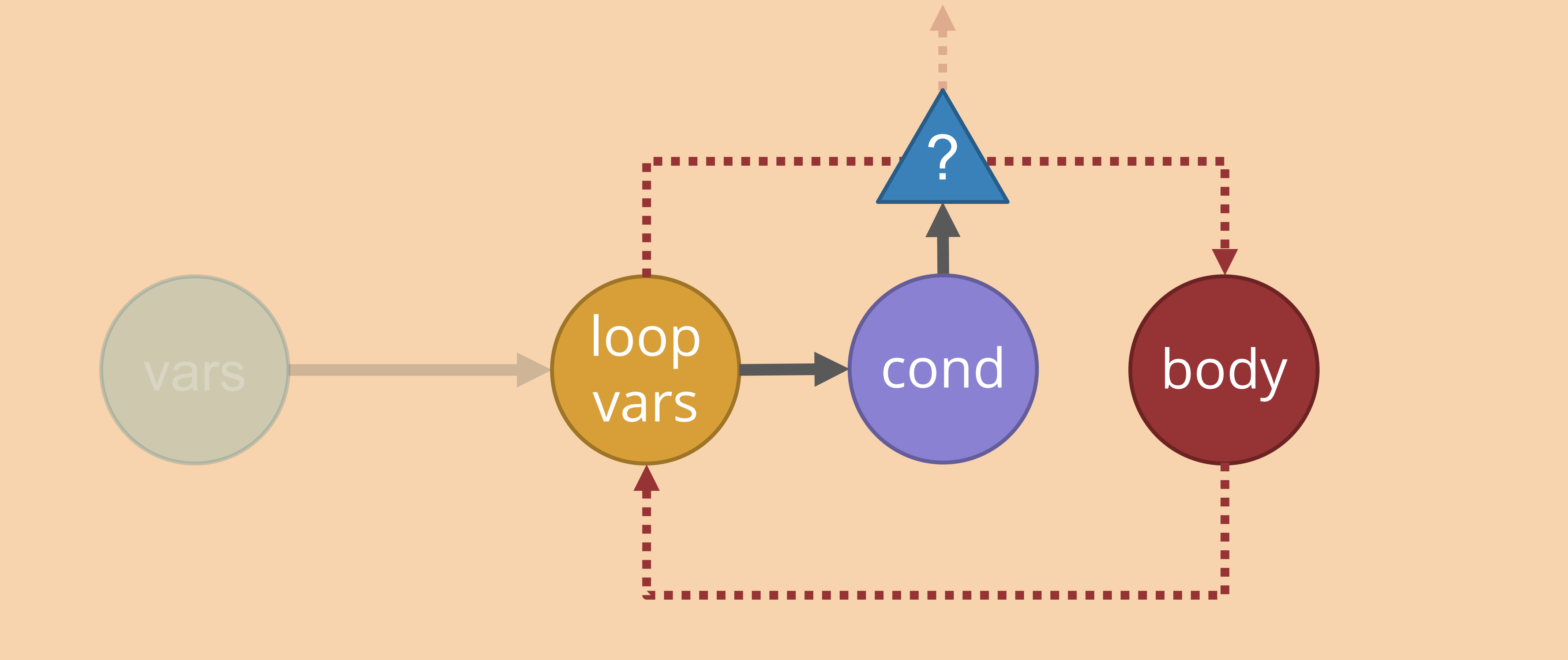
# If cond is true, we pass the vars to body



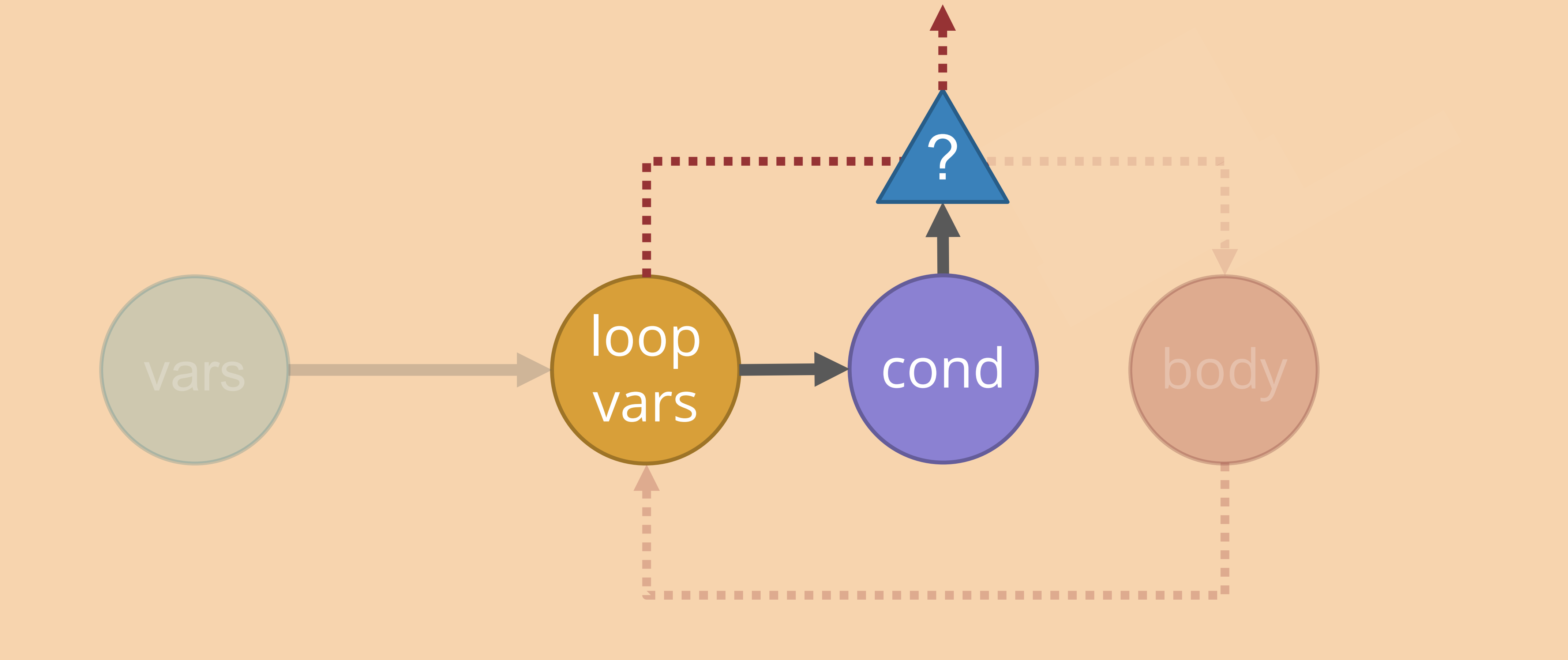
# body's outputs become the new loop vars



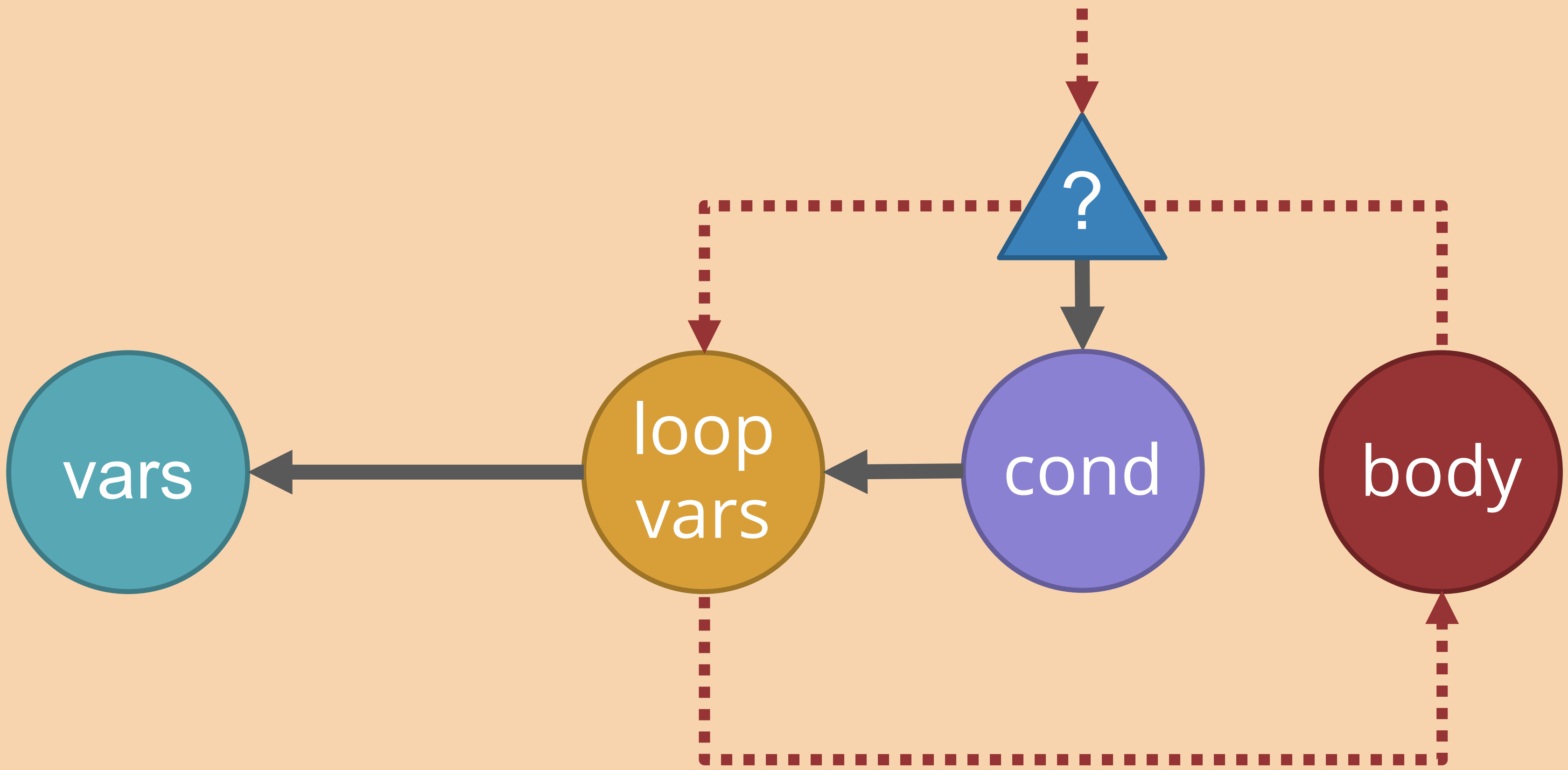
# The loop continues while cond evaluates to true



# Once cond is false, we return the current loop vars



# `while_loop` compatible with auto-differentiation



# Basic `while_loop` example: 100 loops

```
def cond(i, val):
    return i < 100
def body(i, val):
    return i+1, val + 5
loop = tf.while_loop(cond, body, (0, 0))
```

# Define our condition

```
def cond(i, val):
    return i < 100
def body(i, val):
    return i+1, val + 5
loop = tf.while_loop(cond, body, (0, 0))
```

# Define the body

```
def cond(i, val):
    return i < 100
def body(i, val):
    return i+1, val + 5
loop = tf.while_loop(cond, body, (0, 0))
```

# Build the loop!

```
def cond(i, val):
    return i < 100
def body(i, val):
    return i+1, val + 5
loop = tf.while_loop(cond, body, (0, 0))
```

# Notice that cond and body have same inputs

```
def cond(i, val):
    return i < 100
def body(i, val):
    return i+1, val + 5
loop = tf.while_loop(cond, body, (0, 0))
```

# And that the values are modified in the body

```
def cond(i, val):
    return i < 100
def body(i, val):
    return i+1, val + 5
loop = tf.while_loop(cond, body, (0, 0))
```

# Reusing variables is simple

```
def body(i, val):
    w = tf.get_variable('w', …)
    return i+1, tf.matmul(val, w)
```

## Don't have to declare `scope.reuse_variables()`

```
def body(i, val):
    w = tf.get_variable('w', …)
    return i+1, tf.matmul(val, w)
```

Reusing variables + feeding data into itself → RNN!

`tf.dynamic_rnn` is implemented with a `tf.while_loop`

Full implementation beyond scope of lecture

-But small RNN example is in notebook

# Optional parameters

- `shape_invariants` (default: `None`)
  - Allows you to specify which `loop_vars` can have variable shape
- `parallel_iterations` (default: `10`)
  - Number of allowed parallel iterations (if possible)
- `swap_memory` (default: `False`)
  - Allows (or disallows) GPU-CPU memory swap (RNN backprop is hungry)
- `back_prop` (default: `True`)
  - Allows (or disallows) backpropagation for this loop.

# `tf.while_loop` notes

- Faster than refeeding with loops of `sess.run()`
  - Roughly 30% improvement
- *Much* faster than unrolling with many static ops
  - Both in graph creation and in run time
- Like with conditionals, TensorBoard graph can get ugly
  - Use name/variable scope for `cond` and `body`

# WRAPPING THIS BABY UP

# Today we covered TF's main control flow ops

- Dependency management
  - `tf.control_dependencies`, `tf.group`, `tf.tuple`
- Conditional statements
  - `tf.cond`, `tf.case`
- Loops
  - `tf.while_loop`

# With native control flow:

Data transfer **overhead is minimized**

## With native control flow:

Graph logic is **self-contained**

## With native control flow:

Enables use of **differentiation and queues**

# THANKS!

GitHub: @samjabrahams
Twitter: @sabraha
Email: sam@samabrahams.com

Presentation template by [SlidesCarnival]