C++ Semantics

And the meaning of things

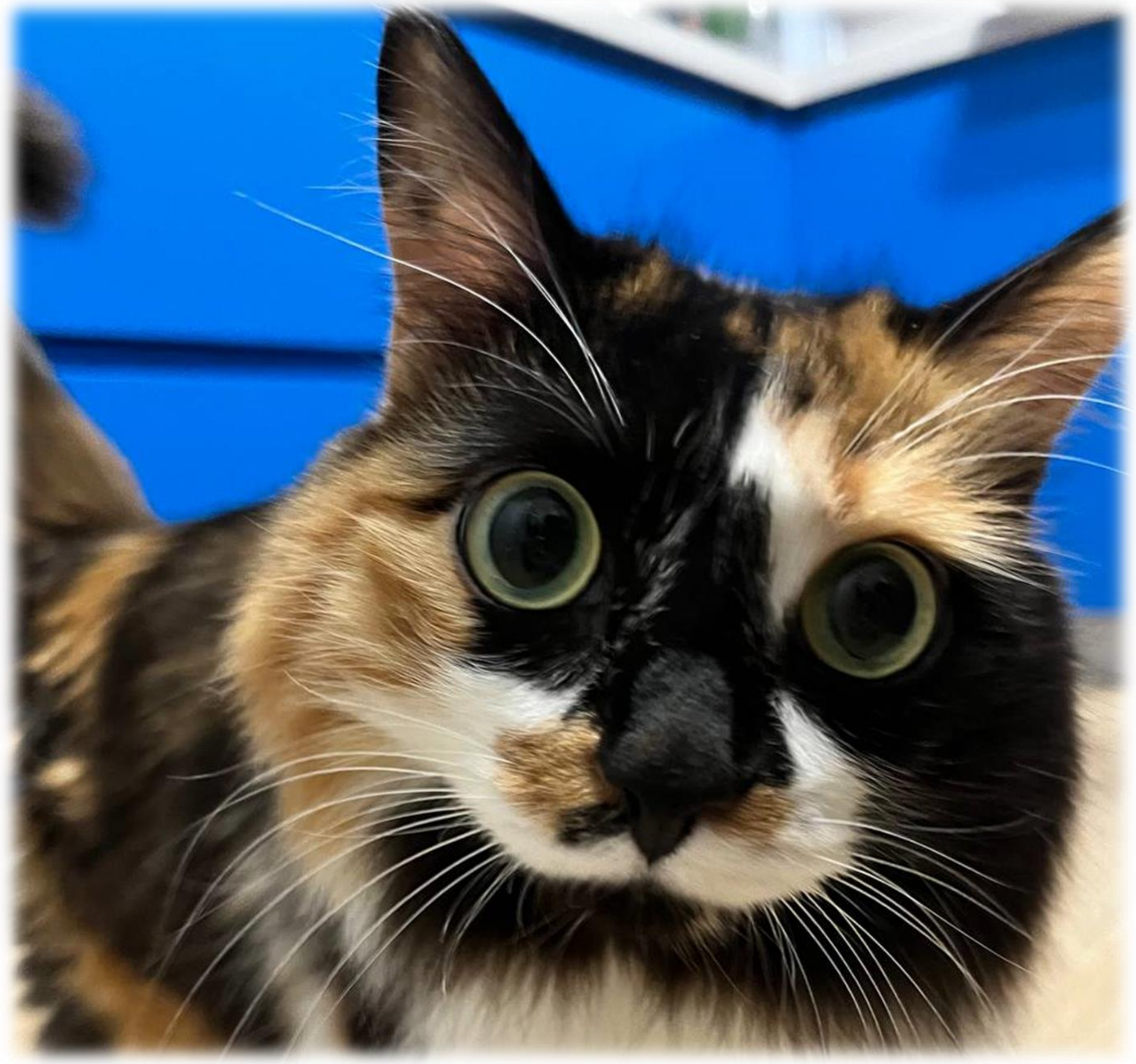
PVS-Studio

Yuri Minaev

Architect

Yuri Minaev

Architect at PVS-Studio



Syntax vs Semantics

Syntax

the arrangement of words and phrases to create well-formed sentences in a language

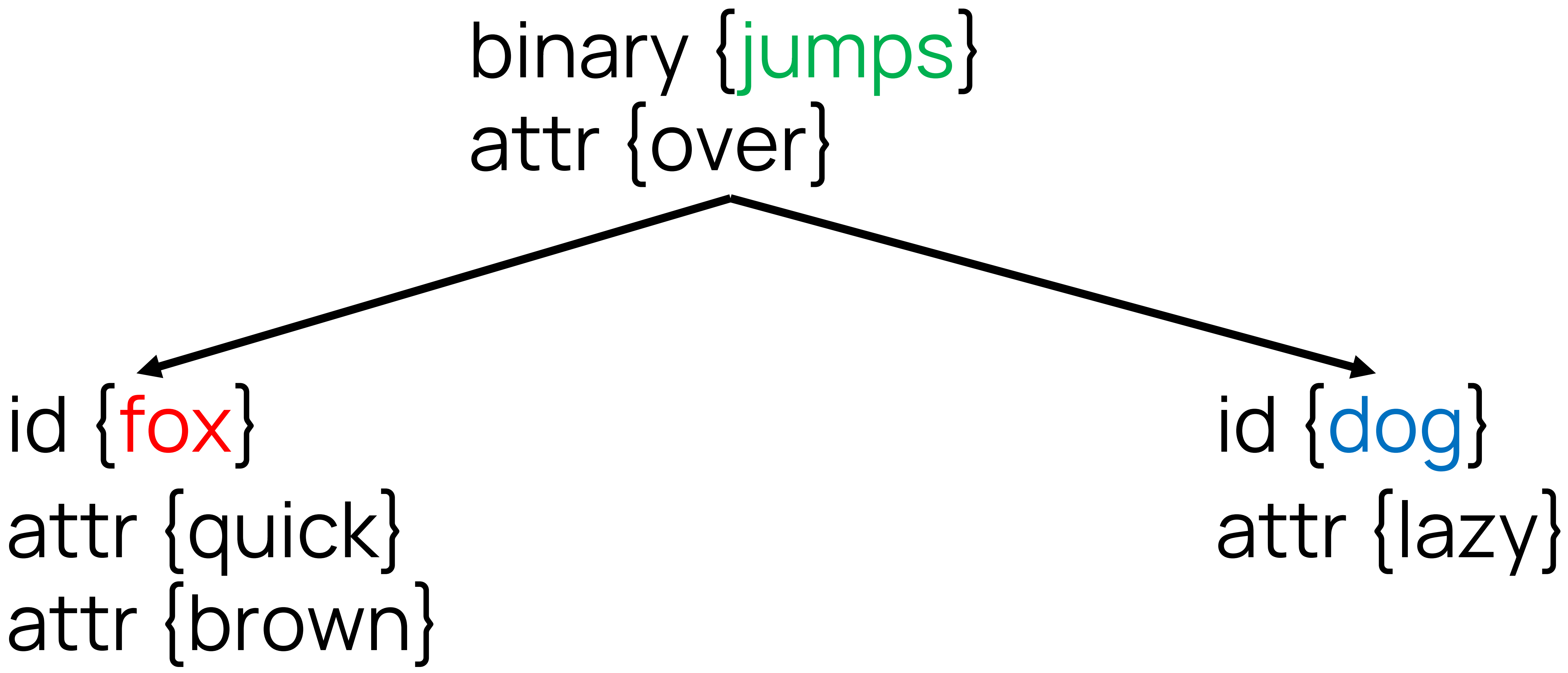
Semantics

the branch of linguistics and logic concerned with meaning

Syntax vs Semantics

The quick brown fox jumps over the lazy dog

Syntax vs Semantics



binary {jumps}
attr {over}

id {fox}
attr {quick}
attr {brown}

id {dog}
attr {lazy}

Syntax vs Semantics

jump	accelerate upward while maintaining forward momentum
over	above an object

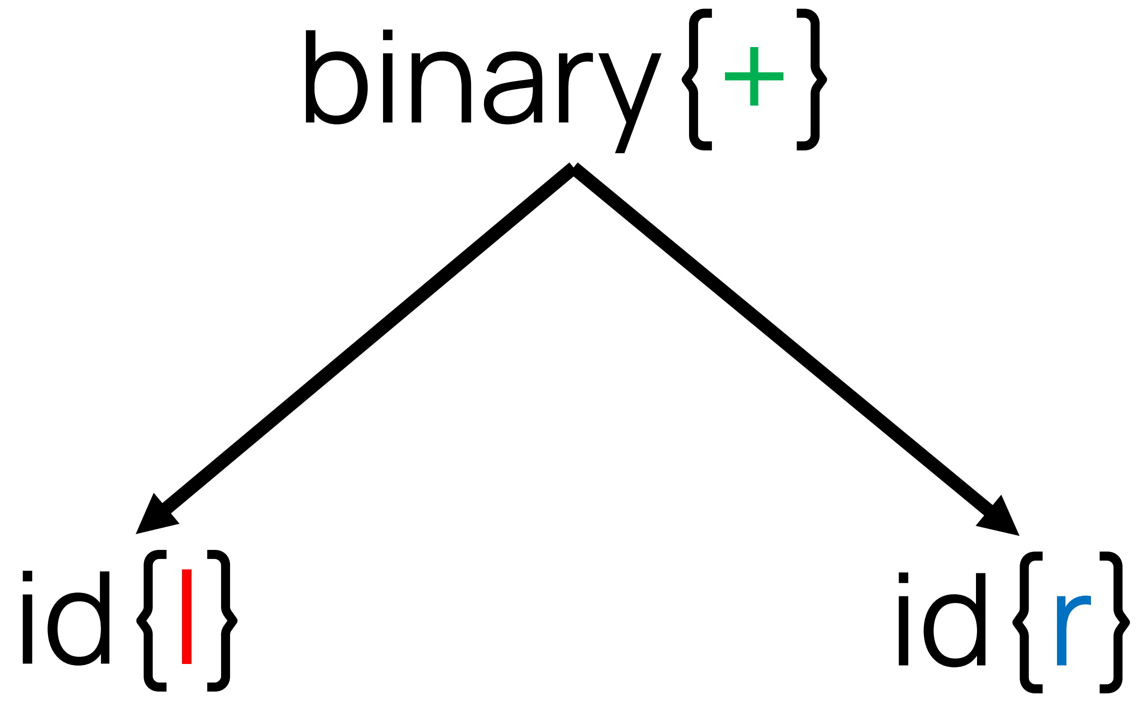
fox	animal, mammal, carnivore, canid, vulpes vulpes
quick	capable of fast actions
brown	RGB(150, 75, 0)

dog	animal, mammal, carnivore, canid, canis familiaris
lazy	unwilling to use energy

Syntax vs Semantics

```cpp
struct lhs;
struct rhs;

auto operator+(const lhs&, const rhs&);

auto meow()
{
  auto l = lhs{};
  auto r = rhs{};
  return l + r;
}
```

Syntax vs Semantics



+	function of (struct lhs, struct rhs)
l	variable of type struct lhs
r	variable of type struct rhs

Grammatic correctness and semantic nonsense

```
struct lhs;
struct rhs;

auto meow()
{
  auto l = lhs{};
  auto r = rhs{};
  return l + r;
}
```

Grammatic correctness and semantic nonsense

```
struct lhs;
struct rhs;

auto meow()
{
  auto l = lhs{};
  auto r = rhs{};
  return l + r;    <-- fail
}
```

Grammatic correctness and semantic nonsense

```
struct lhs;
struct rhs;

auto meow()
{
  auto l = lhs{}; <-- fail
  auto r = rhs{}; <-- fail
  return l + r;   <-- fail
}
```

Name Resolution

Simple case

```
void purr(int);

void meow(int a)
{
  purr(a);
}
```

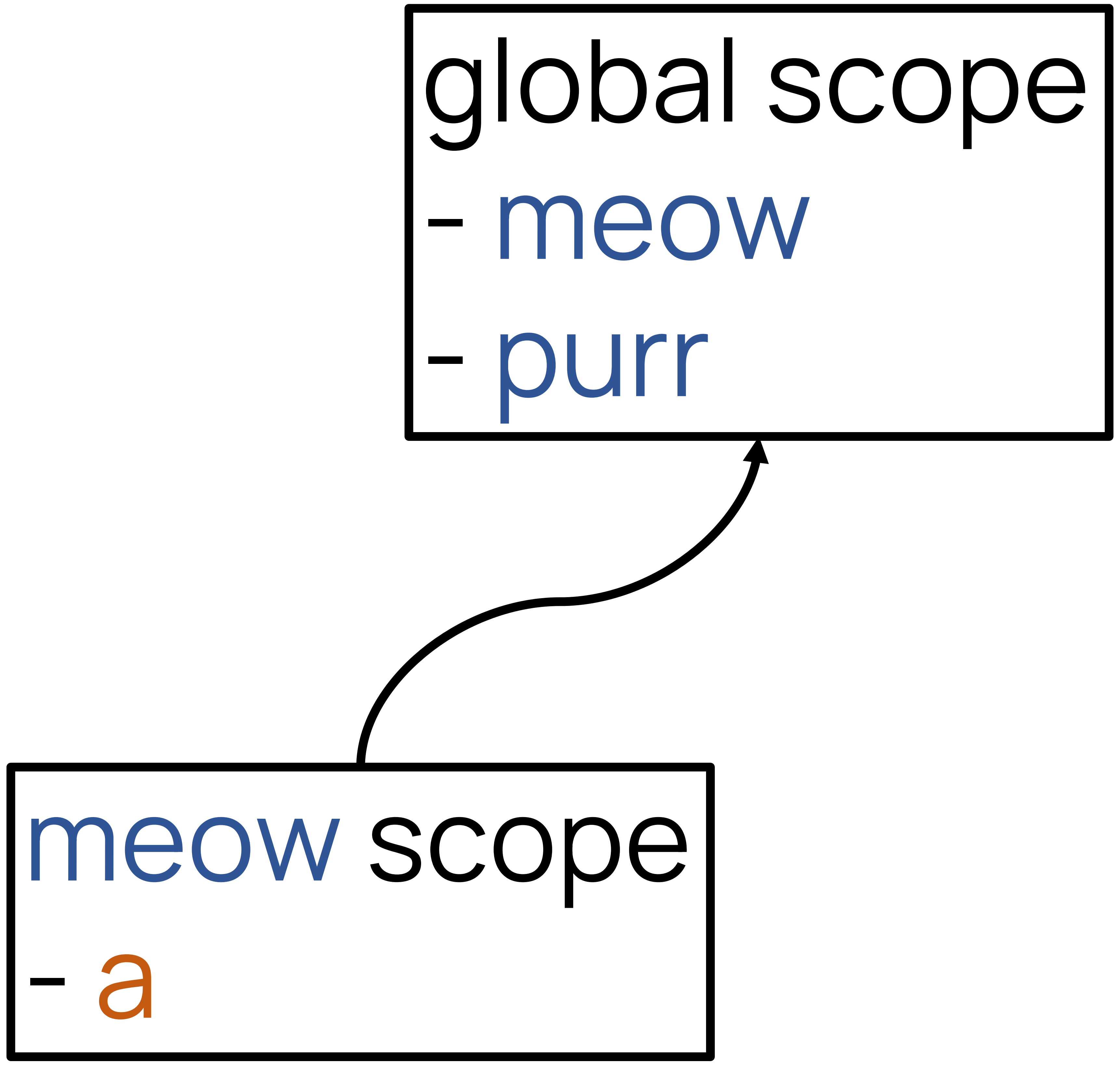
Simple case

meow

purr

a

Simple case



global scope
- meow
- purr

meow scope
- a

Symbol table

Symbol table

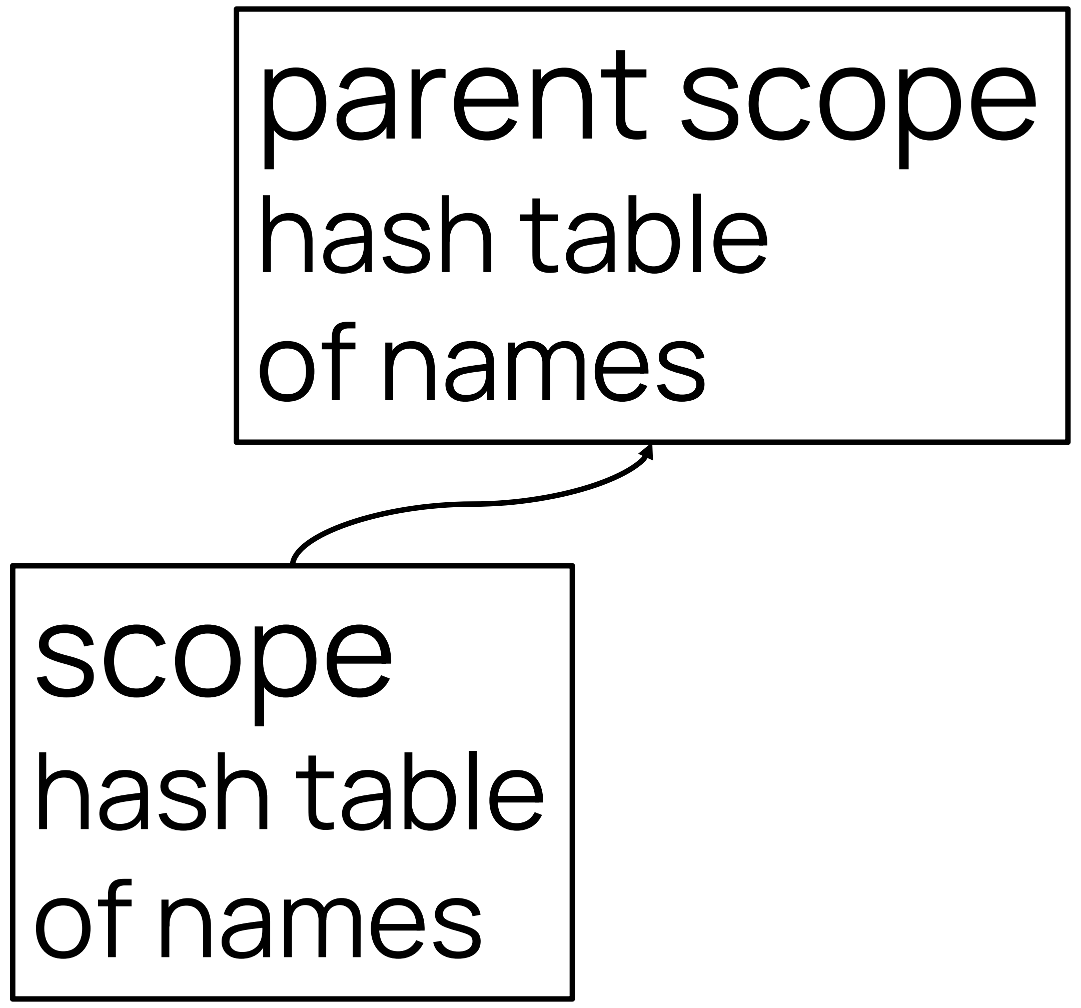
Scope-based approach

scope
hash table
of names

Name-based approach

Symbol table

Scope-based approach



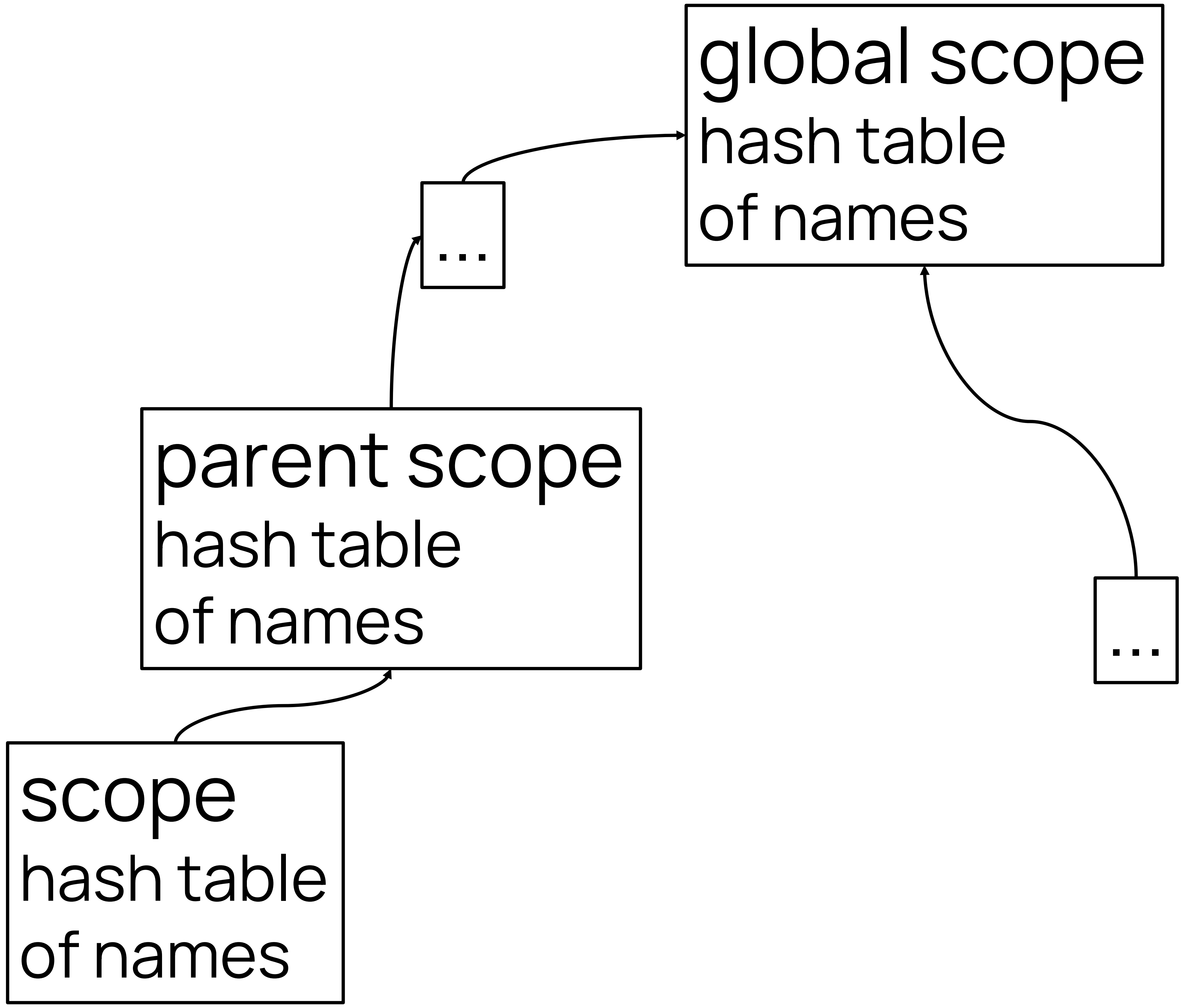
parent scope
hash table
of names

scope
hash table
of names

Name-based approach

Symbol table

Scope-based approach



global scope
hash table
of names

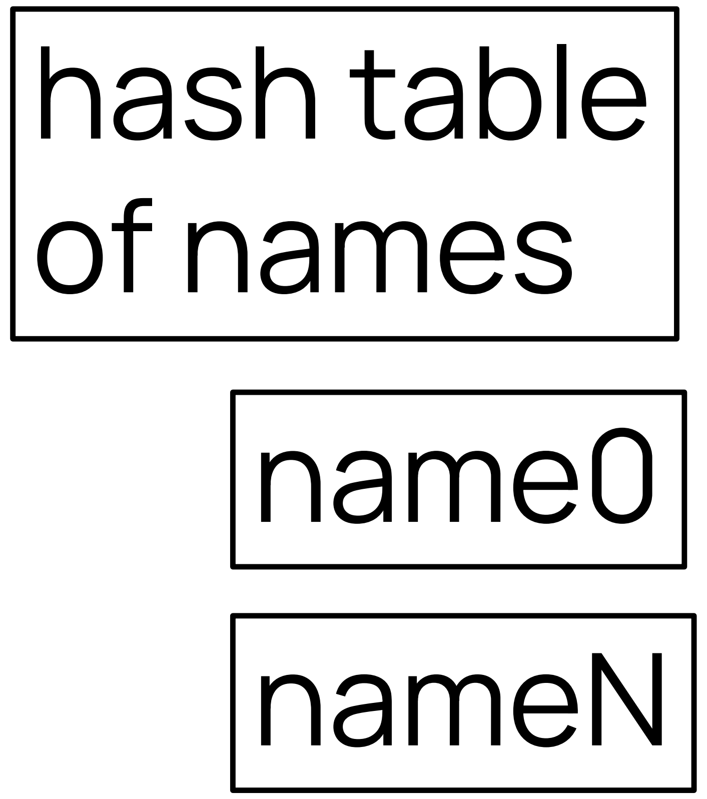
...

parent scope
hash table
of names

...

scope
hash table
of names

Name-based approach



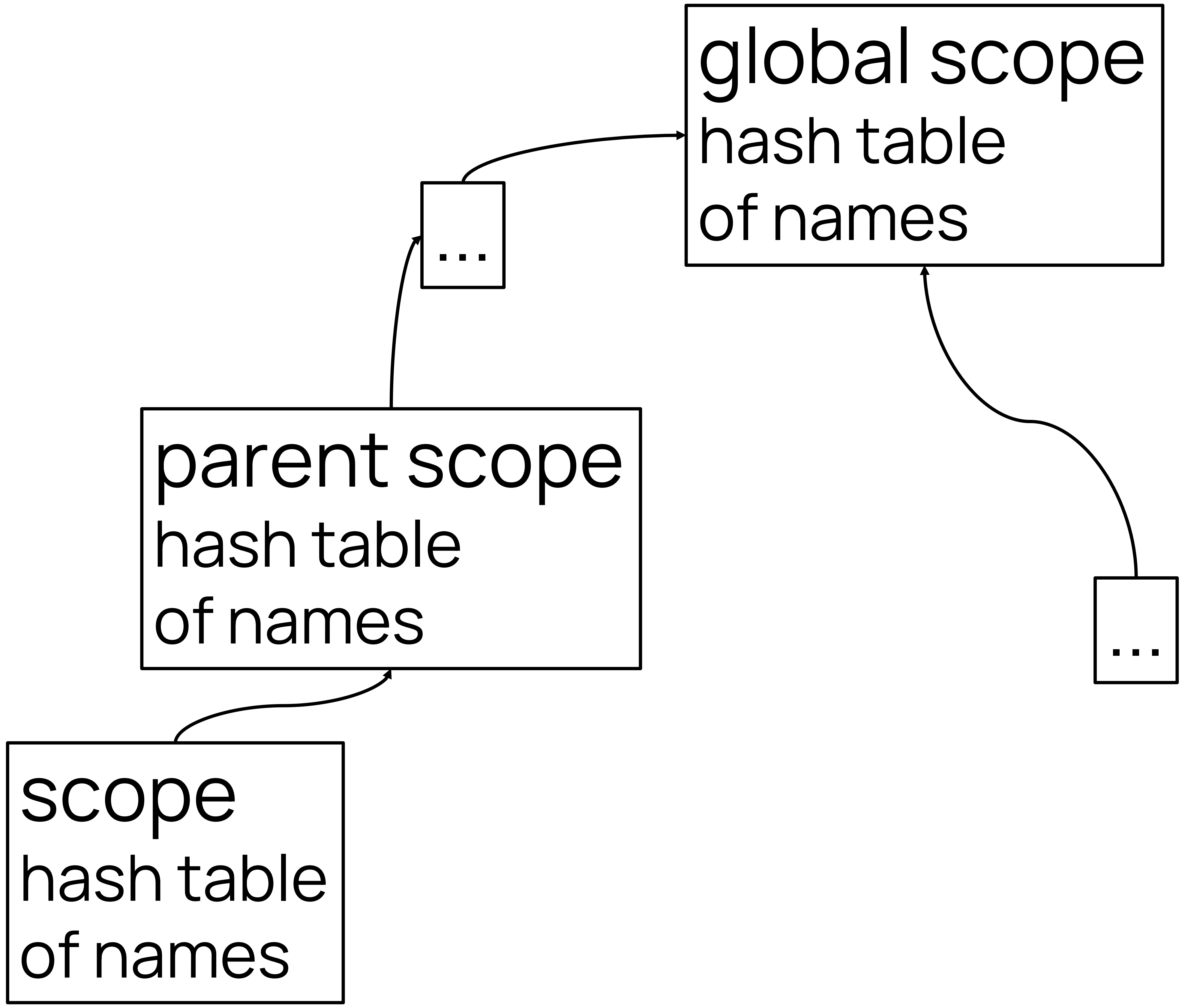
hash table
of names

name0

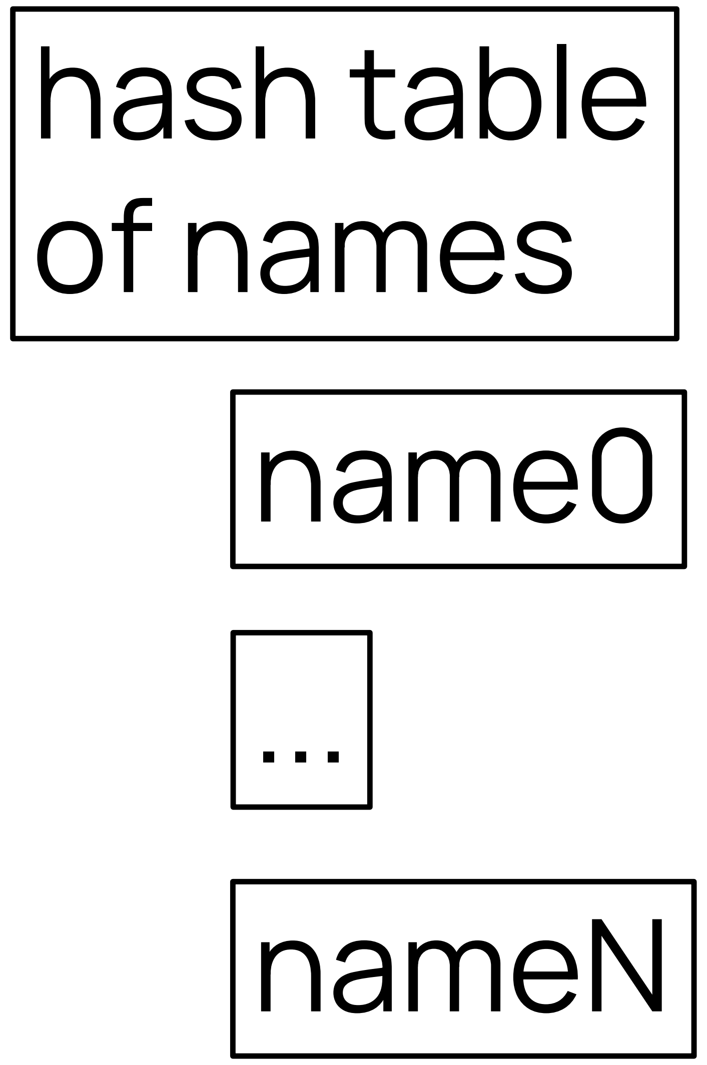
nameN

Symbol table

Scope-based approach

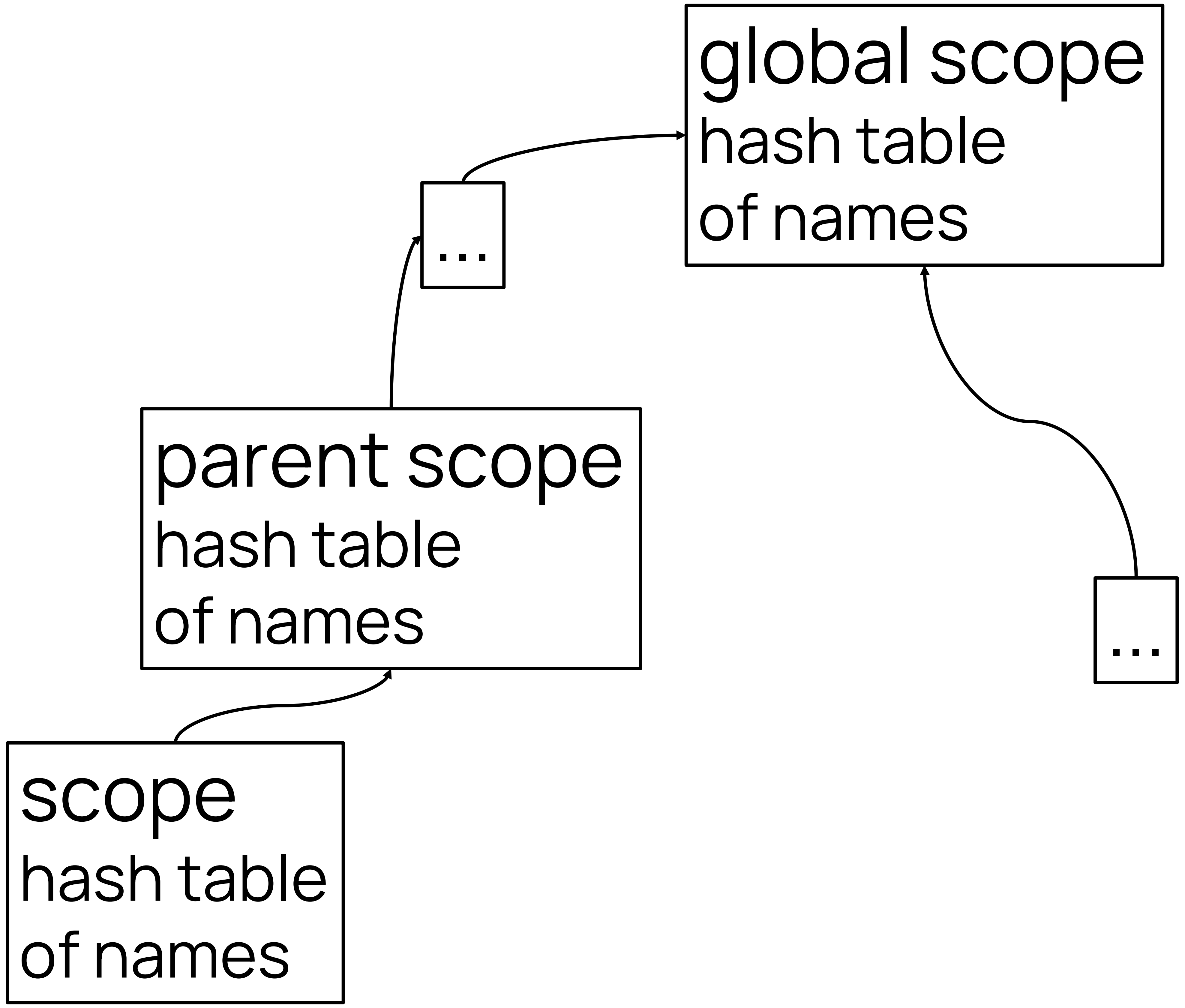


Name-based approach



Symbol table

Scope-based approach



global scope
hash table
of names

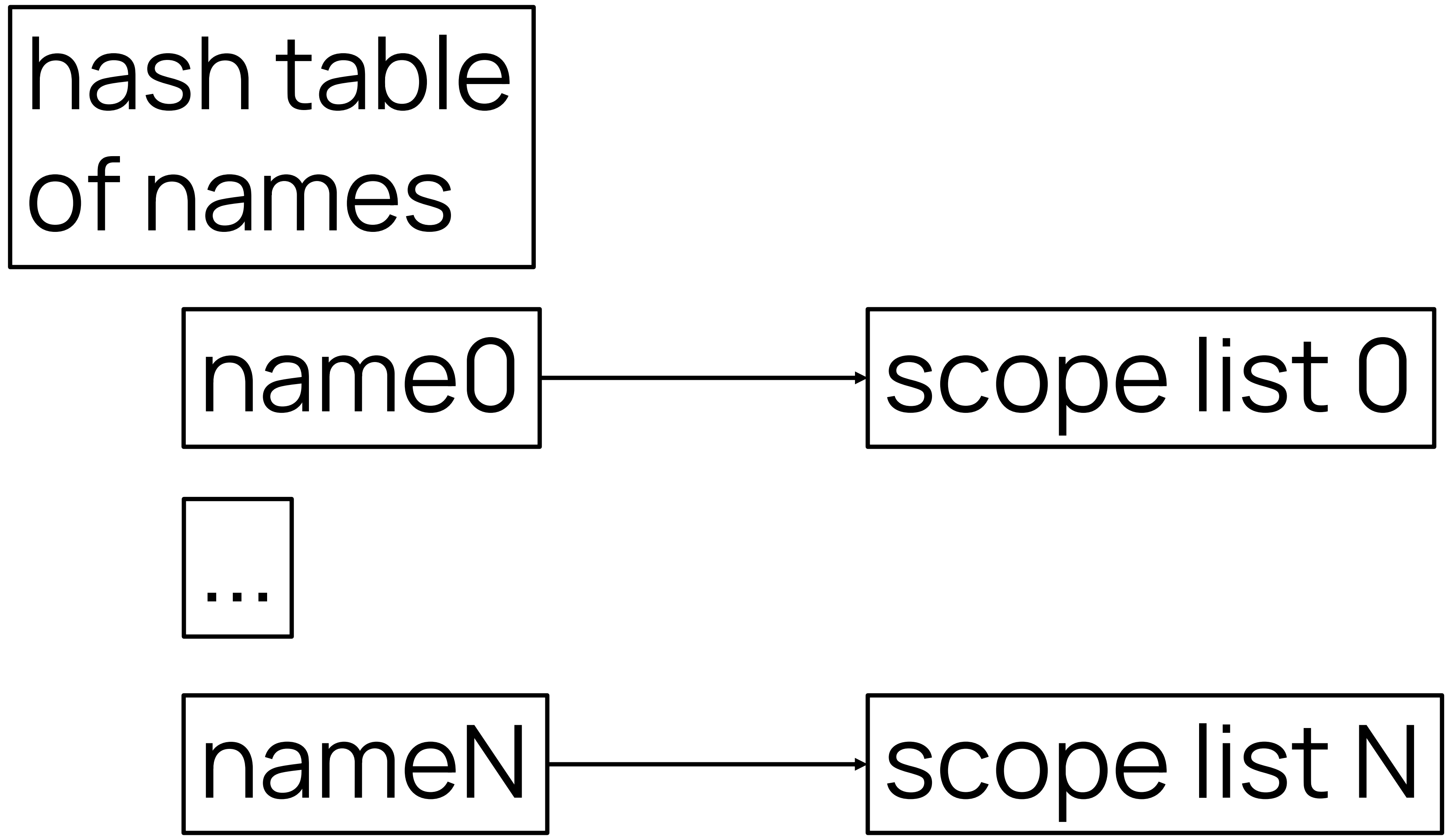
...

parent scope
hash table
of names

...

scope
hash table
of names

Name-based approach



hash table
of names

name0

scope list 0

...

nameN

scope list N

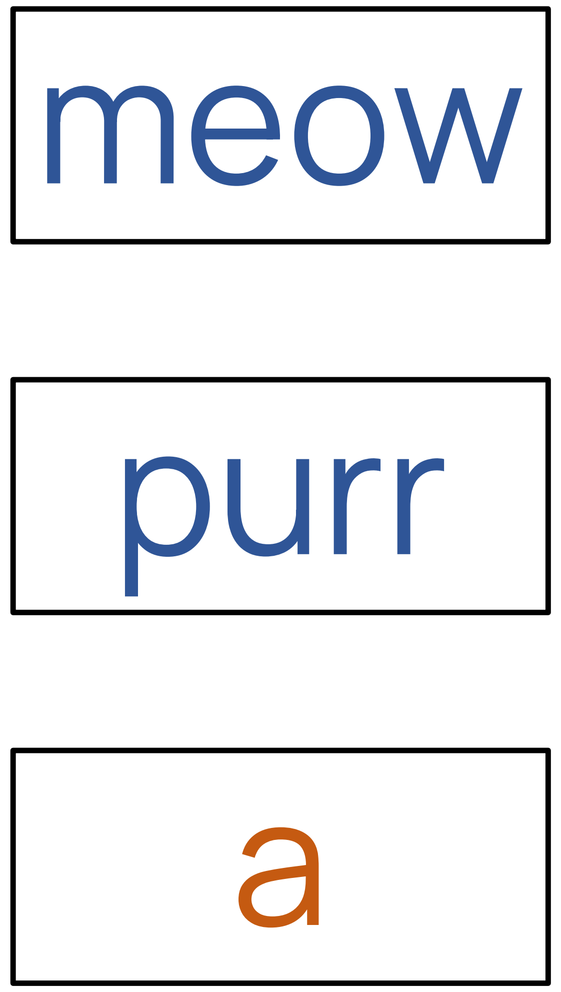
Symbol table. Name-based

meow

purr

a

Symbol table. Name-based



meow

purr

a

Symbol table. Name-based

`	null	scope
meow	`	func set
purr	`	func set
a	meow	variable

Unqualified lookup

Unqualified lookup

```
void purr(int);

void meow(int a)
{
  purr(a);
}
```

`	null	scope
meow	`	func set
purr	`	func set
a	meow	variable

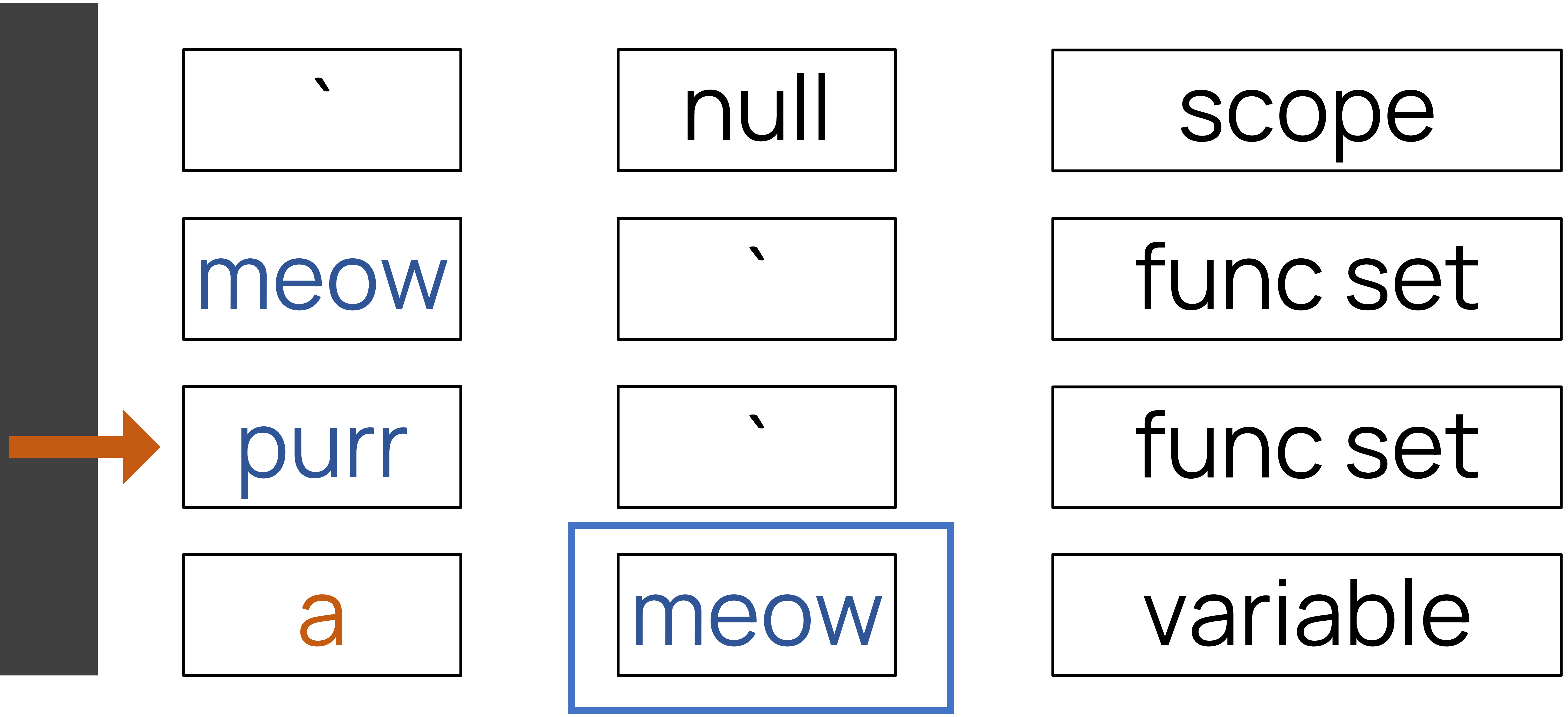
Unqualified lookup

```
void purr(int);

void meow(int a)
{
  purr(a);
}
```

`	null	scope
meow	`	func set
purr	`	func set
a	meow	variable

Unqualified lookup

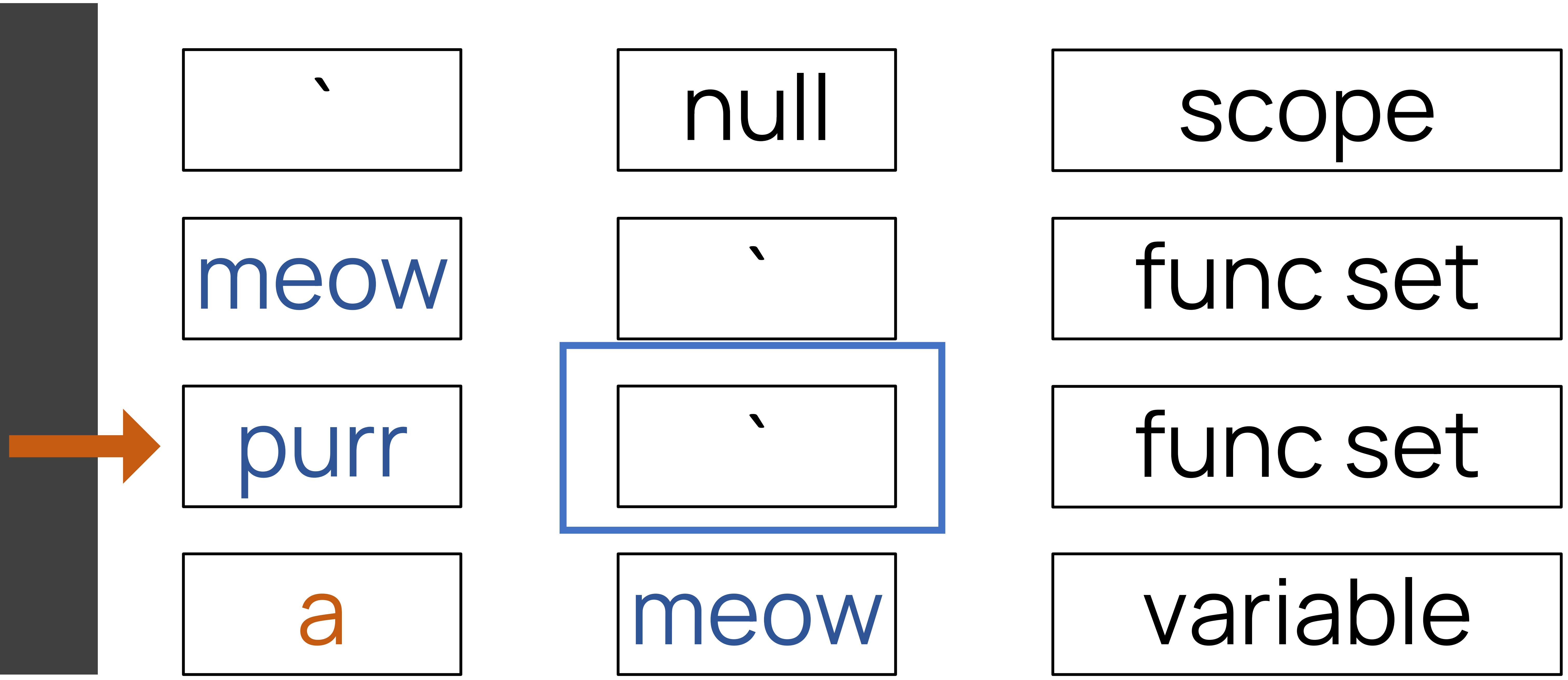
```
void purr(int);

void meow(int a)
{
  purr(a);
}
```



`	null	scope
meow	`	func set
purr	`	func set
a	meow	variable

Unqualified lookup

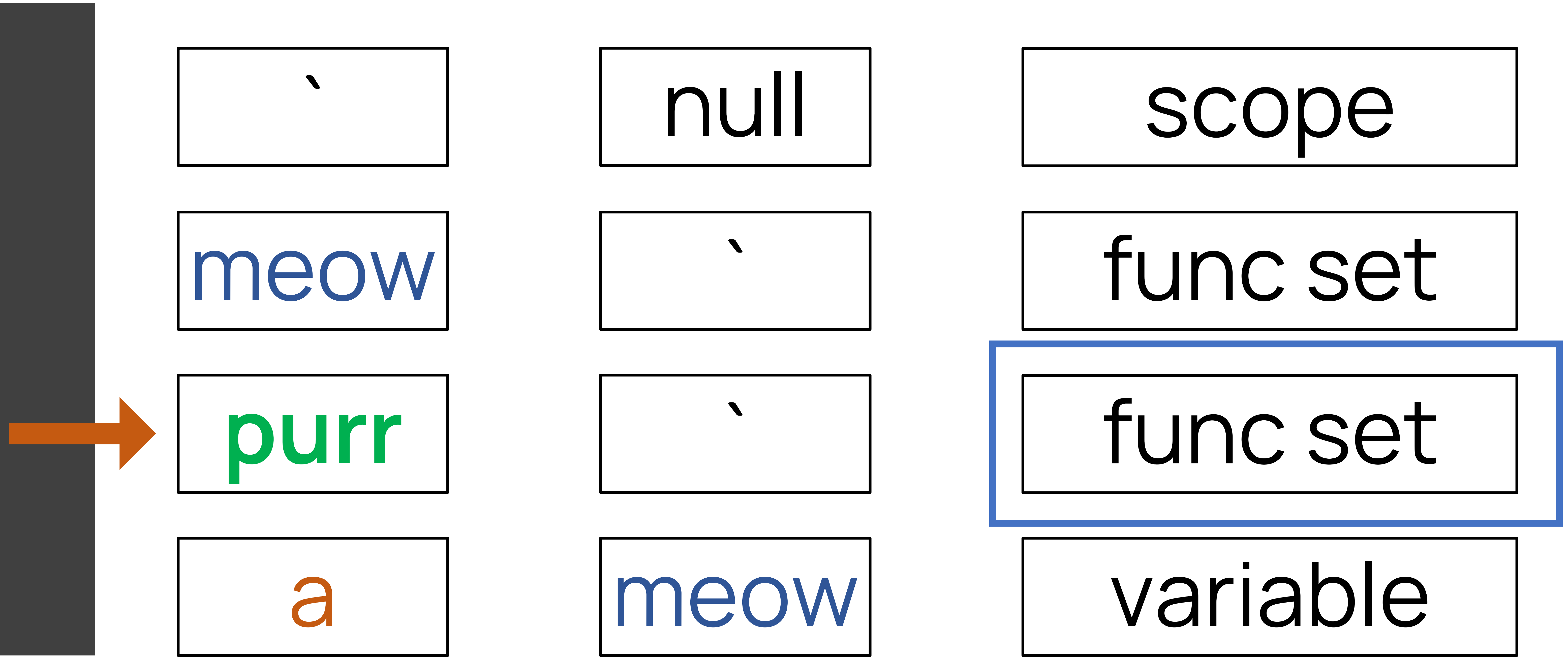
```
void purr(int);

void meow(int a)
{
  purr(a);
}
```



`	null	scope
meow	`	func set
purr	`	func set
a	meow	variable

Unqualified lookup

```
void purr(int);

void meow(int a)
{
  purr(a);
}
```



`	null	scope
meow	`	func set
purr	`	func set
a	meow	variable

Unqualified lookup

purr: func set{ void(int) } at scope { ` }

Unqualified lookup

purr: func set{ void(int) } at scope { ` }

a: variable{ int } at scope { meow }

Unqualified lookup

```
void purr(int);

void meow(int a)
{
  purr(a);
}
```

Unqualified lookup

```
namespace thing
{
  void purr(int);
}

void meow(int a)
{
  purr(a);
}
```

Unqualified lookup

```cpp
namespace thing
{
  void purr(int);
}

void meow(int a)
{
  thing::purr(a);
}
```

Qualified lookup

Qualified lookup

```
namespace thing
{
  void purr(int);
}

void meow(int a)
{
  thing::purr(a);
}
```

`	null	scope
meow	`	func set
purr	`	func set
a	meow	variable

Qualified lookup

```
namespace thing
{
  void purr(int);
}

void meow(int a)
{
  thing::purr(a);
}
```

`	null	scope
thing	`	scope
meow	`	func set
purr	thing	func set
a	meow	variable

Qualified lookup

```
namespace thing
{
  void purr(int);
}

void meow(int a)
{
  thing::purr(a);
}
```

`	null	scope
thing	`	scope
meow	`	func set
purr	thing	func set
a	meow	variable

thing: scope{ namespace } at scope { ` }

Qualified lookup

```
namespace thing
{
  void purr(int);
}

void meow(int a)
{
  thing::purr(a);
}
```

`	null	scope
thing	`	scope
meow	`	func set
purr	thing	func set
a	meow	variable

thing: scope{ namespace } at scope { ` }

Qualified lookup

```
namespace thing
{
  void purr(int);
}

void meow(int a)
{
  thing::purr(a);
}
```

`	null	scope
thing	`	scope
meow	`	func set
purr	thing	func set
a	meow	variable

thing: scope{ namespace } at scope { ` }

Qualified lookup

```
namespace thing
{
  void purr(int);
}

void meow(int a)
{
  thing::purr(a);
}
```

`	null	scope
thing	`	scope
meow	`	func set
purr	thing	func set
a	meow	variable

thing: scope{ namespace } at scope { ` }

Qualified lookup

thing: scope{ namespace } at scope { ` }
purr: func set{ void(int) } at scope { thing }

Obscure lookup

Obscure lookup

```
namespace ns
{
  struct thing{};
  void purr(thing);
}

void meow()
{
  ns::thing a{};
  purr(a);
}
```

nowhere to be found on the path { meow -> ` }

Visibility vs Reachability

Visibility vs Reachability

```
auto meow()
{
  struct thing{ void func(); };   <-- not visible outside meow
  return thing{};
}
```

Visibility vs Reachability

```
auto meow()
{
  struct thing{ void func(); };    <-- not visible outside meow
  return thing{};
}

void purr()
{
  meow().func();  <-- reachable outside meow
}
```

Visibility vs Reachability

```
struct thing
{
  void pub();

private:
  void pr();
};
```

Visibility vs Reachability

```
struct thing
{
  void pub();

private:
  void pr();
};

auto meow(thing th)
{
}
```

Visibility vs Reachability

```
struct thing
{
  void pub();

private:
  void pr();
};

auto meow(thing th)
{
  th.pub();
}
```

Visibility vs Reachability

```
struct thing
{
  void pub();    <-- visible outside thing

private:
  void pr();
};

auto meow(thing th)
{
  th.pub();   <-- reachable outside thing
}
```

Visibility vs Reachability

```
struct thing
{
  void pub();    <-- visible outside thing

private:
  void pr();
};

auto meow(thing th)
{
  th.pub();   <-- reachable outside thing
  th.pr();
}
```

Visibility vs Reachability

```
struct thing
{
  void pub();   <-- visible outside thing

private:
  void pr();    <-- visible outside thing
};

auto meow(thing th)
{
  th.pub();  <-- reachable outside thing
  th.pr();   <-- NOT reachable outside thing
}
```

Obscure lookup aka ADL

```
namespace ns
{
  struct thing{};
  void purr(thing);
}

void meow()
{
  ns::thing a{};
  purr(a);
}
```

`	null	scope
ns	`	scope
thing	ns	record
meow	`	func set
purr	ns	func set
a	meow	variable

Obscure lookup aka ADL

```
namespace ns
{
  struct thing{};
  void purr(thing);
}

void meow()
{
  ns::thing a{};
  purr(a);
}
```

`	null	scope
ns	`	scope
thing	ns	record
meow	`	func set
purr	ns	func set
a	meow	variable

Obscure lookup aka ADL

```
namespace ns
{
  struct thing{};
  void purr(thing);
}

void meow()
{
  ns::thing a{};
  purr(a);
}
```

`	null	scope
ns	`	scope
thing	ns	record
meow	`	func set
purr	ns	func set
a	meow	variable

Obscure lookup aka ADL

```
namespace ns
{
  struct thing{};
  void purr(thing);
}

void meow()
{
  ns::thing a{};
  purr(a);
}
```

`	null	scope
ns	`	scope
thing	ns	record
meow	`	func set
purr	ns	func set
a	meow	variable

Obscure lookup aka ADL

purr: func set{ void(ns::thing) } at scope { ns }

Imports

Imports

```
namespace thing
{
  void purr();
}

void meow()
{
  using thing::purr;
  purr();
}
```

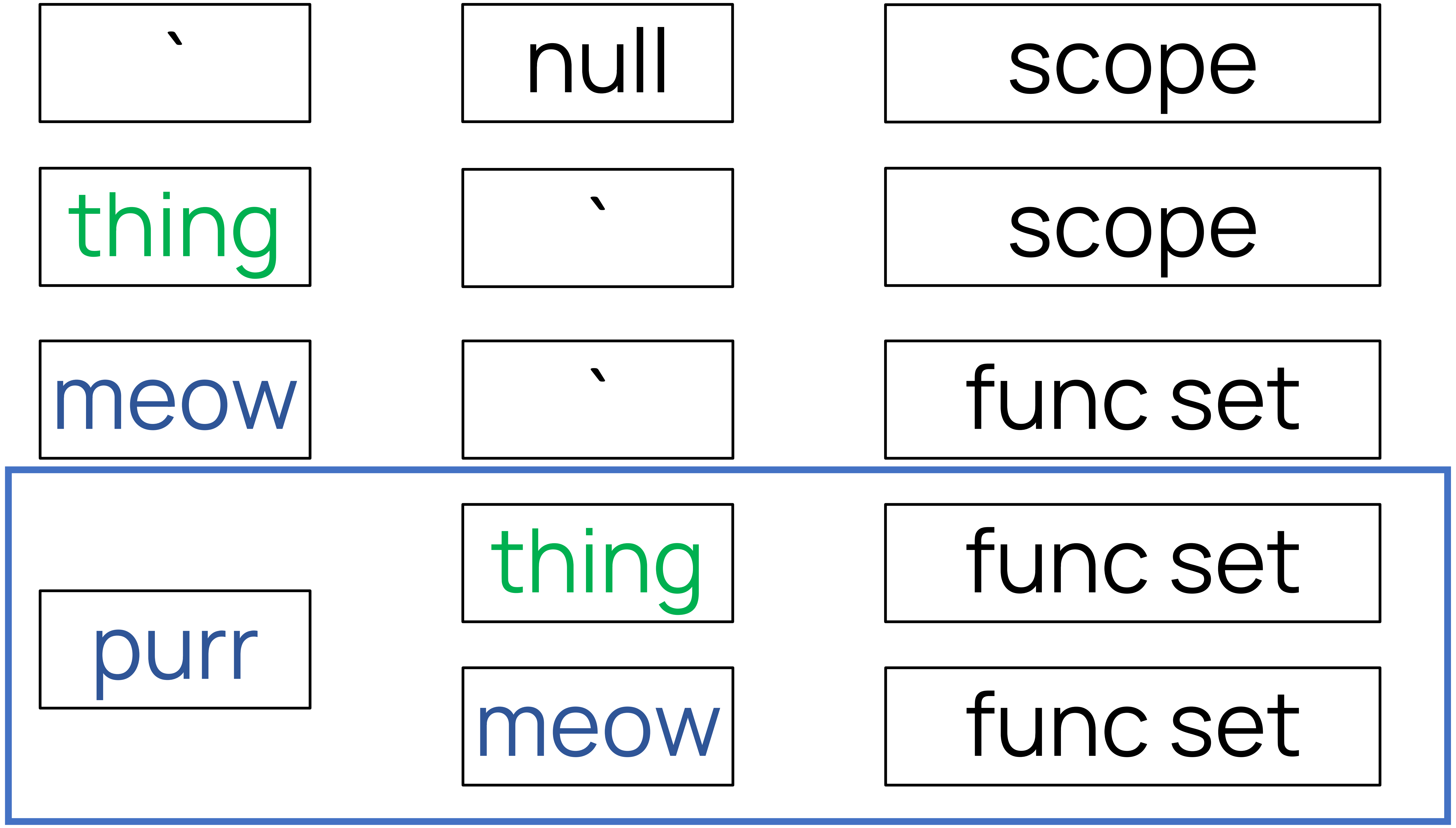
Imports

```
namespace thing
{
  void purr();
}

void meow()
{
  using thing::purr;
  purr();
}
```

`	null	scope
thing	`	scope
meow	`	func set
purr	thing	func set

Imports

```
namespace thing
{
  void purr();
}

void meow()
{
  using thing::purr;
  purr();
}
```



`	null	scope
thing	`	scope
meow	`	func set
purr	thing	func set
	meow	func set

Imports

```
namespace thing
{
  int variable = 0;
}

int variable = 42;

int meow()
{
  using thing::variable;
  return variable;
}
```

Imports

```
namespace thing
{
  void purr();
}

void meow()
{
  using namespace thing;
  purr();
}
```

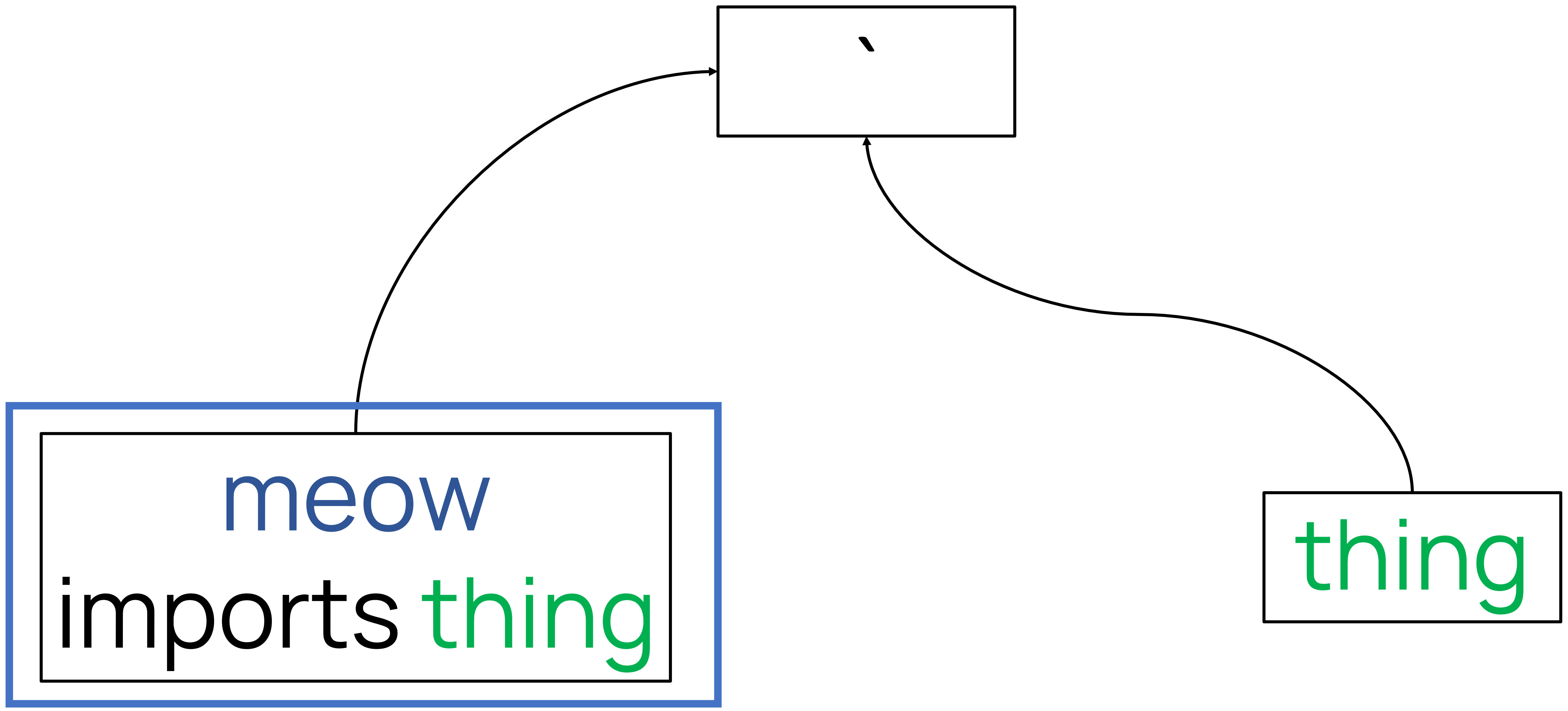
Imports

```
namespace thing
{
  void purr();
}

void meow()
{
  using namespace thing;
  purr();
}
```

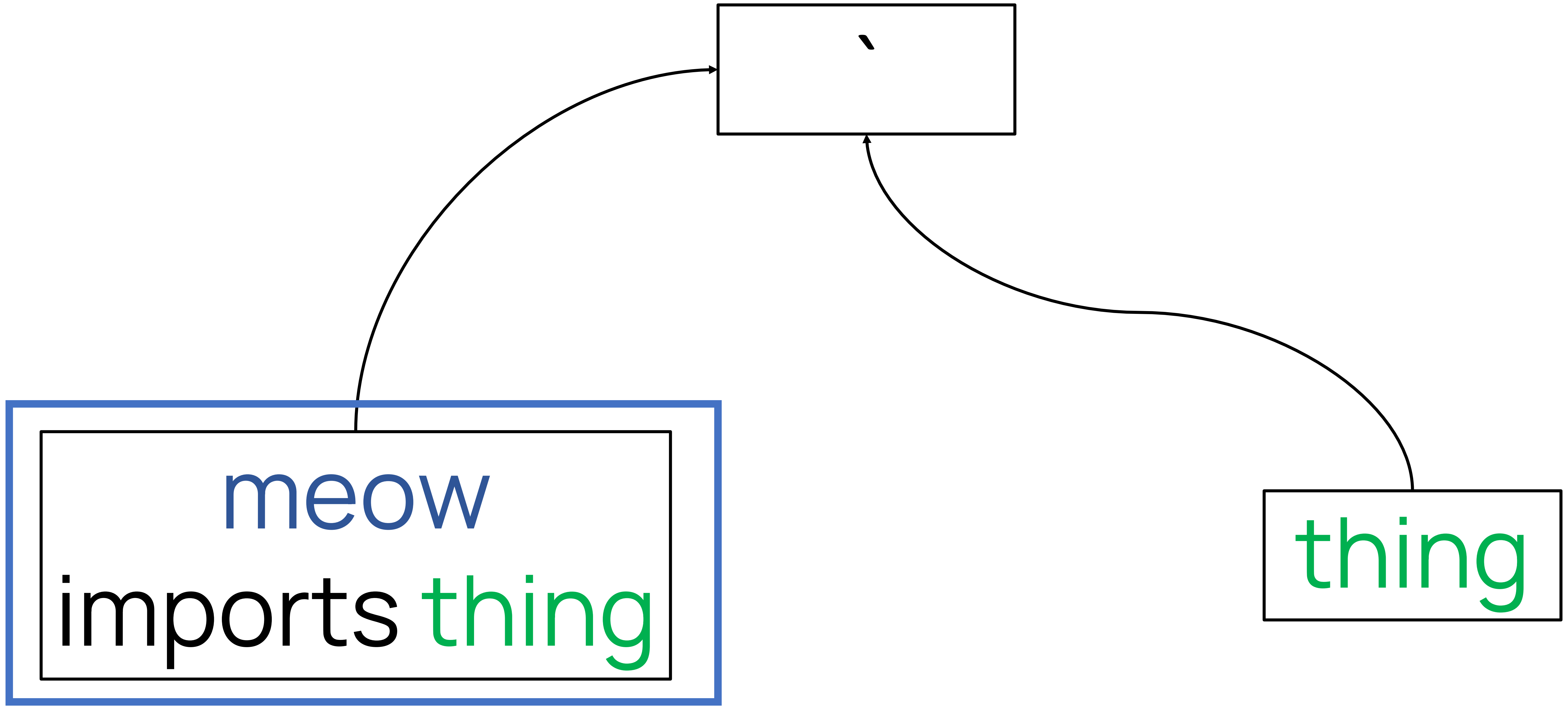
`	null	scope
thing	`	scope
meow	`	func set
purr	thing	func set

Imports

```
namespace thing
{
  void purr();
}

void meow()
{
  using namespace thing;
  purr();
}
```



Imports

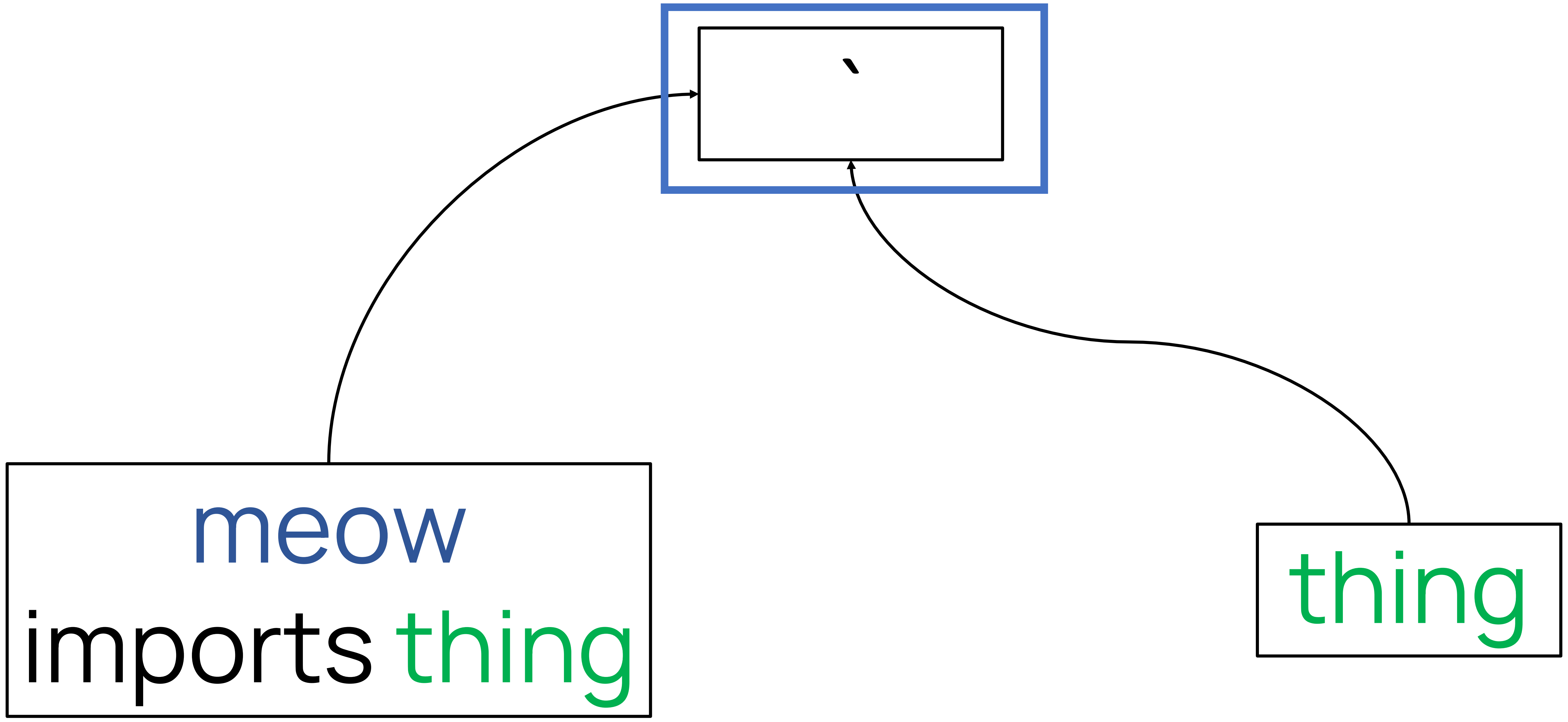


Imports
thing

purr is not in scope { meow }

there's no **direct path** from { thing } to { meow }

Imports

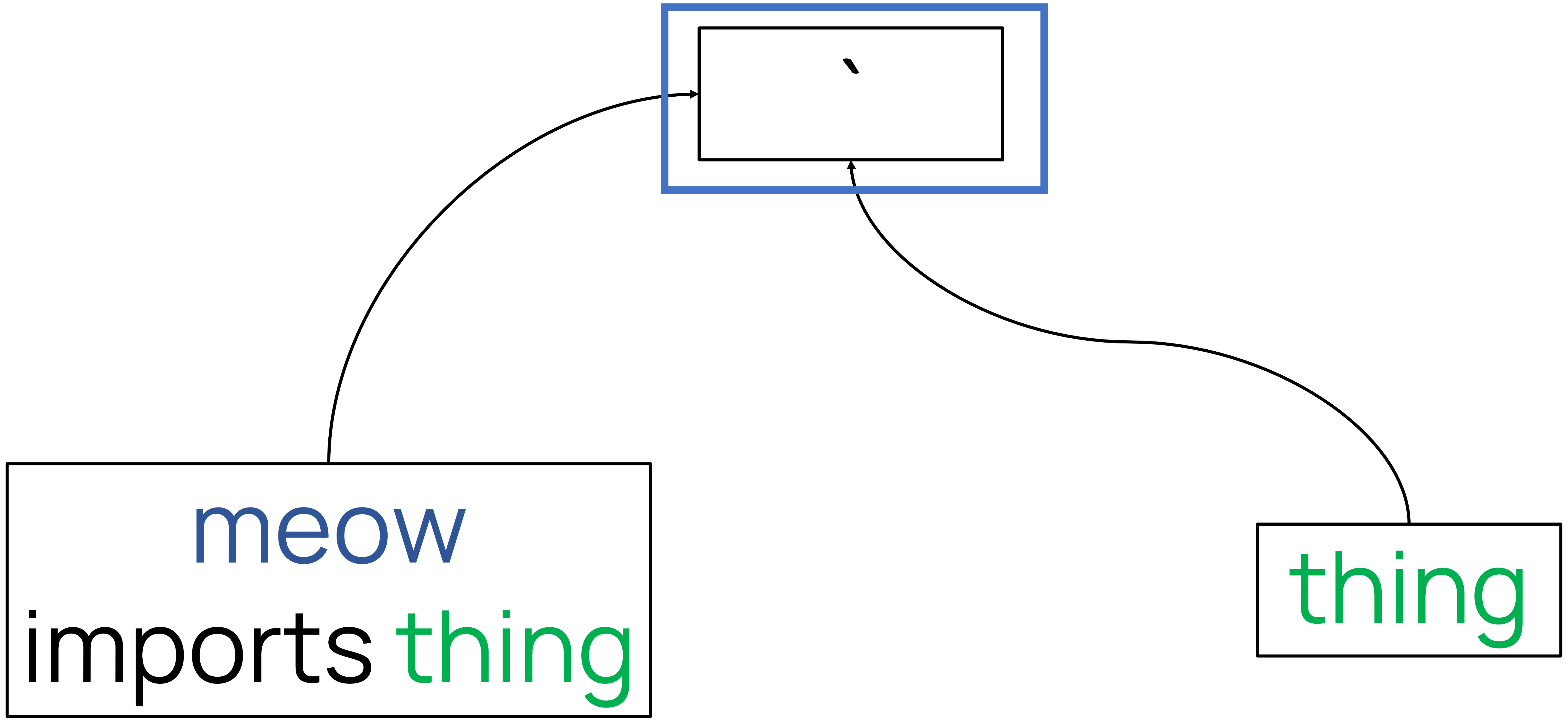


Imports
thing

purr is not in scope { meow }

there's no **direct path** from { thing } to { meow }

Imports

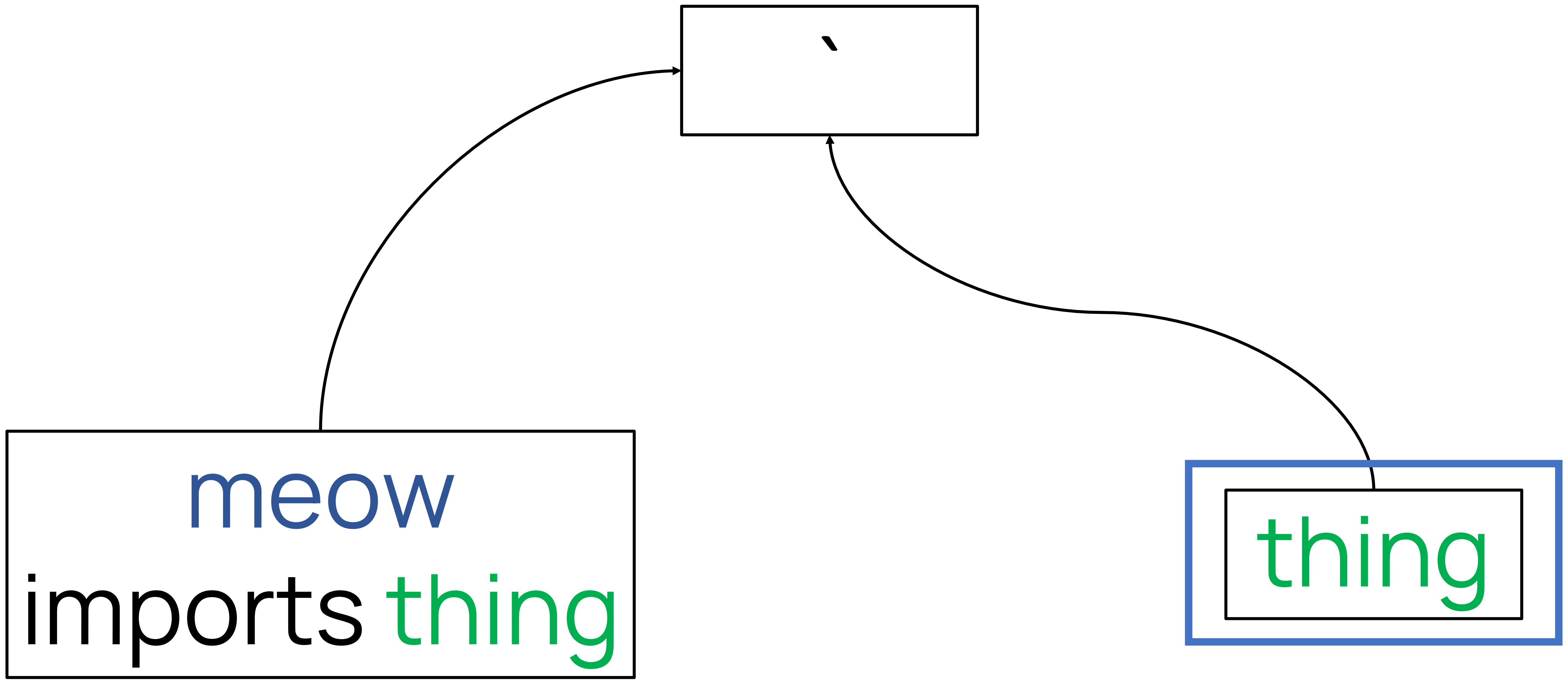


Imports
thing

purr is not in scope { ` }

there's a **direct path** from { thing } to { ` }

Imports

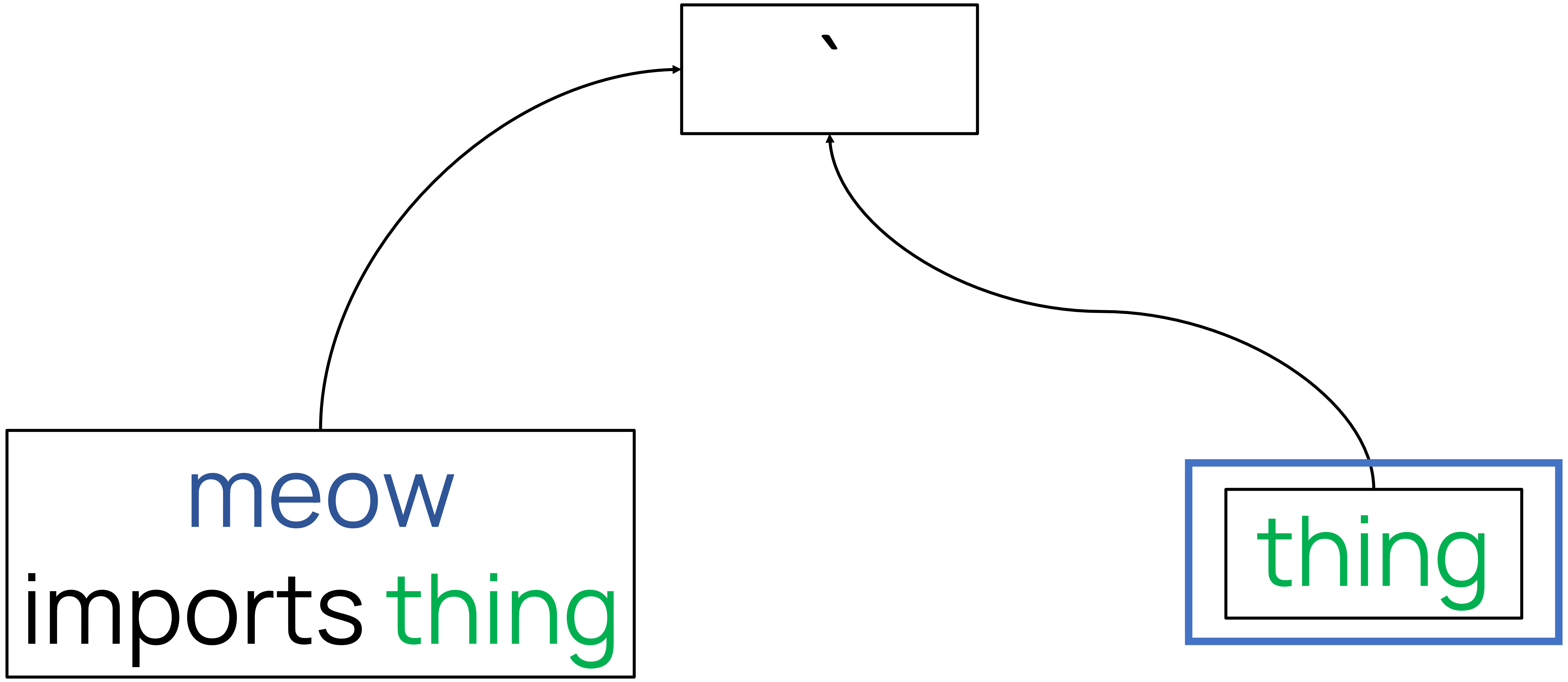


Imports

purr is not in scope { ` }

there's a **direct path** from { thing } to { ` }

Imports



Imports

purr is in scope { thing }

purr: func set{ void() } at scope { thing }

Why so complicated?

Why so complicated?

```
namespace thing
{
  int variable = 0;
}

int meow()
{
  using namespace thing;
  return variable;
}
```

Why so complicated?

```cpp
namespace thing
{
  int variable = 0;
}

int meow()
{
  int variable = 42;
  using namespace thing;
  return variable;
}
```

Why so complicated?

```
namespace thing
{
  int variable = 0;
}

int meow()
{
  int variable = 42;
  {
    using namespace thing;
    return variable;
  }
}
```

Why so complicated?

```
namespace thing
{
  int variable = 0;
}

int variable = 42;

int meow()
{
  {
    using namespace thing;
    return variable;  <-- ambiguous
  }
}
```

Why so complicated?

```
namespace thing
{
  int variable = 0;
}



 we start searching for thing::variable from here


int meow()
{
  int variable = 42;
  {
    using namespace thing;
    return variable;
  }
}
```

Why so complicated?

```
namespace thing
{
  int variable = 0;
}


 we start searching for thing::variable from here
 despite 'using namespace' being buried deep inside meow


int meow()
{
  int variable = 42;
  {
    using namespace thing;
    return variable;
  }
}
```

Classes

Classes

```
struct thing
{
};
```

Classes

```
struct thing
{
  void meow()
  {
    purr();
  }
};
```

Classes

```
struct thing
{
  void meow()
  {
    purr();
  }
  void purr()
  {
    field = 42;
  }
};
```

Classes

```
struct thing
{
  void meow()
  {
    purr();
  }
  void purr()
  {
    field = 42;
  }
  int field{};
};
```

Classes

```
struct thing
{
  void meow()
  {
    purr();
  }
  void purr()
  {
    field = 42;
  }
  int field{};
};
```

Classes can't be parsed in a single pass

Classes. Pass #1

```
struct thing
{
  void meow()
  {
    purr();
  }
  void purr()
  {
    field = 42;
  }
  int field{};
};
```

Classes. Pass #1

```
struct thing
{
  void meow()
  {
    purr();
  }
  void purr()
  {
    field = 42;
  }
  int field{};
};
```

Classes. Pass #1

```
struct thing
{
  void meow();
  void purr();   type is incomplete here
  int field{};
};
```

Classes. Pass #1

```
struct thing
{
  void meow();
  void purr();   type is incomplete here
  int field{};
};

 type becomes complete here
```

Classes. Pass #1

```
struct thing
{
  void meow();
  void purr();   type is incomplete here
  int field{};
};

 type becomes complete here

 now we can parse func bodies
```

Classes. Pass #2

```
struct thing
{
  void meow();
  void purr();
  int field{};
};
```

```
void thing::meow()
{
  purr();
}

void thing::purr()
{
  field = 42;
}
```

Function overloads

Function overloads

```
void meow(int);
void meow(int&, int);
void meow(int*, int);
void meow(float, int);
```

Function overloads

```
void meow(int);
void meow(int&, int);
void meow(int*, int);
void meow(float, int);

// somewhere else
meow(1, 2);
```

Function overloads

```
void meow(int);
void meow(int&, int);
void meow(int*, int);
void meow(float, int);

// somewhere else
meow(1, 2);
```

meow | ` | func set

meow:

```
func set{ void(int),
          void(int&, int),
          void(int*, int),
          void(int, float) }
at scope { ` }
```

Function overloads

```
call: meow(rvalue: 1, rvalue: 2);
```

meow:

```
 func set{ void(int),                      (match,      empty)
           void(int&, int),                (l-ref bind, match)
           void(int*, int),                (invalid,    match)
           void(int, float) }              (match,      int-to-float)
 at scope { ` }
```

Function overloads

```
call: meow(rvalue: 1, rvalue: 2);
```

void(int)	(match,	empty)
void(int&, int)	(l-ref bind,	match)
void(int*, int)	(invalid,	match)
void(int, float)	(match,	int-to-float)

Function overloads

```
call: meow(rvalue: 1, rvalue: 2);
```

void(int)	(match,	empty)
void(int&, int)	(invalid,	match)
void(int*, int)	(invalid,	match)
void(int, float)	(match,	int-to-float)

Function overloads

```
call: meow(rvalue: 1, rvalue: 2);
```

void(int)	(match,	empty)
void(int, float)	(match,	int-to-float)
void(int&, int)	(invalid,	match)
void(int*, int)	(invalid,	match)

Function overloads

```
call: meow(rvalue: 1, rvalue: 2);
```

void(int)	(match,	empty)
void(int, float)	(match,	int-to-float)
void(int&, int)	(invalid,	match)
void(int*, int)	(invalid,	match)

Function overloads

```
call: meow(rvalue: 1, rvalue: 2);
```

void(int, float)	(match,	int-to-float)
void(int)	(match,	empty)
void(int&, int)	(invalid,	match)
void(int*, int)	(invalid,	match)

Function overloads

```
call: meow(rvalue: 1, rvalue: 2);
```

void(int, float) best match

void(int)

void(int&, int)

void(int*, int)

Templates

Templates

```
template <typename T>
struct thing
{
};
```

Templates

```
template <typename T>
struct thing
{
  void do_stuff()
  { }
};
```

Templates

```cpp
template <typename T>
struct thing
{
  void do_stuff()
  { }

  int field{};
};
```

Templates

```
template <typename T>
struct thing
{
  void do_stuff()
  {
    static_assert(false);
  }

  int field{};
};
```

Templates

```
template <typename T>
struct thing
{
  void do_stuff()
  {
    static_assert(false);
  }

  int field{};
};
```

```
template <>
struct thing<int>
{
  void do_stuff()
  {
    // something
  }
};
```

Templates

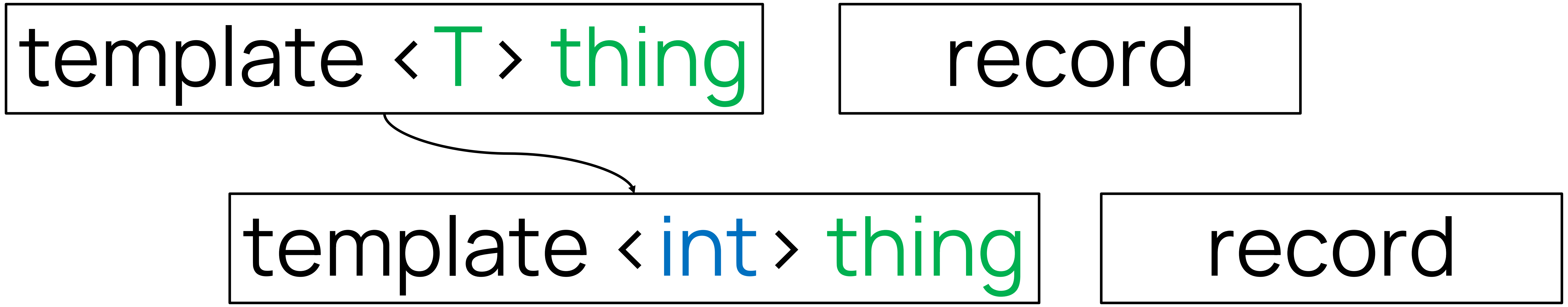
```
template <typename T>
struct thing
{
  void do_stuff()
  {
    static_assert(false);
  }

  int field{};
};
```

```
template <>
struct thing<int>
{
  void do_stuff()
  {
    // something
  }
};
```

```
void meow(thing<float>& th)
{
  // ...
}
```

Templates

```
void meow(thing<float>& th)
{
  // ...
}
```



template ‹T› thing

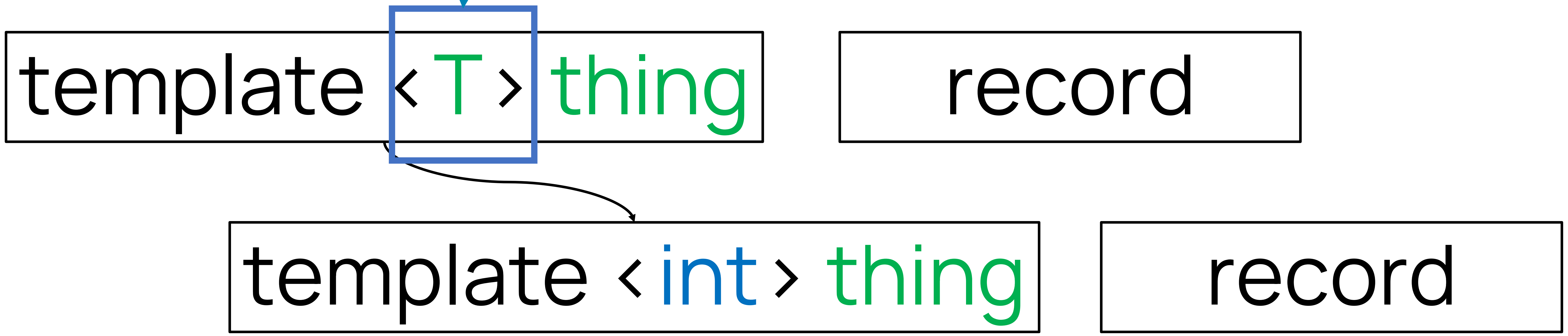
record

template ‹int› thing

record

Templates

```
void meow(thing<float>& th)
{
  // ...
}
```



template <T> thing

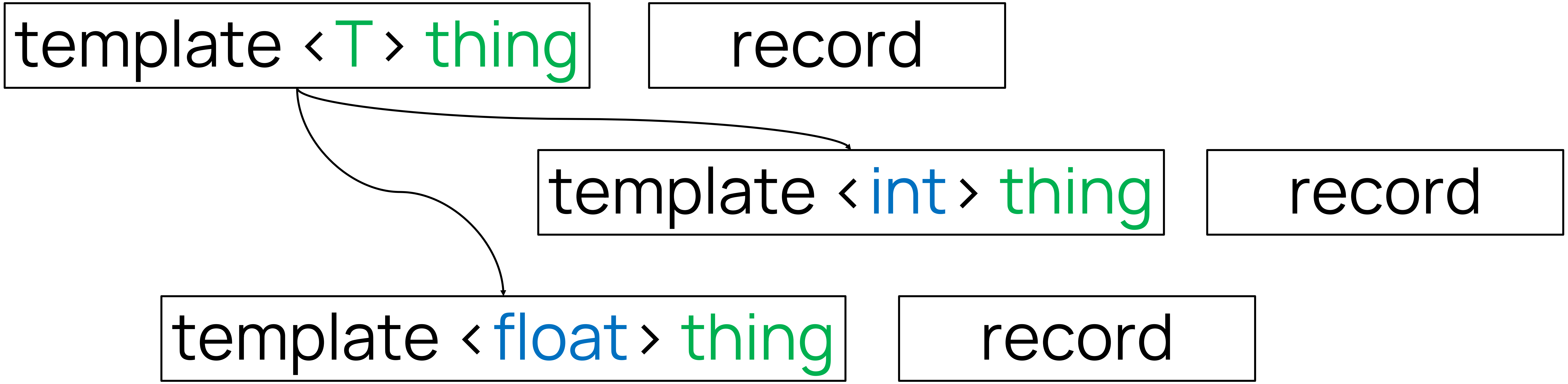
record

template <int> thing

record

Templates

```
void meow(thing<float>& th)
{
   // ...
}
```



template ‹T› thing

record

template ‹int› thing

record

template ‹float› thing

record

Templates

```
void meow(thing<float>& th)
{
  // ...
}

template <> struct thing<float>
{
 // ???
};
```

Templates

```
void meow(thing<float>& th)
{
  // ...
}

template <> struct thing<float>
{
};
```

Templates

```
void meow(thing<float> th)
{
  // ...
}

template <> struct thing<float>
{
};
```

Templates

```
void meow(thing<float> th)
{
  // ...
}

template <> struct thing<float>
{
  int field{};
};
```

Templates

```
void meow(thing<float> th)
{
  th.do_stuff();
}

template <> struct thing<float>
{
  int field{};
};
```

Templates

```
void meow(thing<float> th)
{
  th.do_stuff();
}
```

```
<source>:6:19: error: static assertion failed
    6 |       static_assert(false);
      |                     ^~~~~
<source>:14:6: note: in instantiation of member function 'thing<float>::do_stuff'
   14 |   th.do_stuff();
      |      ^
```

Templates are lazy

```
template <typename T> struct meow;
```

Templates are lazy

```
template <typename T> struct meow;

template <typename T, typename M = meow<T>>
struct purr
{
  using type = typename M::type;
};
```

Templates are lazy

```
template <typename T> struct meow;

template <typename T, typename M = meow<T>>
struct purr
{
  using type = typename M::type;
};

purr<int> hai();  // ok
```

Templates are lazy

```
template <typename T> struct meow;

template <typename T, typename M = meow<T>>
struct purr
{
  using type = typename M::type;
};

purr<int>::type hai();  // error
```

Templates are lazy

```cpp
template <typename T> struct meow;

template <typename T, typename M = meow<T>>
struct purr {
  using type = typename M::type;
};

template <typename T> struct meow {
  using type = T;
};

purr<int>::type hai();  // ok
```

Templates are lazy. But…

```
template <typename T>
struct thing
{
  void do_stuff()
  {
    static_assert(false);
  }

  int field{};
};
```

Templates are lazy. But…

```cpp
template <typename T>
struct thing
{
  void do_stuff()
  {
    static_assert(false);
  }

  int field{};
};

template struct thing<float>;
```

Templates are lazy. But...

```
template struct thing<float>;
```

```
<source>:6:19: error: static assertion failed
    6 |     static_assert(false);
      |                   ^~~~~
<source>:14:6: note: in instantiation of member function 'thing<float>::do_stuff'
   14 |   th.do_stuff();
      |      ^
```

Templates are lazy. But...

```
template struct thing<float>;
```

https://quuxplusone.github.io/blog/2021/08/06/dont-explicitly-instantiate-std-templates/

Constraints

Constraints

```
template <typename T> struct is_pointer
{
  static constexpr auto value = false;
};
```

Constraints

```
template <typename T> struct is_pointer
{
  static constexpr auto value = false;
};

template <typename T> struct is_pointer<T*>
{
  static constexpr auto value = true;
};
```

Constraints

```cpp
template <typename T> struct is_pointer
{
  static constexpr auto value = false;
};

template <typename T> struct is_pointer<T*>
{
  static constexpr auto value = true;
};

template <typename T>
concept ptr = is_pointer::value;
```

Constraints

```
template <typename T> struct is_pointer {
  static constexpr auto value = false; };

template <typename T> struct is_pointer<T*> {
  static constexpr auto value = true; };

template <typename T> concept ptr = is_pointer::value;
```

```
template <ptr T> struct thing { };
```

Constraints

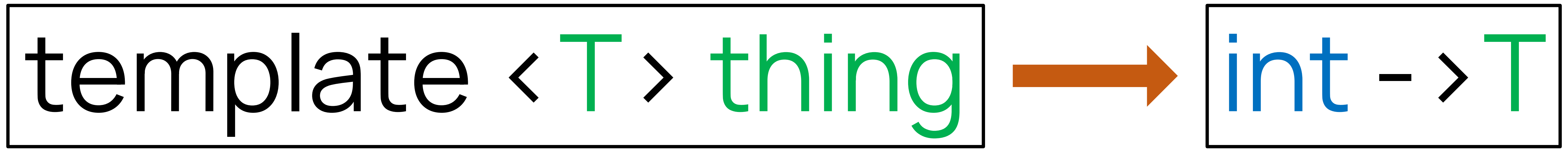
```
template <typename T> struct is_pointer {
  static constexpr auto value = false; };

template <typename T> struct is_pointer<T*> {
  static constexpr auto value = true; };

template <typename T> concept ptr = is_pointer::value;
```

```
template <ptr T>
struct thing
{
};
```

```
void meow()
{
  thing<int> th;  <-- fail
}
```

Constraints

```
thing<int> th;
```

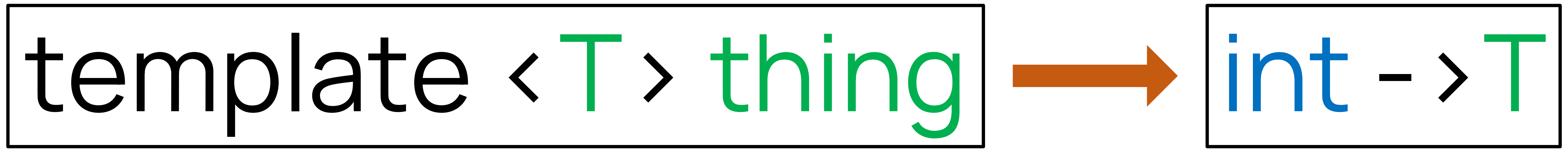


template <T> thing

int -> T

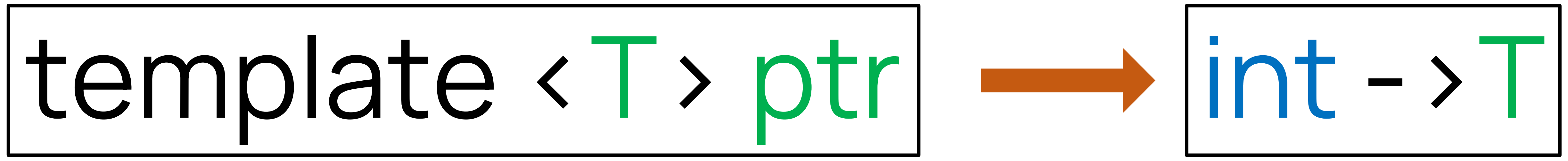
Constraints

```
thing<int> th;
```



template <T> thing → int -> T

```
ptr<int>
```



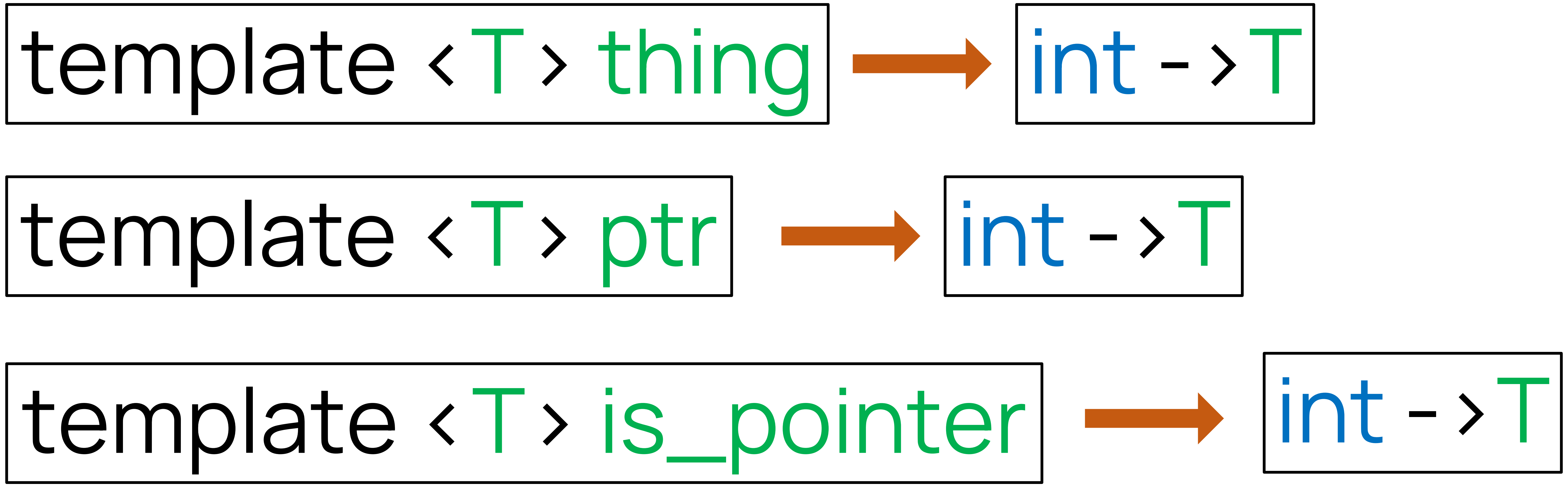
template <T> ptr → int -> T

Constraints

```
thing<int> th;
```

```
ptr<int>
```

```
is_pointer<int>
```



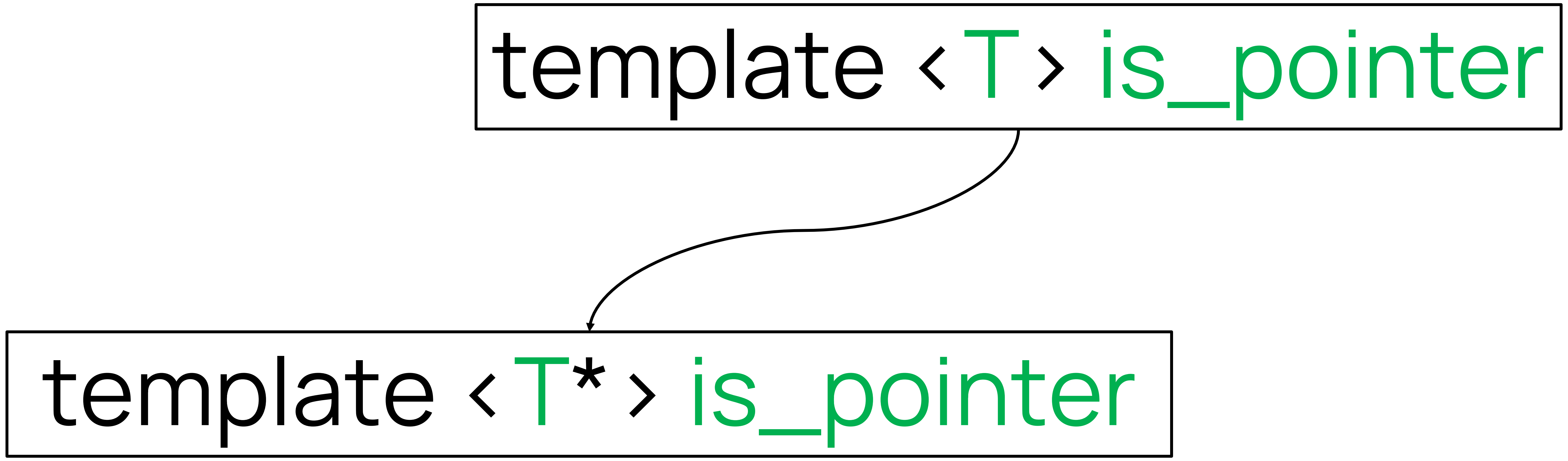
template ‹T› thing → int -› T

template ‹T› ptr → int -› T

template ‹T› is_pointer → int -› T

Constraints

```
is_pointer<int>
```

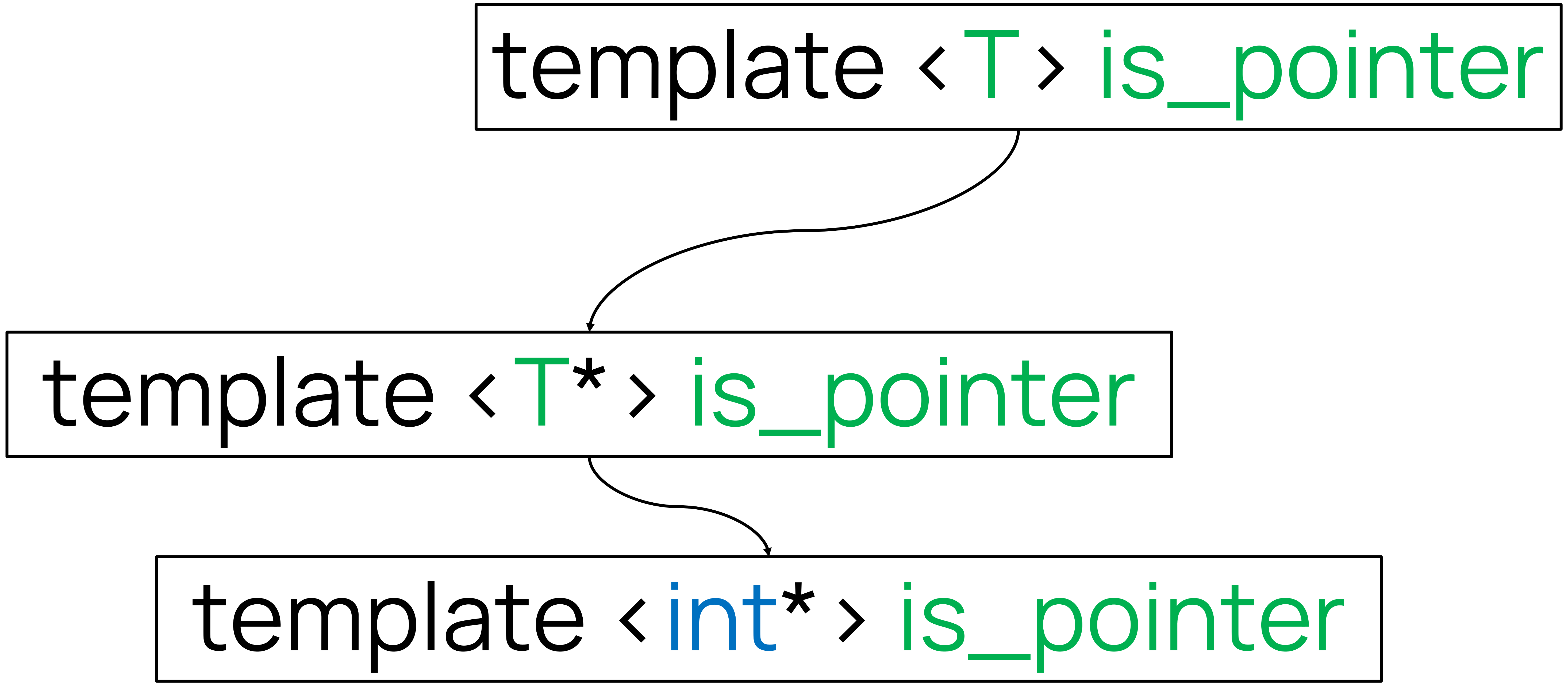


template ‹T› is_pointer

template ‹T*› is_pointer

Constraints

```
is_pointer<int>
```



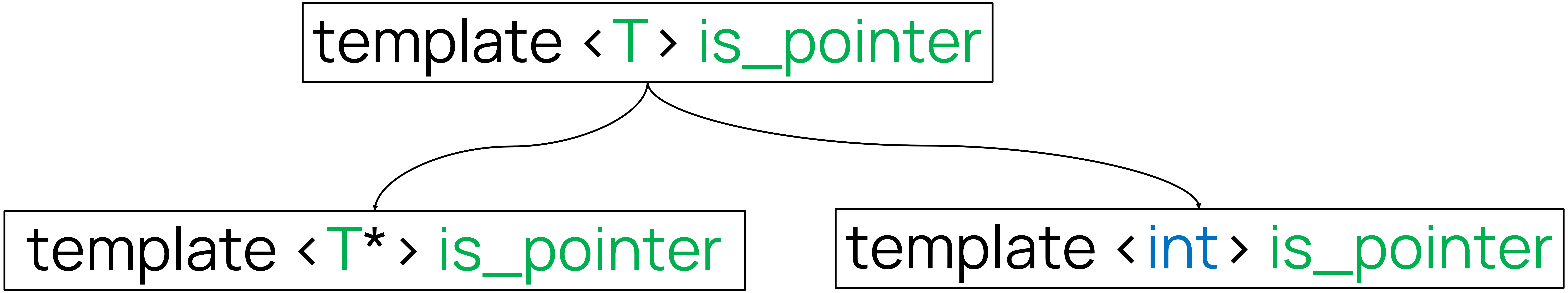
template <T> is_pointer

template <T*> is_pointer

template <int*> is_pointer

Constraints

```
is_pointer<int>
```

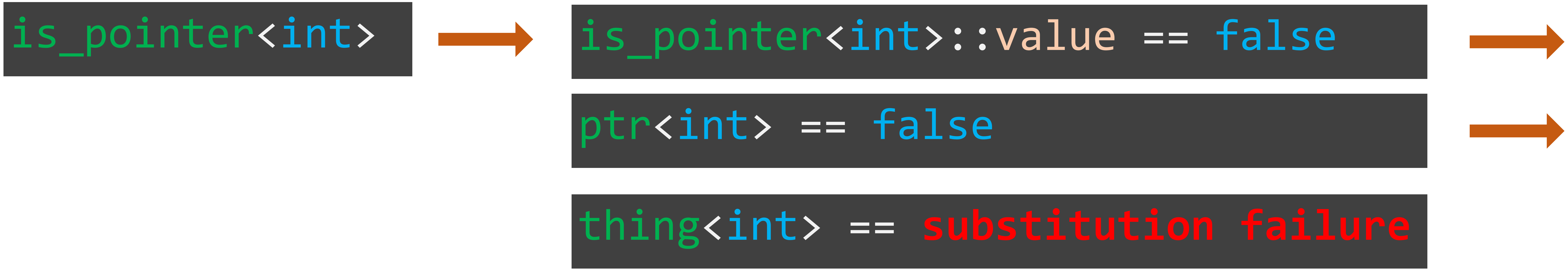


template ‹T› is_pointer

template ‹T*› is_pointer

template ‹int› is_pointer

Constraints



```
is_pointer<int>
```

```
is_pointer<int>::value == false
```

```
ptr<int> == false
```

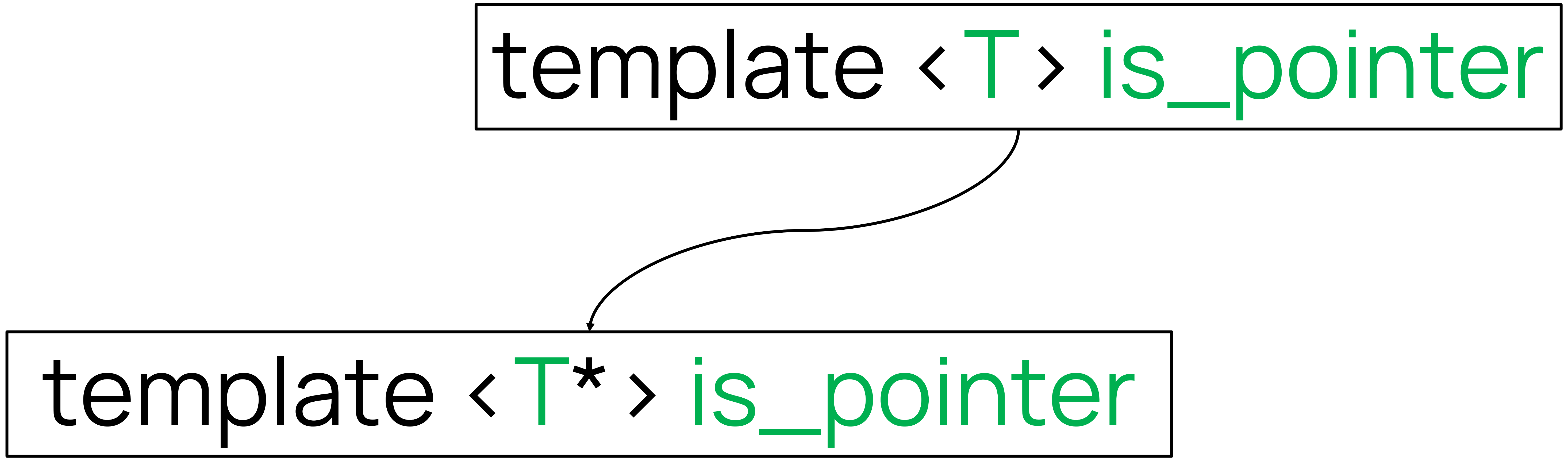
```
thing<int> == substitution failure
```

Constraints

```
is_pointer<int*>
```

Constraints

```
is_pointer<int*>
```

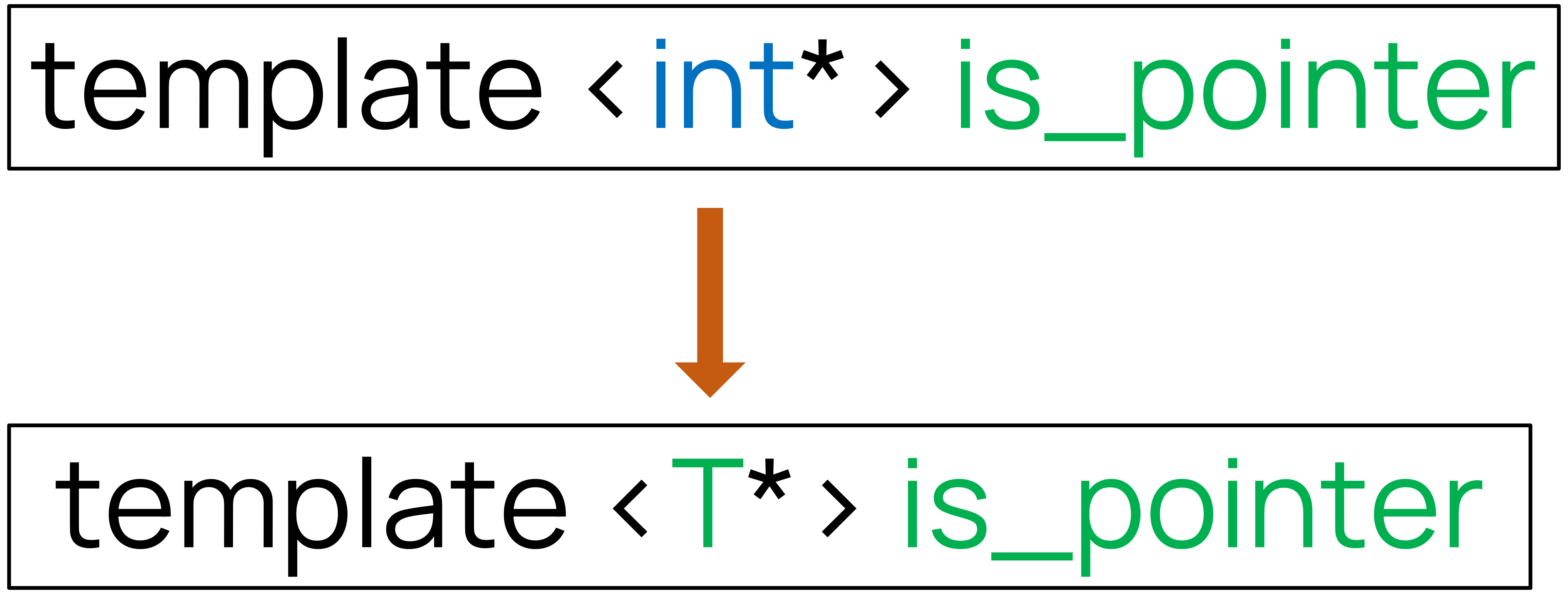


template ‹T› is_pointer

template ‹T*› is_pointer

Constraints

```
is_pointer<int*>
```

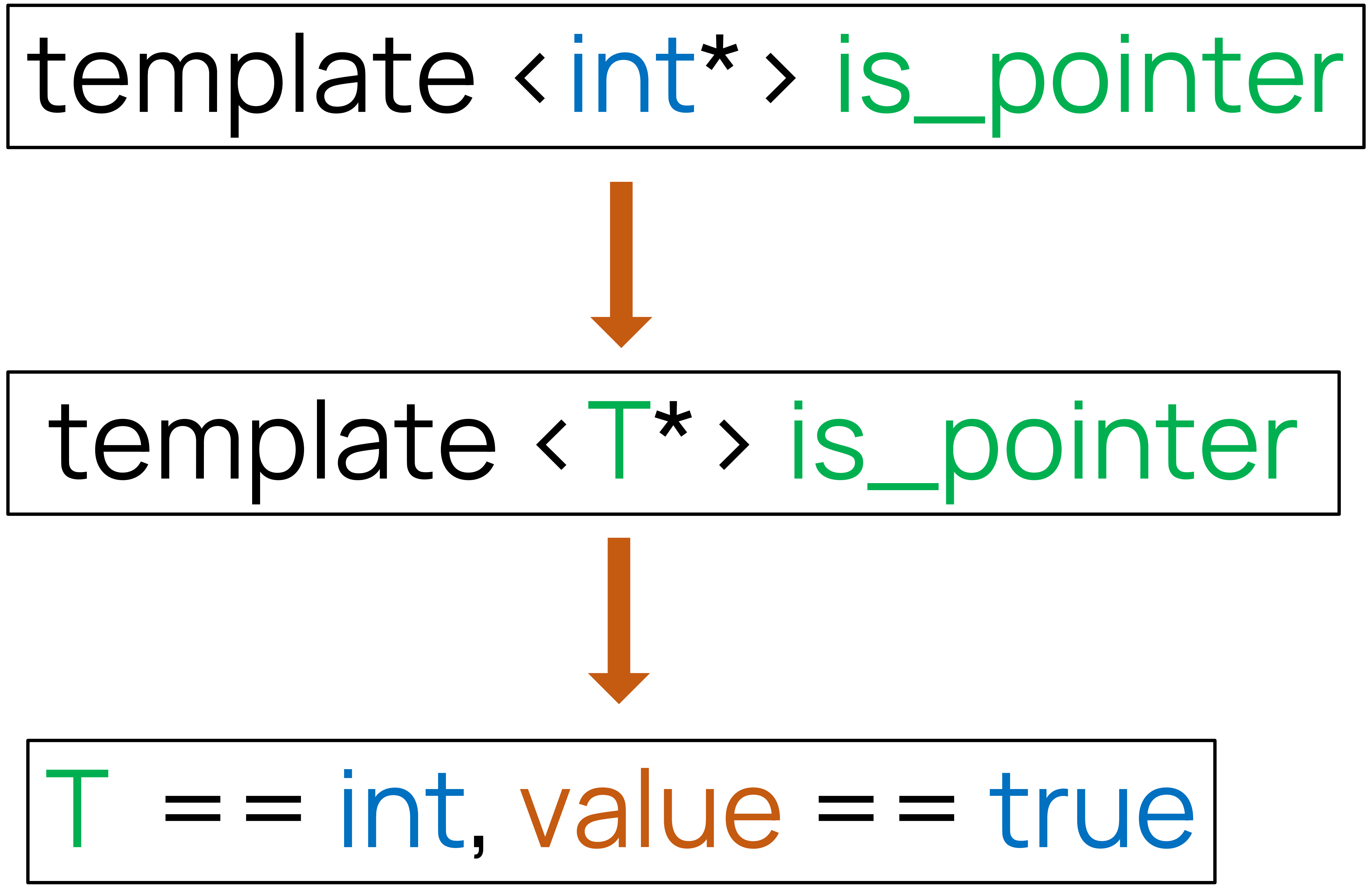


template ‹int*› is_pointer

template ‹T*› is_pointer

Constraints

```
is_pointer<int*>
```



template <int*> is_pointer

template <T*> is_pointer

T == int, value == true

Constraints

```
template <typename T> struct is_pointer
{
  static constexpr auto value = false;
};

template <typename T> struct is_pointer<T*>
{
  static constexpr auto value = true;
};
```

Why?

```
template <typename T> struct is_pointer<T*>
{
  static constexpr auto value = true;
};
```

Why?

```
template <typename T> struct is_pointer<T*>
{
  static constexpr auto value = true;
};


pretend this is what happens:

template <typename T>
void fake_function(T*);
```

Why?

```
template <typename T> struct is_pointer<T*>
{
  static constexpr auto value = true;
};

pretend this is what happens:

template <typename T>
void fake_function(T*);

// ...
int* fake_var;
fake_function(fake_var);
```

Why?

```
template <typename T> struct is_pointer<T*>
{
  static constexpr auto value = true;
};

pretend this is what happens:

template <typename T>
void fake_function(T*);

// ...
int* fake_var;
fake_function(fake_var);  --> fake_function(int*) => T == int
```

Why?

https://en.cppreference.com/w/cpp/language/partial_specialization

C++ Semantics

And the meaning of things

Q&A

PVS-Studio

Yuri Minaev

Architect