Static type checker for Python

Andrzej Wasylkowski
supervised by Paweł Rychlikowski
University of Wrocław

Introducing Python

- Invented in 1989 by Guido van Rossum
- Object-oriented, highly introspective
- Several implementations, lots of libraries
- Used by Google, NASA, ILM
- Easy to extend and to embed
- Dynamically typed, no static type checker

Type checking

```
def fun ():
    if random () <= 0.99:
        return 21
    return “x”

print 3.14 * fun ()
```

Type checking

```
def fun ():
    print 3.14 * fun ()
```

Dynamic type checking at run-time

```
Traceback (most recent call last):
  File "example.py", line 6, in ?
    print 3.14 * fun ()
TypeError: can't multiply sequence to non-int
```

Type checking

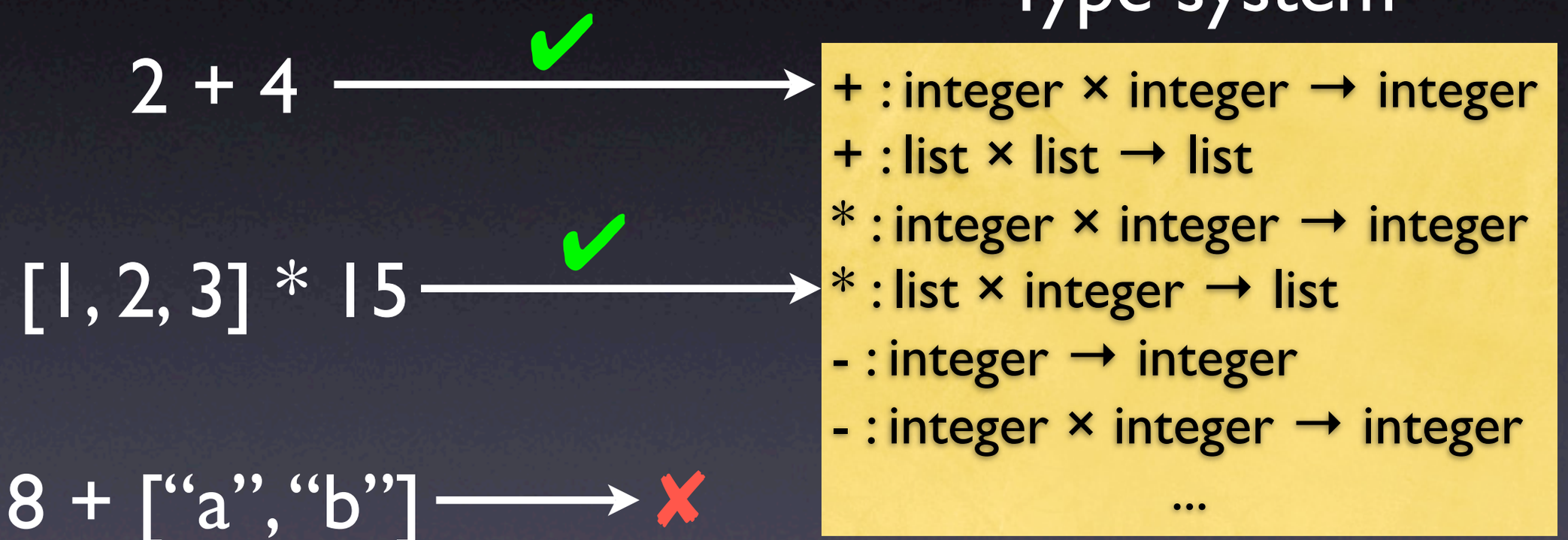
```
def fun ():
    print 3.14 * fun ()
```

Static type checking at compile-time

```
:::warning:::
   example.py:6
   possibly unsupported operand type(s) for *:
   'float' and '(int, str)'
```

Goal: find type errors as soon as possible

Type errors



Type error is a misuse of a value

Python's intricacies (1)

Java code

```
int y;
y = 10;
y = “abc”;
```

✘

Python code

```
y = 10;
y = “abc”
```

✔

In Python only values have types;
variables serve as placeholders

Python's intricacies (2)

```
result = a + b
```

```
result = a.__class__.__add__(a,b)
if result == NotImplemented:
   result = b.__class__.__radd__(b,a)
   if result == NotImplemented:
      error (…)
```

Python's intricacies (2)

```
for x in xs:
   …
```



```
x_iter = xs.__class__.__iter__ (xs)
try:
   while True:
      x = x_iter.next ()
      …
except StopIteration:
   pass
```

All operations are implemented as method calls

Python's intricacies (3)

```
class A:
    pass

x = A()
...
print x + x
```

Will this work?

It depends

Python's intricacies (3)

```
class A:
   pass

x = A()
pass
print x + x
```

✘

```
class A:
   pass

x = A()
def f (a,b):
   return a
A.__add__ = f
print x + x
```

✔

Class' interface may change at run-time

Introducing PTC

```
x = raw_input (“input:”)                    x:str
if len (x) > 10:
   y = 15
else:
   y = “foo”
   y += raw_input (“input: “)

z = y
```

Introducing PTC

```
x = raw_input (“input:”)                    x:str
if len (x) > 10:                            x:str
   y = 15                                x:str,y:int
else:
   y = “foo”
   y += raw_input (“input: “)

z = y
```

Introducing PTC

```
x = raw_input (“input:”)                    x:str
if len (x) > 10:                            x:str
   y = 15                                x:str,y:int
else:                                       x:str
   y = “foo”
   y += raw_input (“input: “)

z = y
```

Introducing PTC

```
x = raw_input (“input:”)                    x:str
if len (x) > 10:                            x:str
   y = 15                                x:str,y:int
else:                                       x:str
   y = “foo”                             x:str,y:str
   y += raw_input (“input: “)

z = y
```

Introducing PTC

```
x = raw_input (“input:”)                    x:str
if len (x) > 10:                            x:str
   y = 15                                x:str,y:int
else:                                       x:str
   y = “foo”                             x:str,y:str
   y += raw_input (“input: “)            x:str,y:str

z = y
```

Introducing PTC

```
x = raw_input (“input:”)                    x:str
if len (x) > 10:                            x:str
   y = 15                                x:str,y:int
else:                                       x:str
   y = “foo”                                x:str,y:str
   y += raw_input (“input: “)               x:str,y:str
                                  x:str,y:(int,str)
z = y
```

Introducing PTC

```
x = raw_input (“input:”)                    x:str
if len (x) > 10:                            x:str
   y = 15                                x:str,y:int
else:                                       x:str
   y = “foo”                             x:str,y:str
   y += raw_input (“input: “)            x:str,y:str
                                  x:str,y:(int,str)
z = y                    x:str,y:(int,str),z:(int,str)
```

Introducing PTC

```
x = raw_input (“input:”)                    x:str
if len (x) > 10:                            x:str
   y = 15                                x:str,y:int
else:                                       x:str
   y = “foo”                             x:str,y:str
   y += raw_input (“input: “)            x:str,y:str
                                  x:str,y:(int,str)
z = y                    x:str,y:(int,str),z:(int,str)
```

PTC analyzes the source code sequentially,
in a flow-sensitive manner

Loops

```
result = 1
while n > 0:
    result *= n
    n -= 1
```

PTC →

```
result = 1
if n > 0:
    result *= n
    n -= 1
    if n > 0:
        result *= n
        n -= 1
        …
```

PTC analyzes loops by unrolling
them user-specified number of times

Function calls

```
def add (a, b):
    sum = a + b
    return sum
```

```
x = add (3, 5)
y = add ([2], [“5”])
```

Function calls

```
def add (a, b):                    a:int,b:int
    sum = a + b
    return sum
```

```
x = add (3, 5)
y = add ([2], [“5”])
```

Function calls

```
def add (a, b):
    sum = a + b
    return sum
```

```
a:int,b:int
sum:(int,long)
```

```
x = add (3, 5)
y = add ([2], [“5”])
```

Function calls

```
def add (a, b):
    sum = a + b
    return sum
```

```
a:int,b:int
sum:(int,long)
```

```
x = add (3, 5)
y = add ([2], [“5”])
```

Function calls

```
def add (a, b):
    sum = a + b
    return sum
```

```
x = add (3, 5)                    x:(int,long)
y = add ([2], [“5”])
```

Function calls

```
def add (a, b):
    sum = a + b
    return sum
```

a:list[int],b:list[str]

```
x = add (3, 5)
y = add ([2], [“5”])
```

x:(int,long)

Function calls

```
def add (a, b):                    a:list[int],b:list[str]
   sum = a + b                          sum:list[(int,str)]
   return sum
```

```
x = add (3, 5)                                  x:(int,long)
y = add ([2], [“5”])
```

Function calls

```
def add (a, b):                    a:list[int],b:list[str]
   sum = a + b                          sum:list[(int,str)]
   return sum
```

```
x = add (3, 5)                                  x:(int,long)
y = add ([2], [“5”])
```

Function calls

```
def add (a, b):
    sum = a + b
    return sum
```

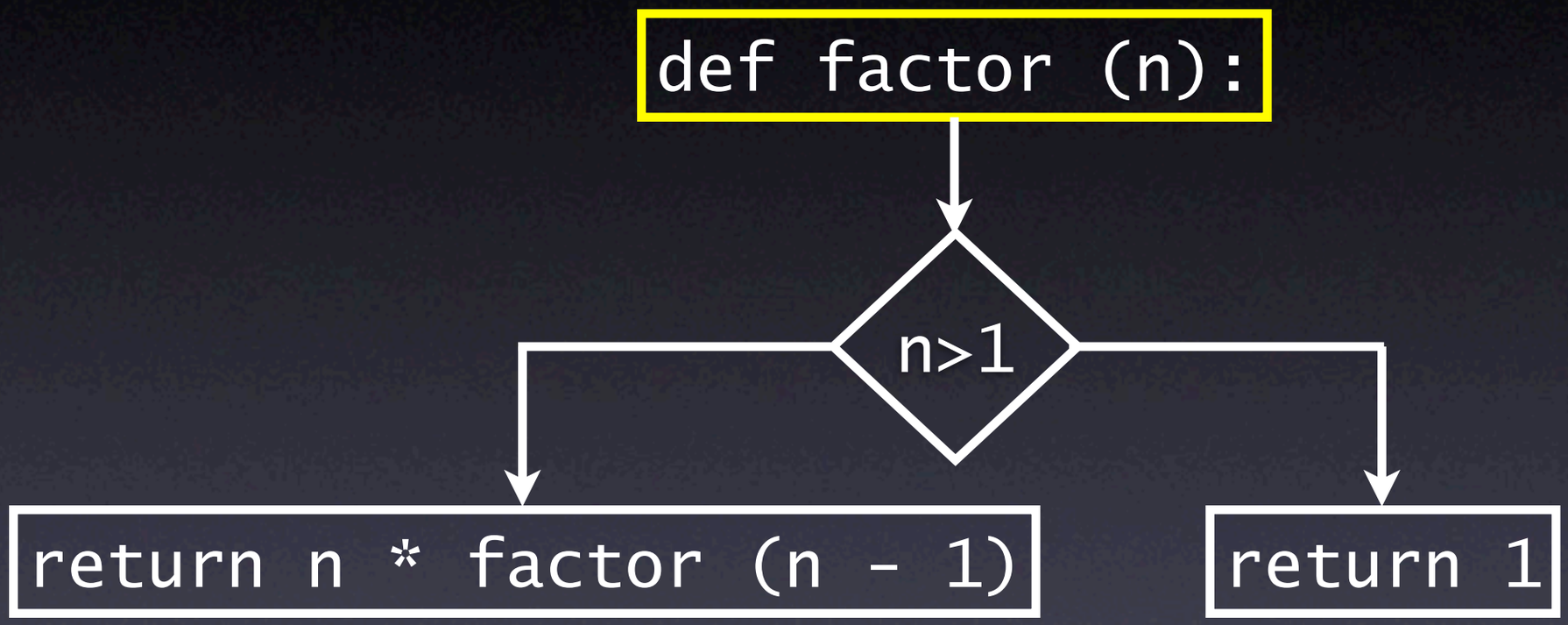
```
x = add (3, 5)                    x:(int,long)
y = add ([2], [“5”])          y:list[(int,str)]
```

PTC analyzes function calls by analyzing the callee’s body at each call site

Recursion

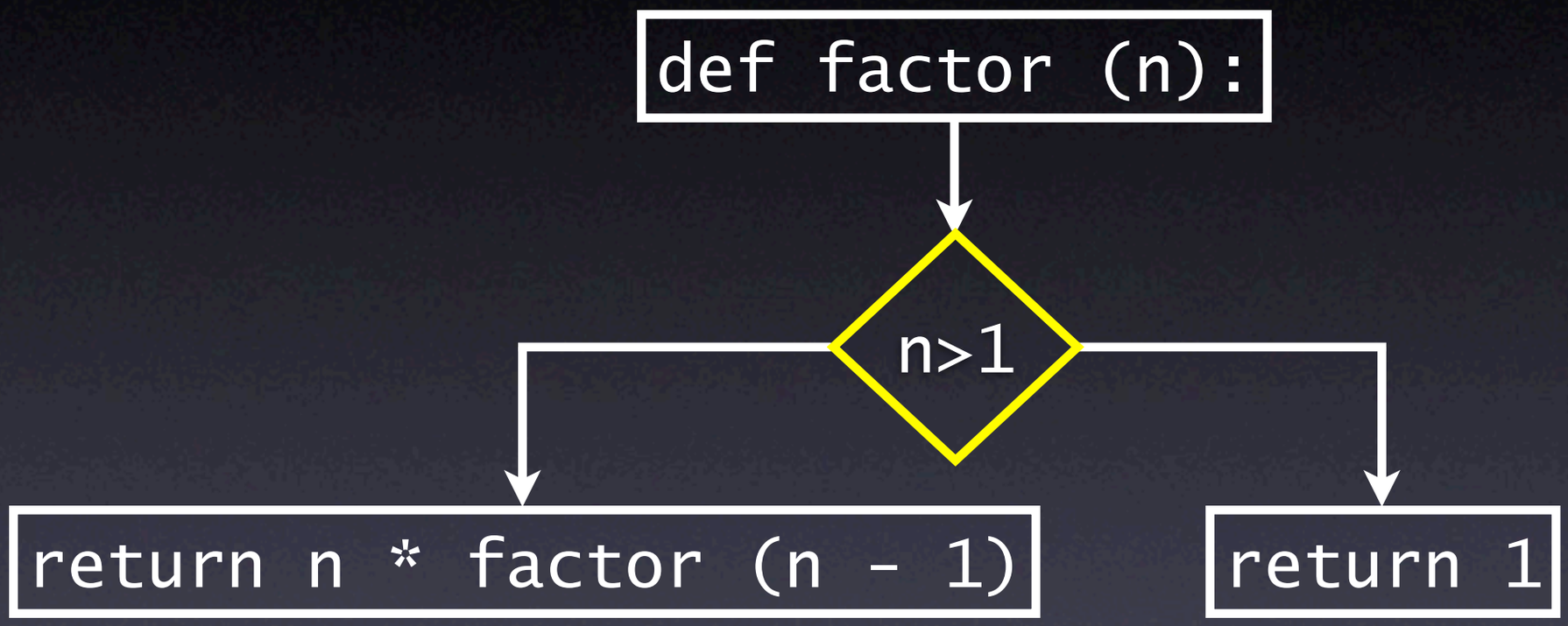
```
def factor (n):
    if n > 1:
        return n * factor (n-1)
    else:
        return 1

x = factor (12)
```

Recursion



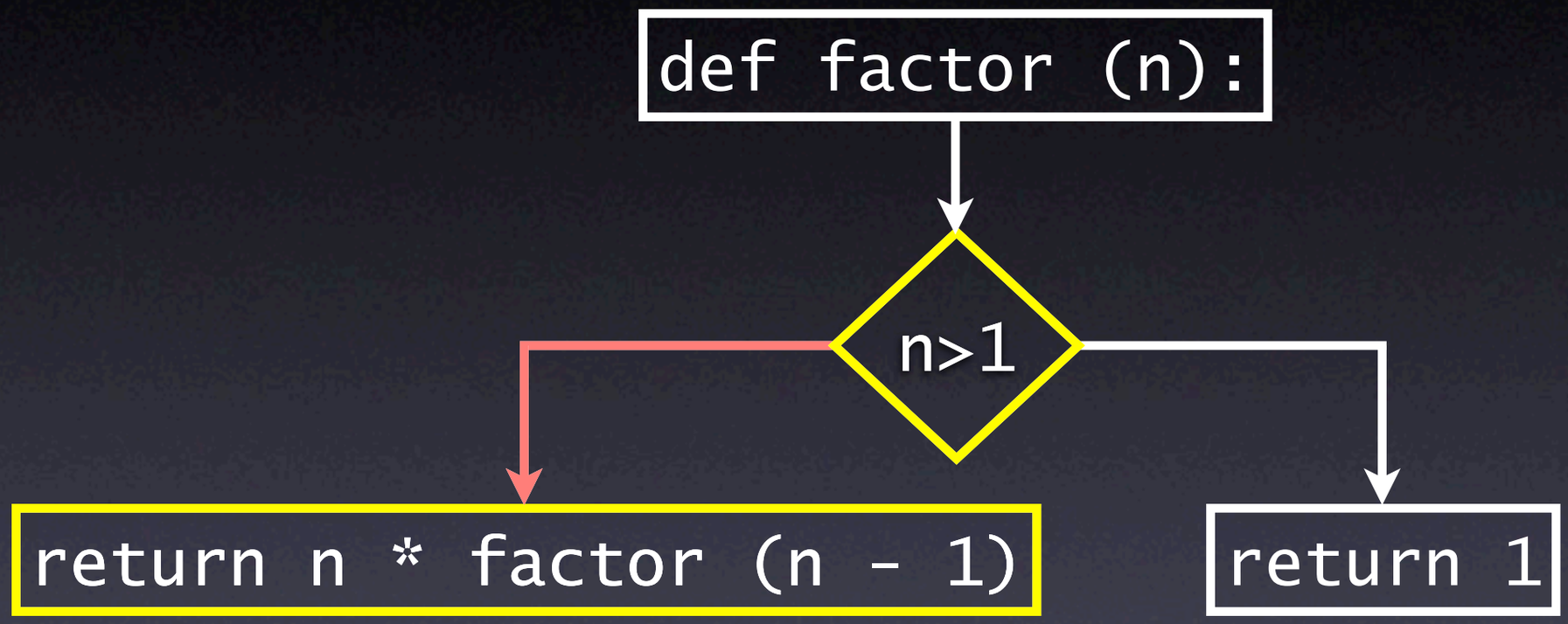
Return values set: {}

Recursion



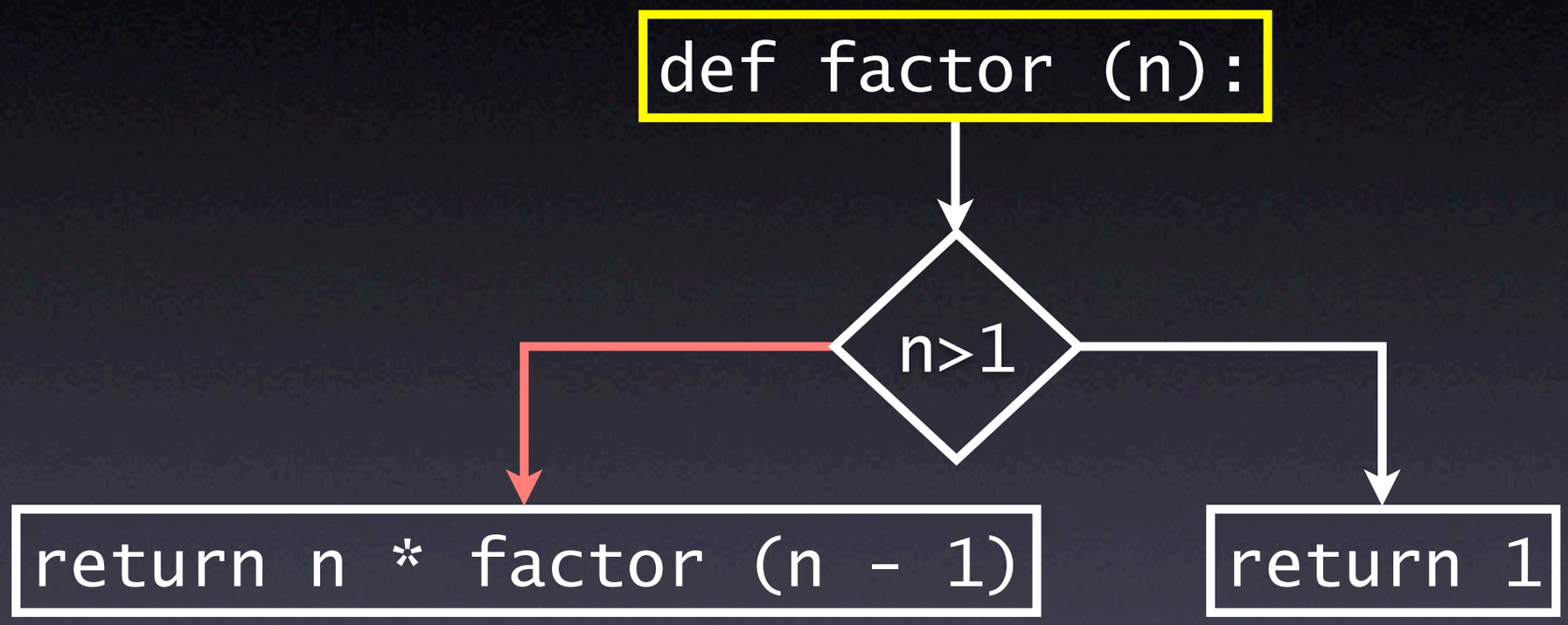
Return values set: {}

Recursion



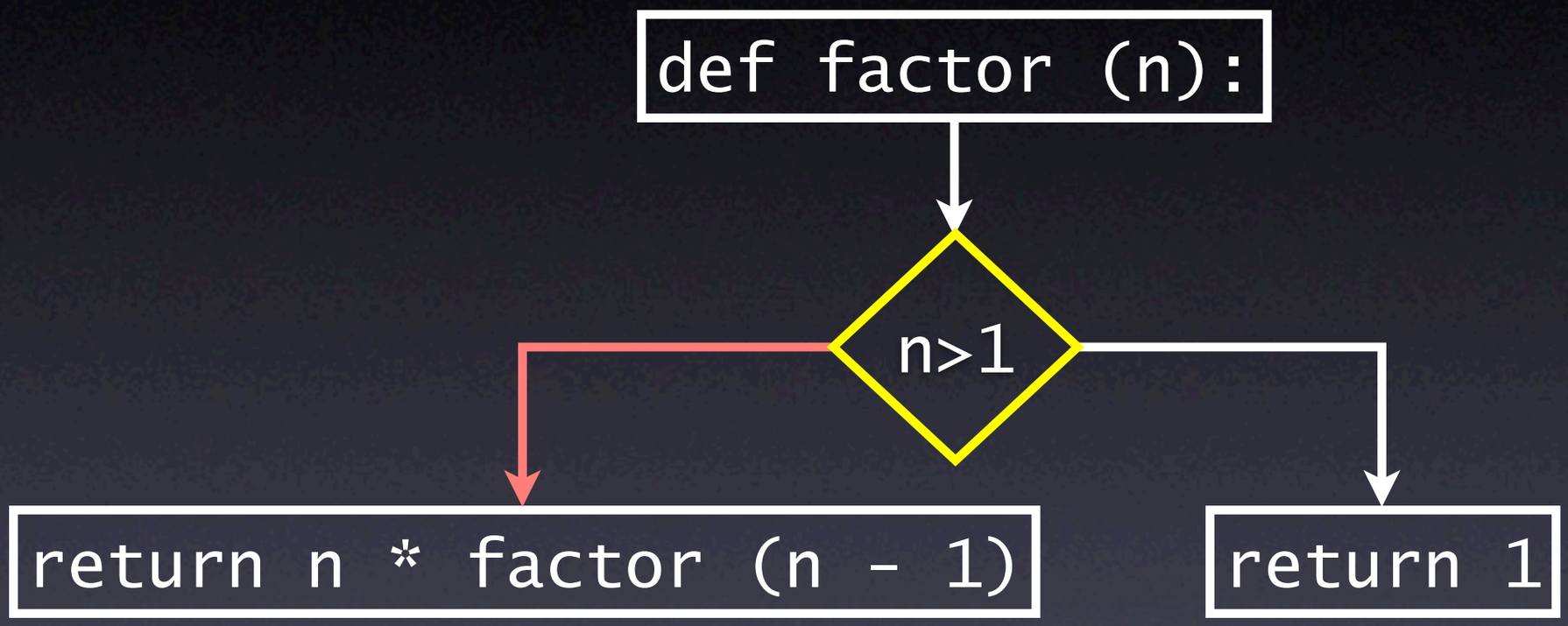
Return values set: {}

Recursion



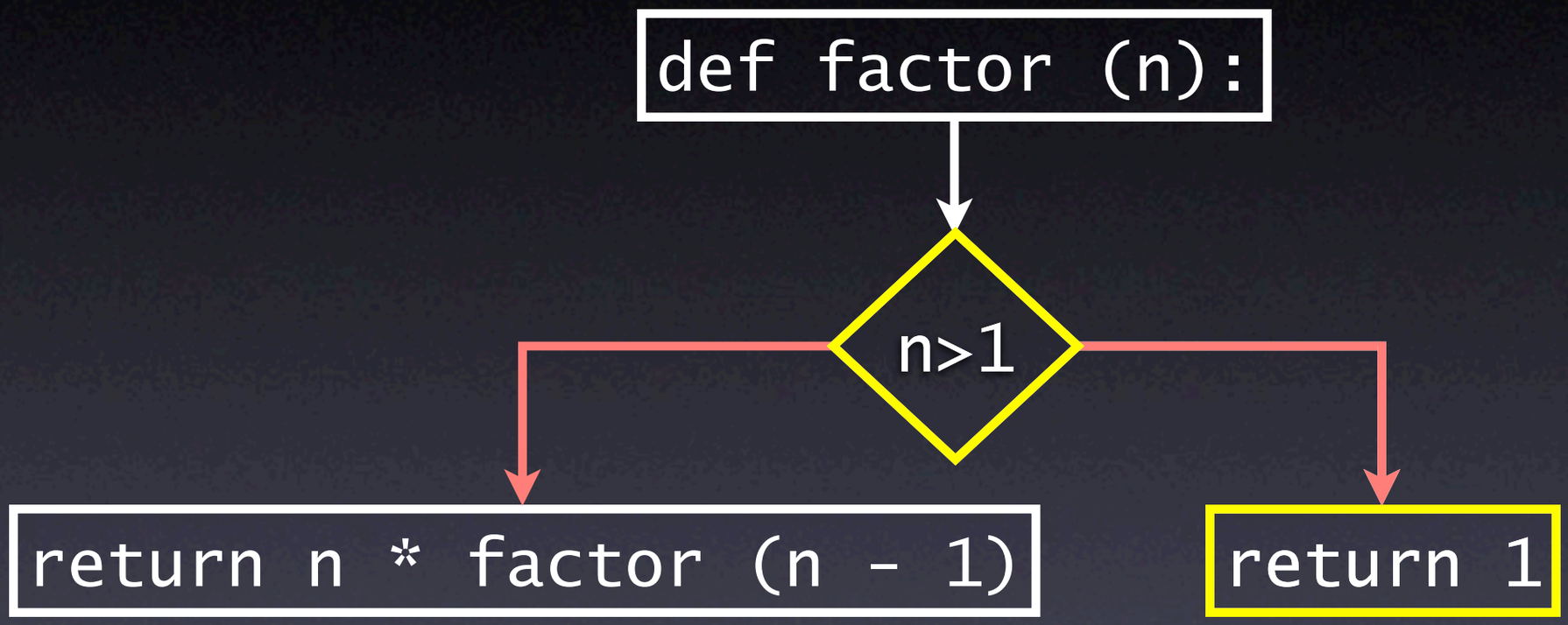
Return values set: {}

Recursion



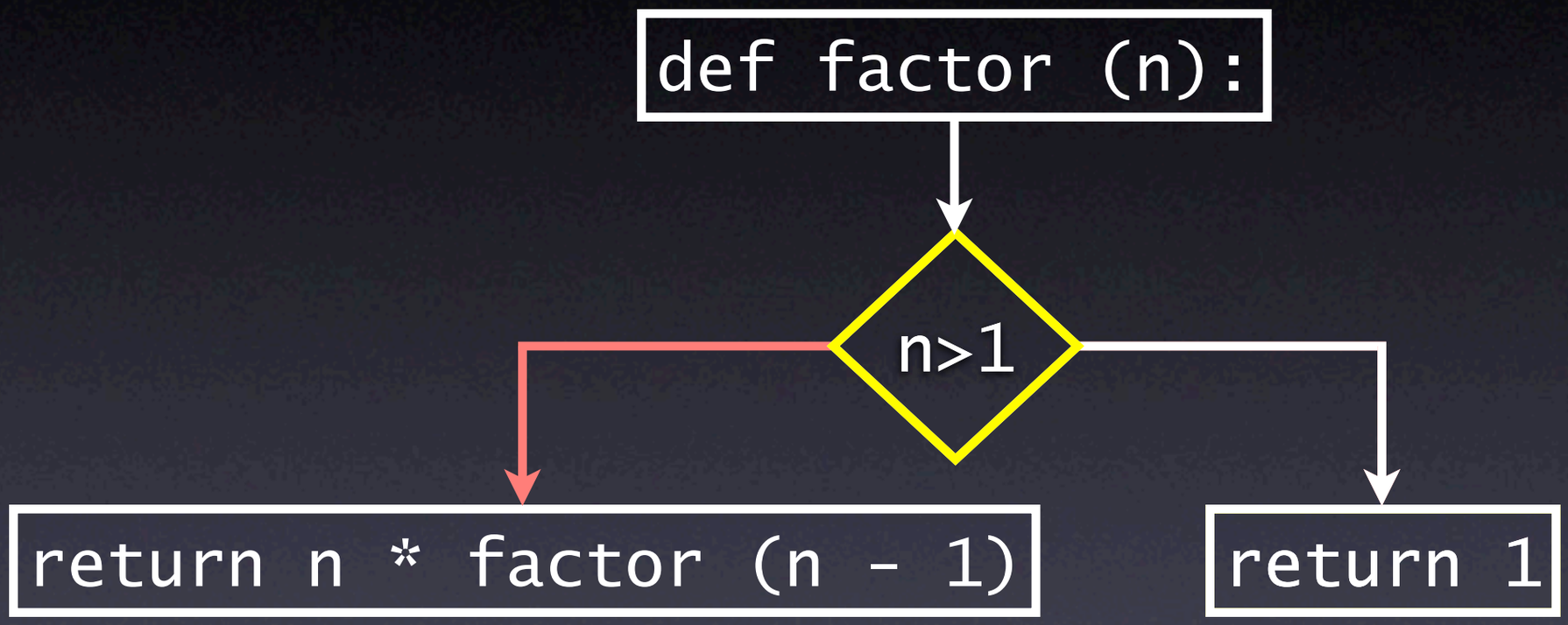
Return values set: {}

Recursion



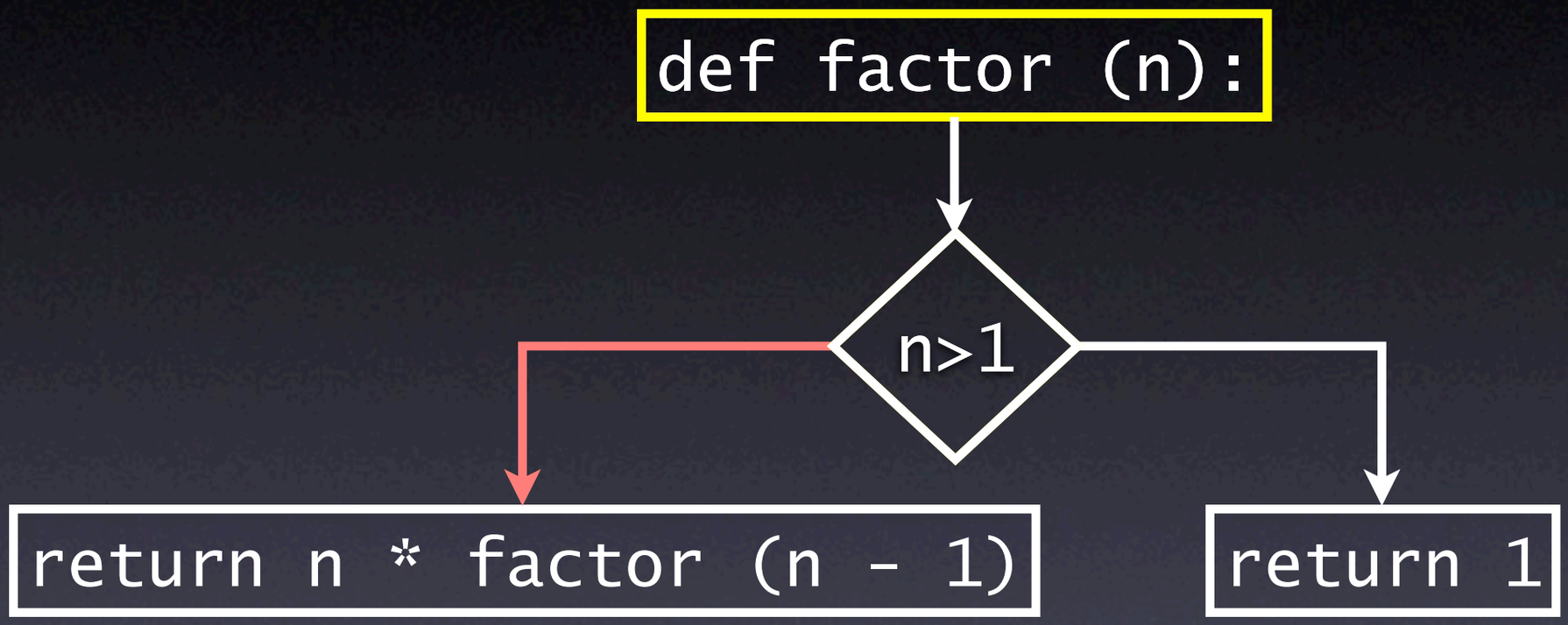
Return values set: `{int}`

Recursion



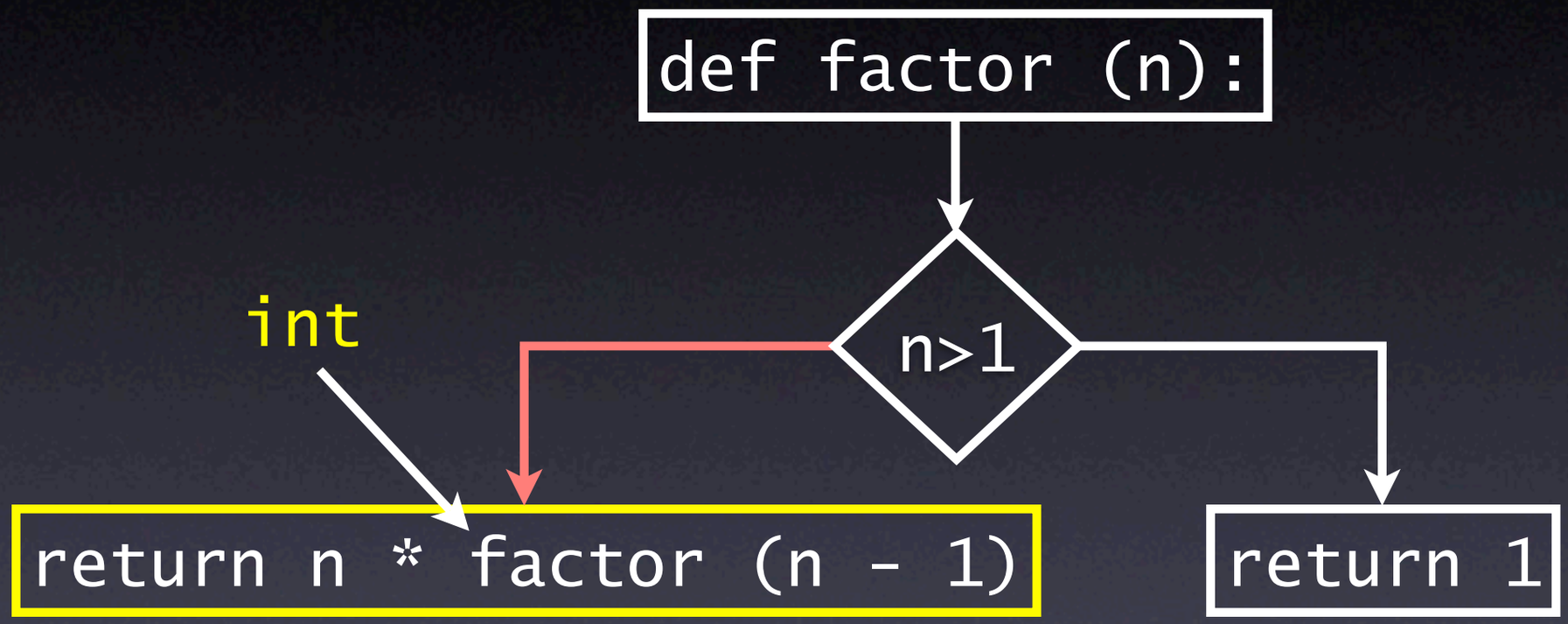
Return values set: `{int}`

Recursion



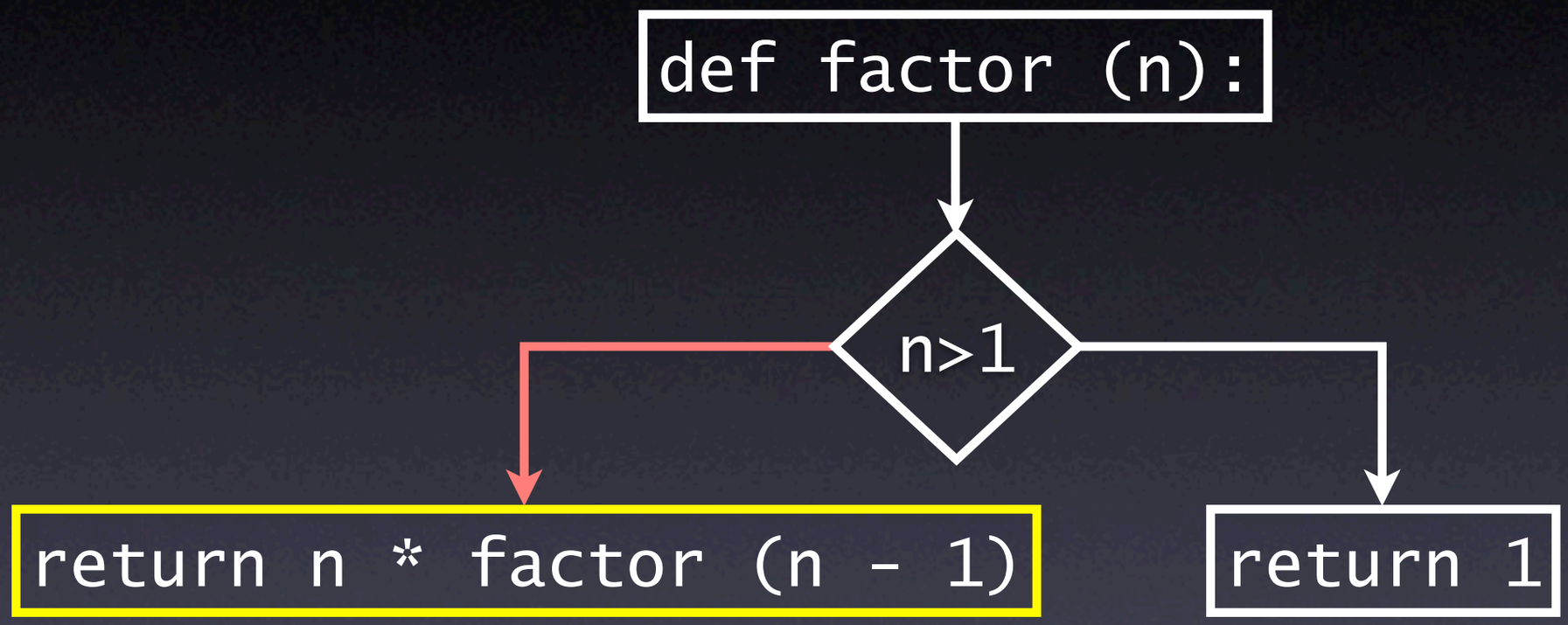
Return values set: `{int}`

Recursion



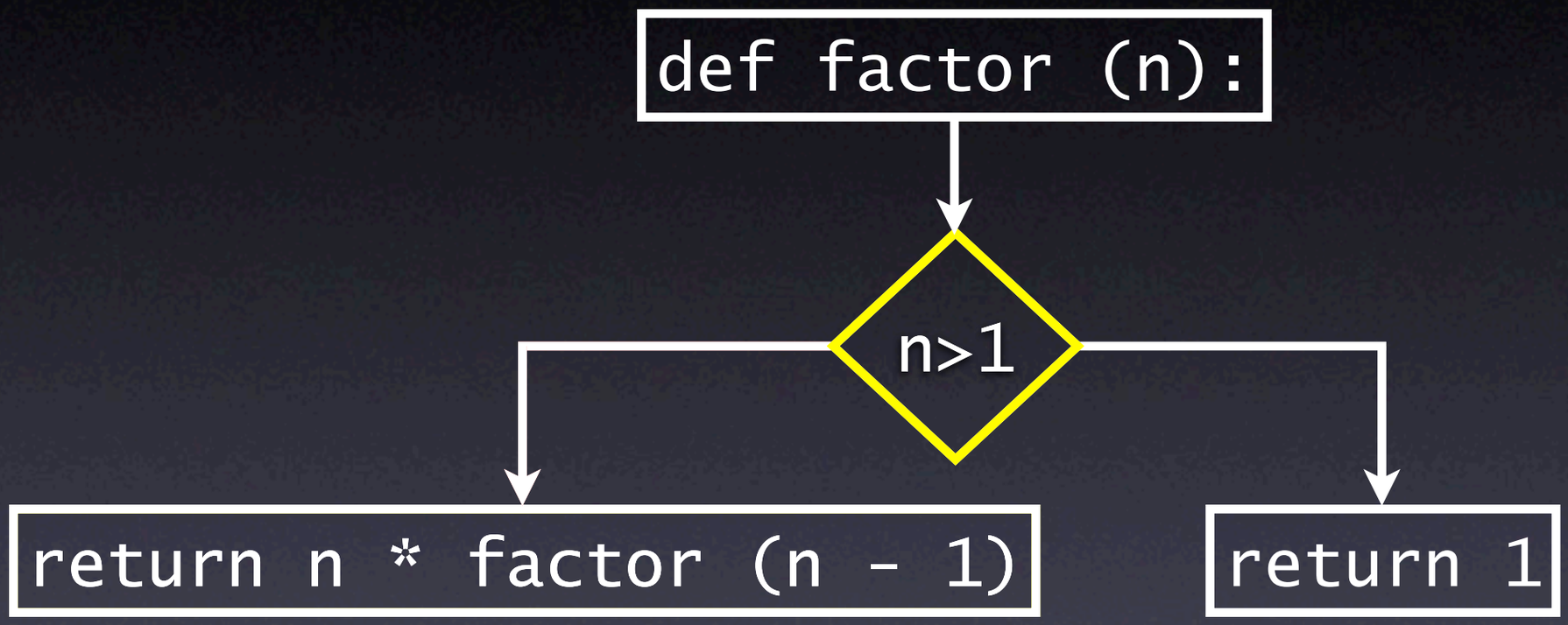
Return values set: {}

Recursion



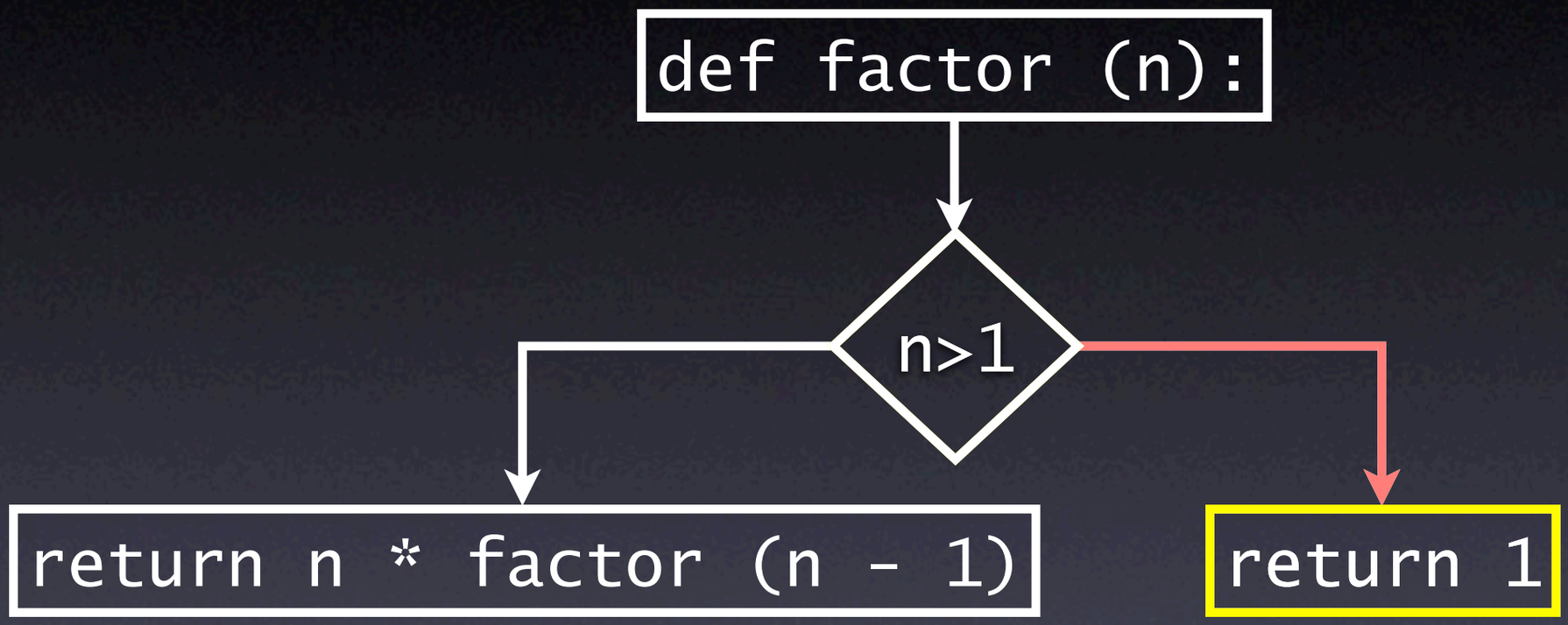
Return values set: `{int,long}`

Recursion



Return values set: `{int,long}`

Recursion



Return values set: `{int,long}`

PTC approximates recursive calls by taking each time a different path through a function

Inaccuracies

```
1  n = 5
2  x = 1
3  if n > 0:
4      x = [1]
5  y = x[0]
```

```
:::warning:::
   ex4.py:5
   possibly unsubscriptable object:
   ('int' object, 'list [int]' object)
```

False positives are a result of incomplete knowledge about a program being checked

Undecidability

```
…
x = 8
while n > 0:
   x = [x]
   n -= 1
```

```
x = 8

x = [8]
```

```
y = x
while m > 0:
   y = y[0]
   m -= 1
```

Undecidability

```
…
x = 8
while n > 0:
   x = [x]
   n -= 1
```

```
x = 8

x = [[8]]
```

```
y = x
while m > 0:
   y = y[0]
   m -= 1
```

Undecidability

```
…
x = 8
while n > 0:
   x = [x]
   n -= 1
```

```
x = 8

x = [[[8]]]
```

```
y = x
while m > 0:
   y = y[0]
   m -= 1
```

Undecidability

```
…
x = 8
while n > 0:
   x = [x]
   n -= 1
```

n-times nesting

```
y = x
while m > 0:
   y = y[0]
   m -= 1
```

Undecidability

```
…
x = 8
while n > 0:
   x = [x]
   n -= 1
y = x
while m > 0:
   y = y[0]
   m -= 1
```

n-times nesting

```
y = [[[8]]]
```

```
y = [[8]]
```

Undecidability

```
…
x = 8
while n > 0:
   x = [x]
   n -= 1
y = x
while m > 0:
   y = y[0]
   m -= 1
```

n-times nesting

```
y = [[[8]]]
```

```
y = [8]
```

Undecidability

```
…
x = 8
while n > 0:
   x = [x]
   n -= 1
```

n-times nesting

```
y = x
while m > 0:
   y = y[0]
   m -= 1
```

m-times unnesting

Undecidability

```
…
x = 8
while n > 0:
    x = [x]
    n -= 1
y = x
while m > 0:
    y = y[0]
    m -= 1
```

This program is type correct only if $m \leq n$

Checking type correctness of Python programs is undecidable

Related work

- S. Kaes (LFP 1992)
- C. Lindig (MSc thesis, 1993)
- A. K. Wright, R. Cartwright (LFP 1994)
- M. Widera (WFLP 2001)
- M. Salib (EP 2004)

Results

- Tested on Python tutorial
- Tested on programs with injected bugs
- Type-checked a few Python libraries
- High accuracy when no “type juggling”

Conclusions

- PTC is a static type checker for Python
- Deals with flow-sensitivity + union types
- Tested on Python tutorial + injected bugs
- Much room for research, e.g.
  behavior instead of types (Haskell)