CSE340 Principles of Programming Languages

Hindley-Milner Type Checking

Automatic Type Inference

What can be inferred about type of ***f*** or ***x*** from this definition?

```
f(x) = x
```

Automatic Type Inference

```
f(x) = x
```

f is a function that takes a single argument. So the type of *f* can be described as: **`T1(*)(T2)`**

Automatic Type Inference

```
f(x) = x
```

The return value of *f* is equal to its input, so their types must match:

`T1 = T2`

Automatic Type Inference

```
f(x) = x
```

So *f* is a function that takes one argument and its return type is the same as its argument's type. Therefore type of *f* is: **`T1(*)(T1)`**

Automatic Type Inference

And we don't know anything about the type of ***x***

```
f(x) = x
```

So ***f*** is a function that takes one argument and its return type is the same as its argument's type. Therefore type of ***f*** is: **`T1(*)(T1)`**

Automatic Type Inference

How about function ***g***?

```
g(x) = x + 1
```

Automatic Type Inference

```
g(x) = x + 1
```

What can be inferred from this term?

Automatic Type Inference

```
g(x) = x + 1
```

x is used in an arithmetic expression involving the integer constant 1. So ***x*** must be of integer type

Automatic Type Inference

```
g(x) = x + 1
```

So the type of function ***g*** should be further restricted to: **`int(*)(int)`**

To perform Hindley-Milner type checking:

- Start by generating the abstract syntax tree of the function
- Assume unknown types for arguments: T1, T2, …
- Examine the tree nodes and apply type constraints to further restrict the types
- The type constraints that can be applied depend on the programming language used. In the following examples we use simple rules similar to those in functional languages like OCaml

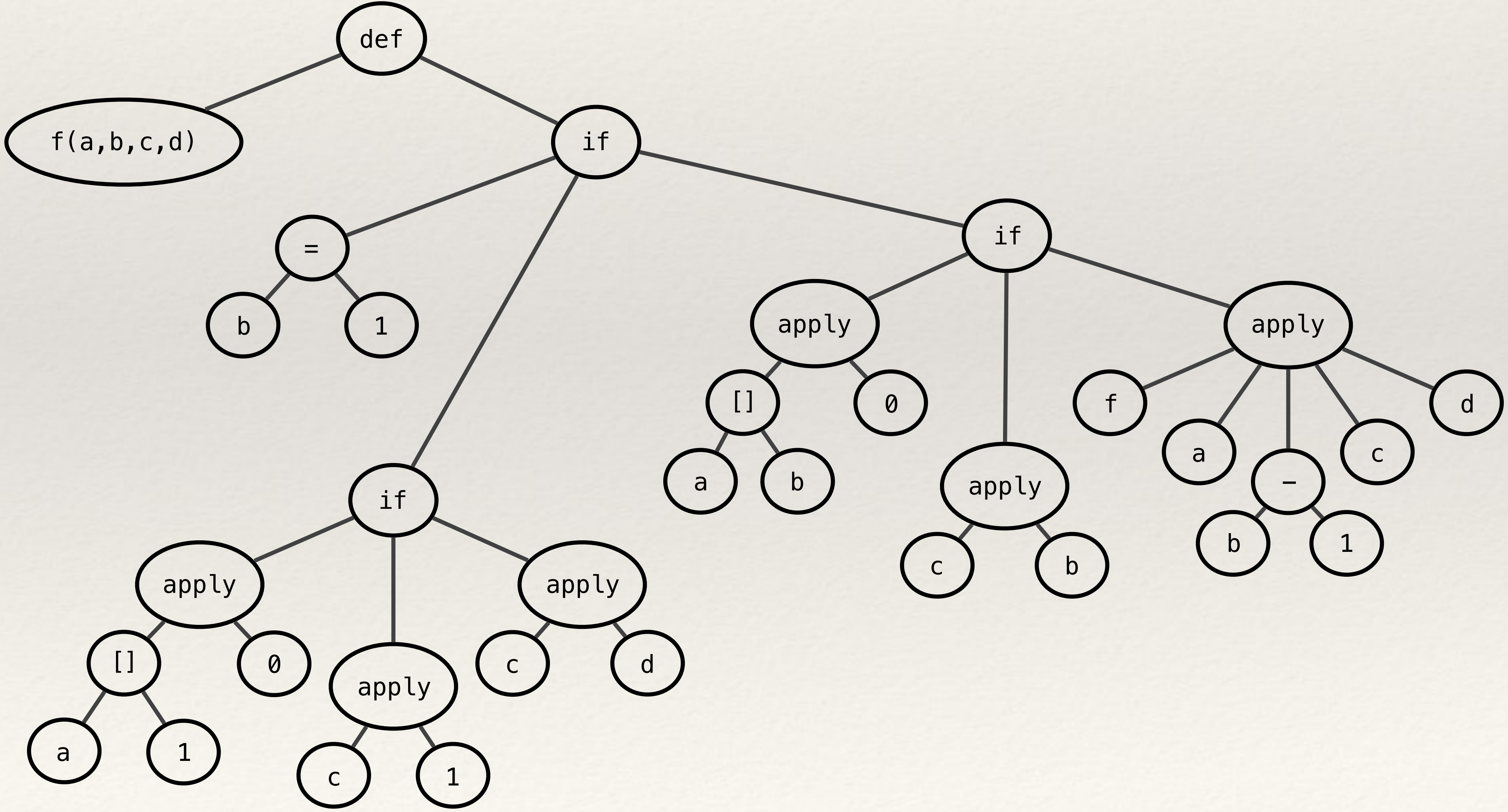
Examples

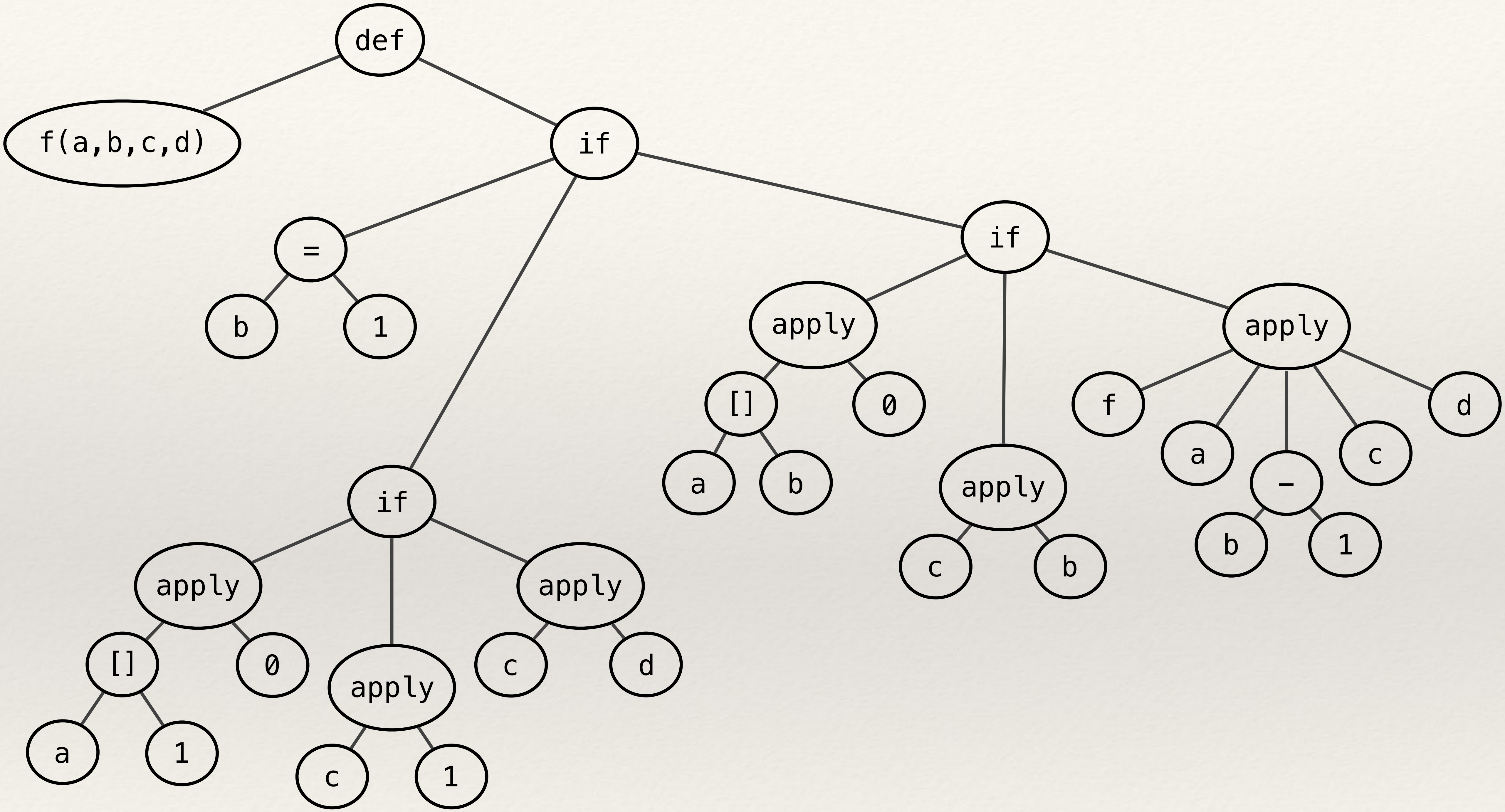
Example #1

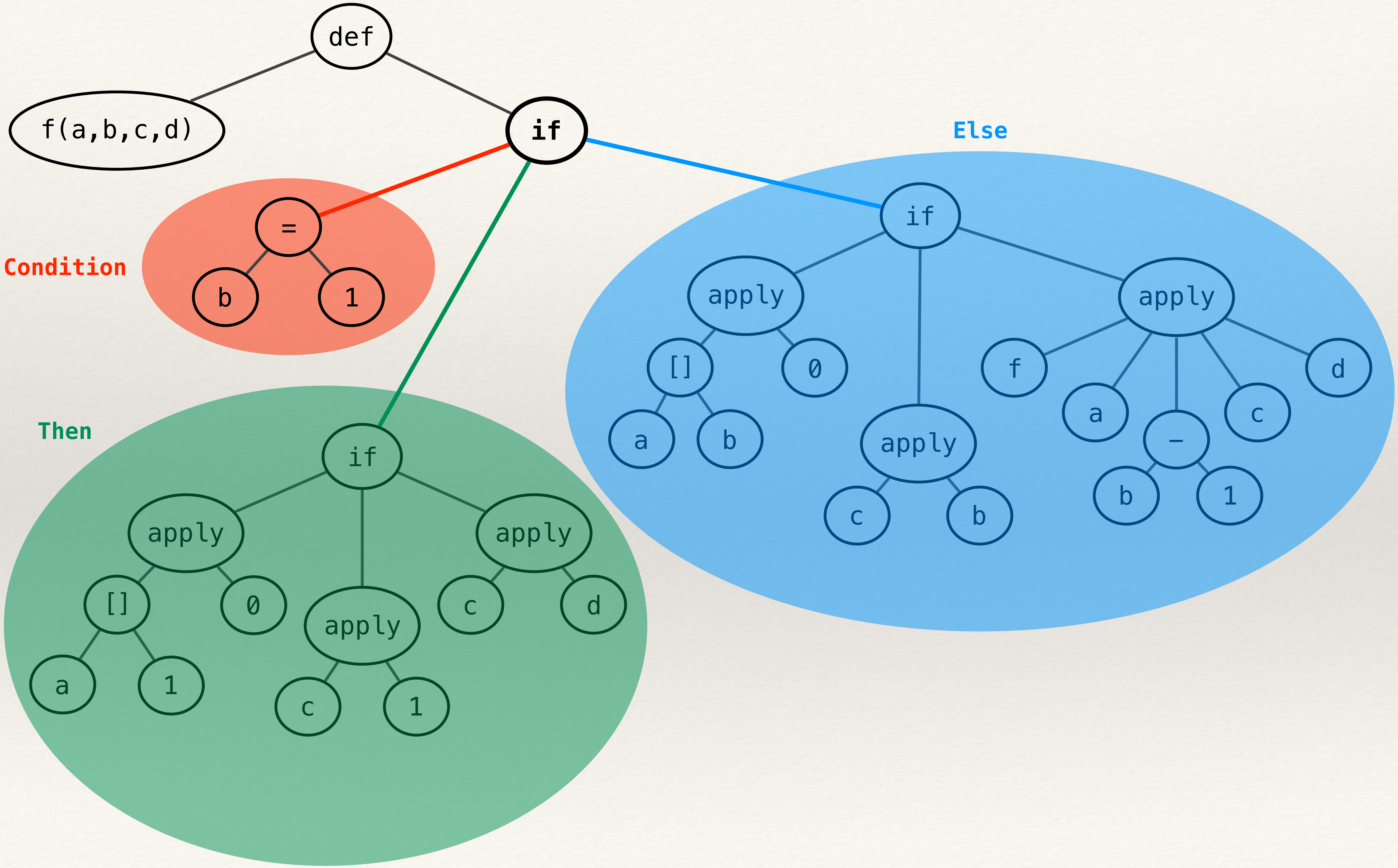
```
f(a,b,c,d) = if b = 1 then
                 if a[1](0) then
                     c(1)
                 else
                     c(d)
             else
                 if a[b](0) then
                     c(b)
                 else
                     f(a, b – 1, c, d)
```

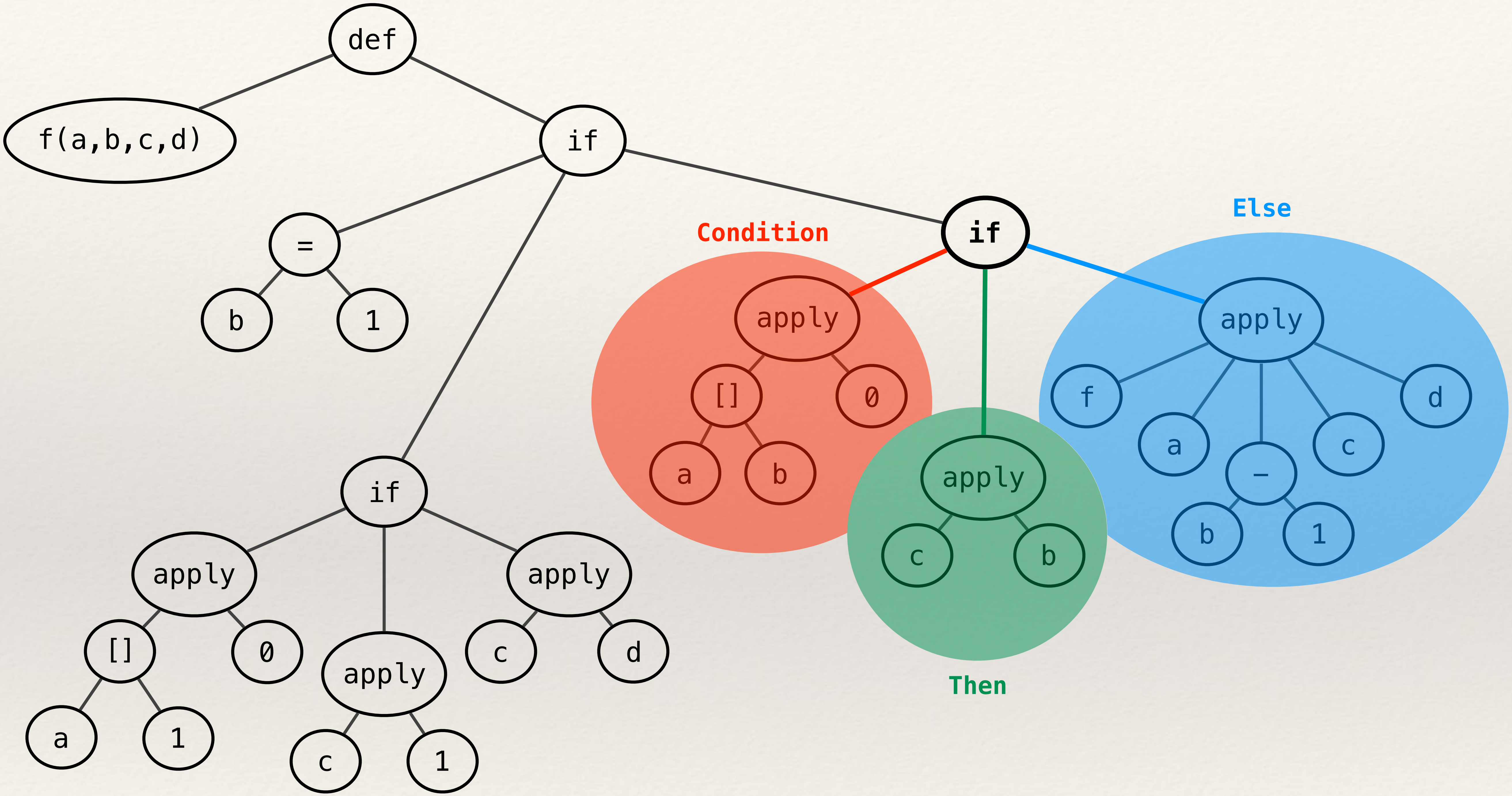
Example #1

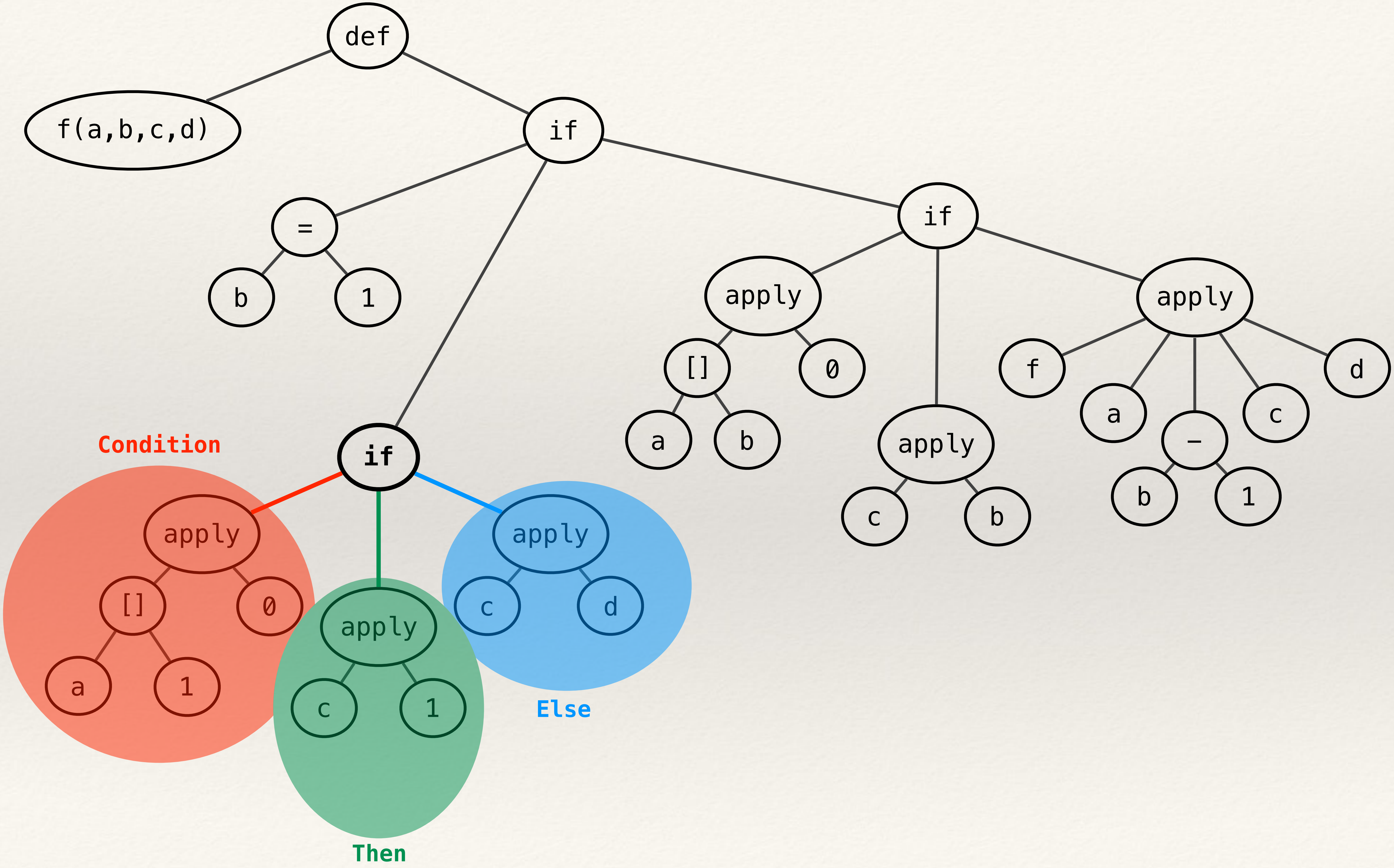
```
f(a,b,c,d) = if b = 1 then
                 if a[1](0) then
                     c(1)
                 else
                     c(d)
             else
                 if a[b](0) then
                     c(b)
                 else
                     f(a, b - 1, c, d)
```

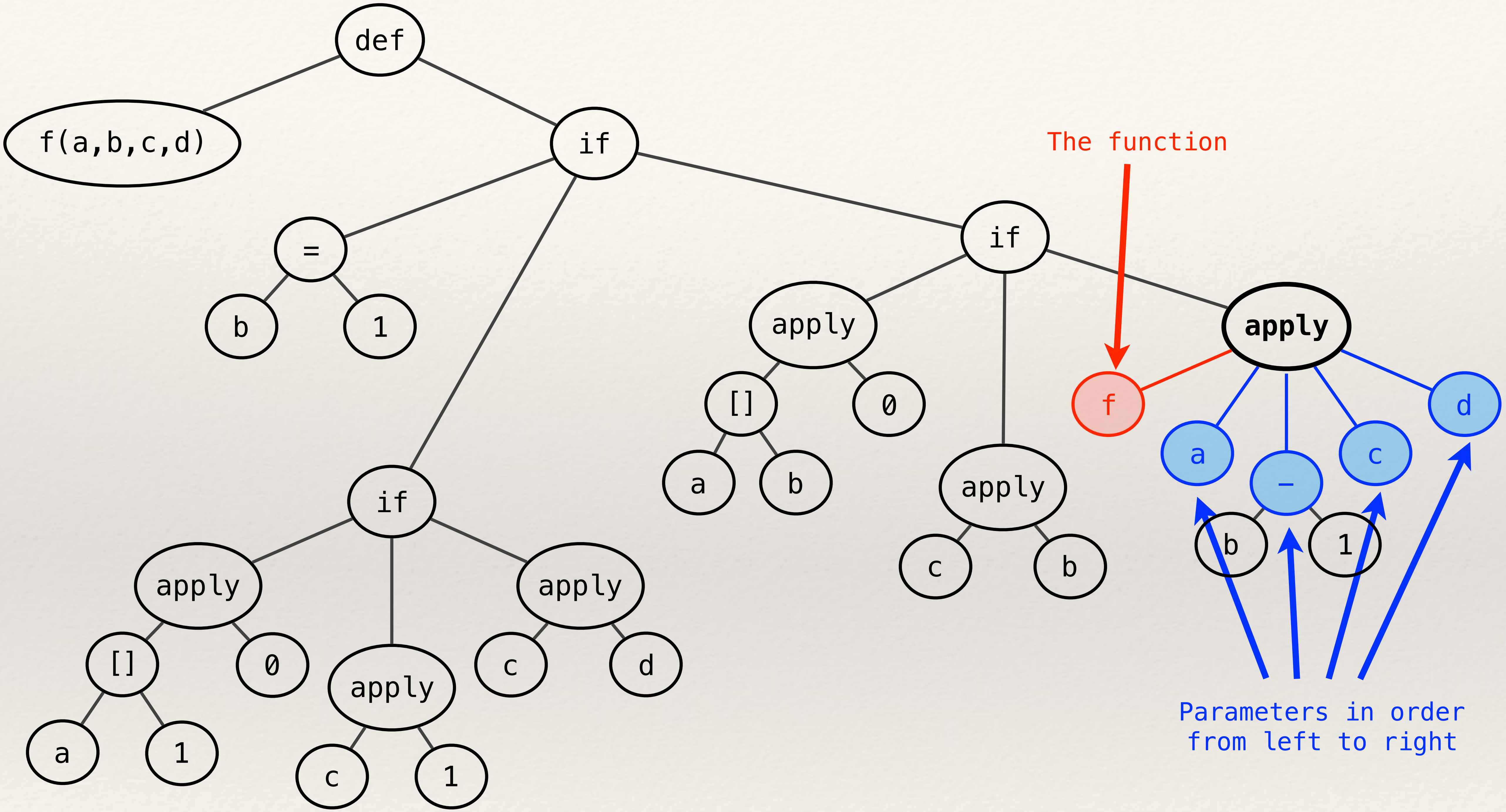



def
f(a,b,c,d)
if
=
b
1
if
apply
[]
a
b
0
apply
c
b
apply
f
a
-
b
1
c
d
if
apply
[]
a
1
0
apply
c
1
apply
c
d


def
f(a,b,c,d)
if
Else
Condition
=
b
1
if
apply
apply
[]
0
f
d
a
b
a
c
apply
–
c
b
b
1
Then
if
apply
apply
[]
0
apply
c
d
a
1
c
1

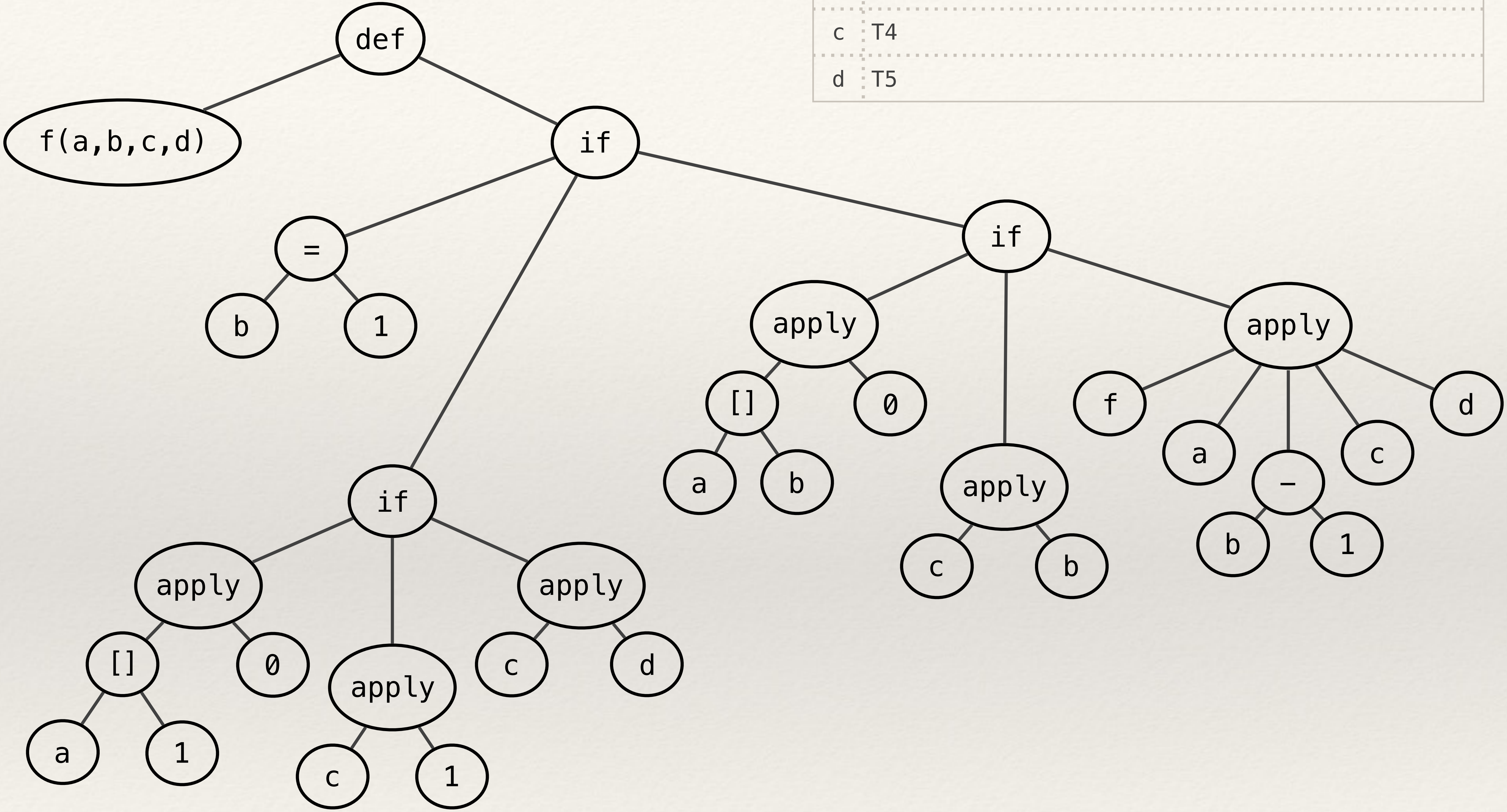

def
f(a,b,c,d)
if
=
b
1
if
Condition
Else
apply
[]
0
a
b
apply
c
b
Then
apply
f
a
-
c
d
b
1
if
apply
[]
0
a
1
apply
c
1
apply
c
d


def
f(a,b,c,d)
if
=
b
1
if
apply
[]
a
b
0
apply
c
b
apply
f
a
–
b
1
c
d
Condition
if
apply
[]
a
1
0
apply
c
1
Then
apply
c
d
Else



The function

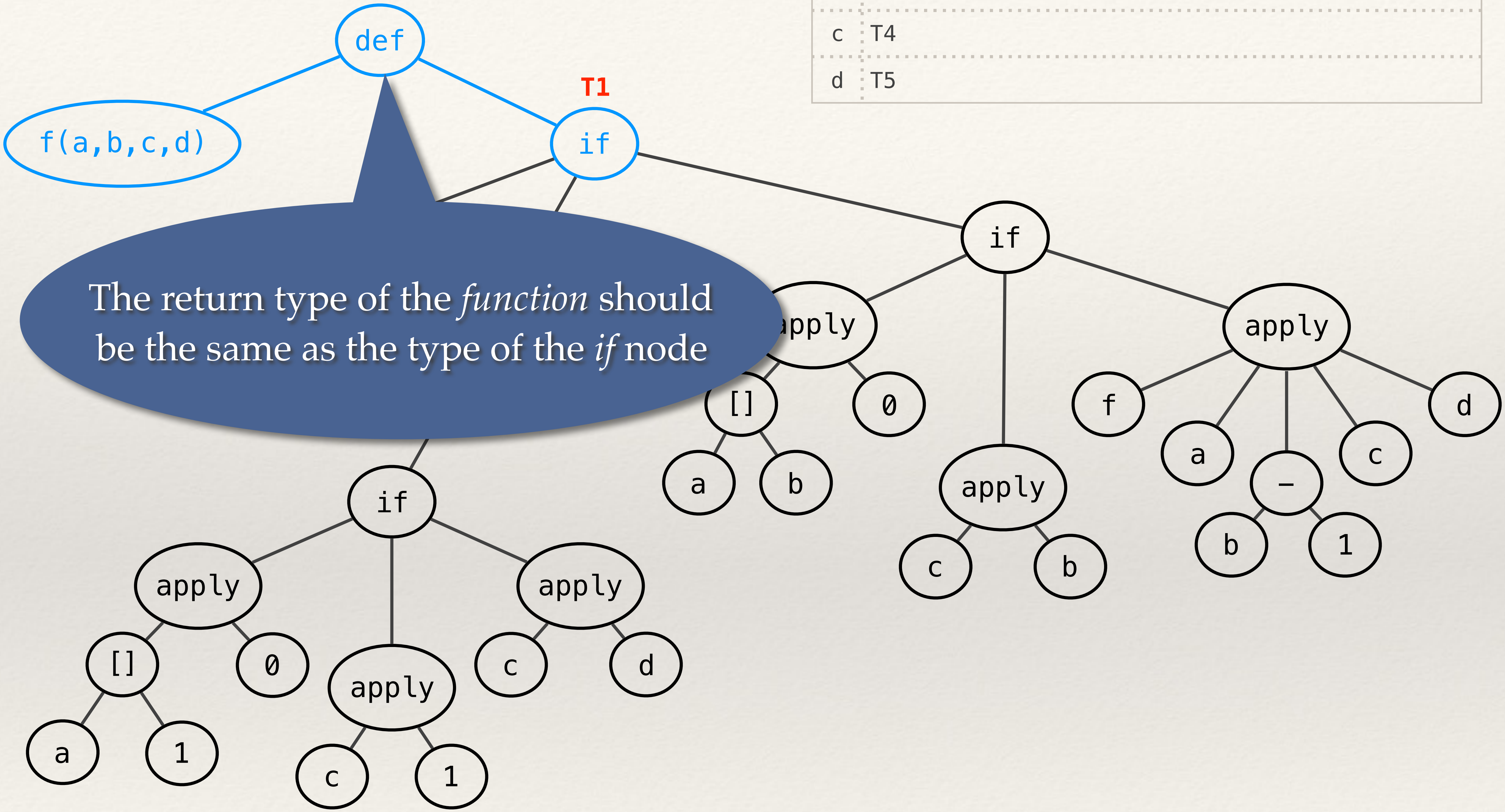
Parameters in order
from left to right



f	T1(*)(T2,T3,T4,T5)
a	T2
b	T3
c	T4
d	T5

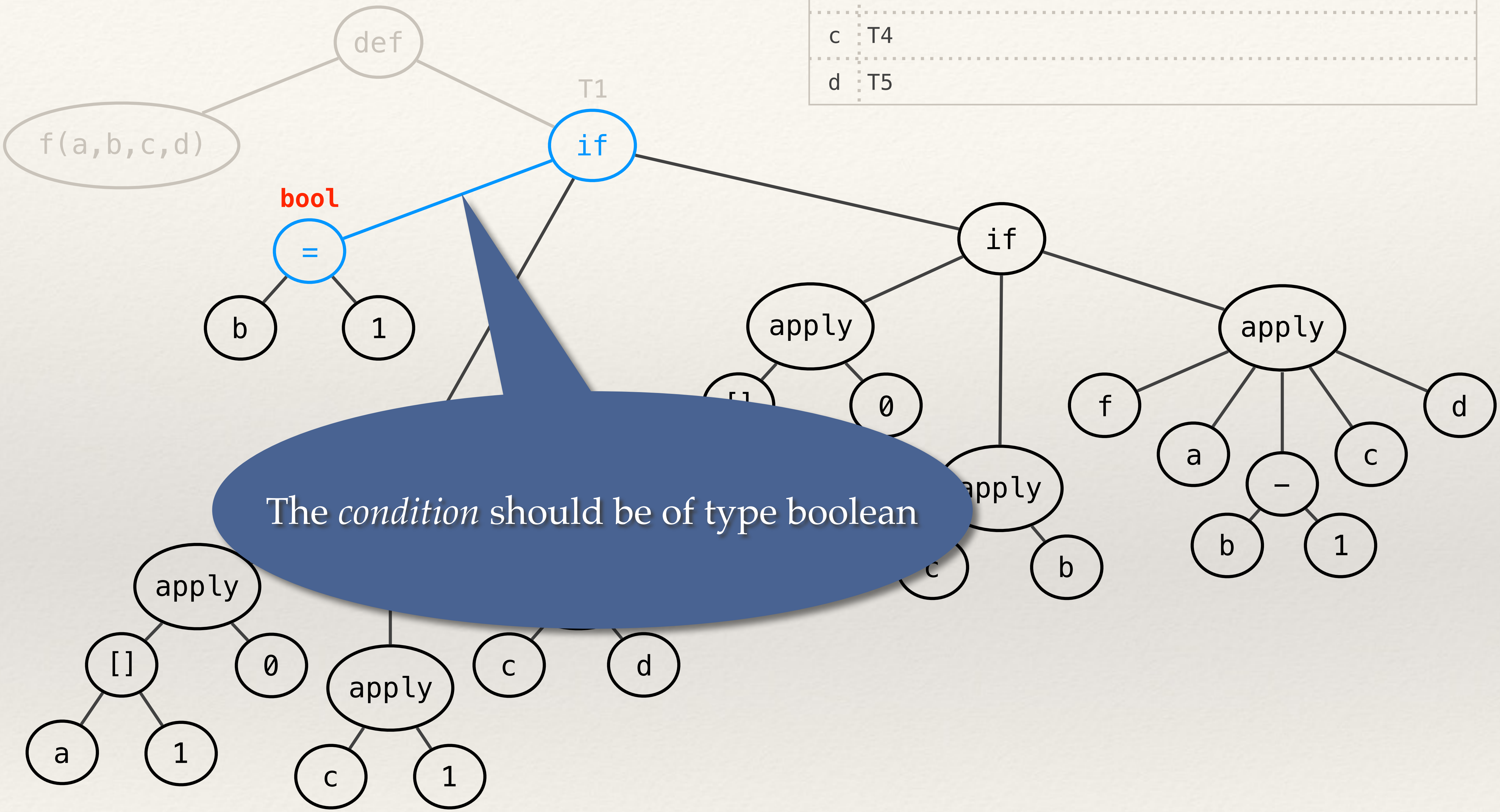
Top-Down order

f	T1(*)(T2,T3,T4,T5)
a	T2
b	T3
c	T4
d	T5



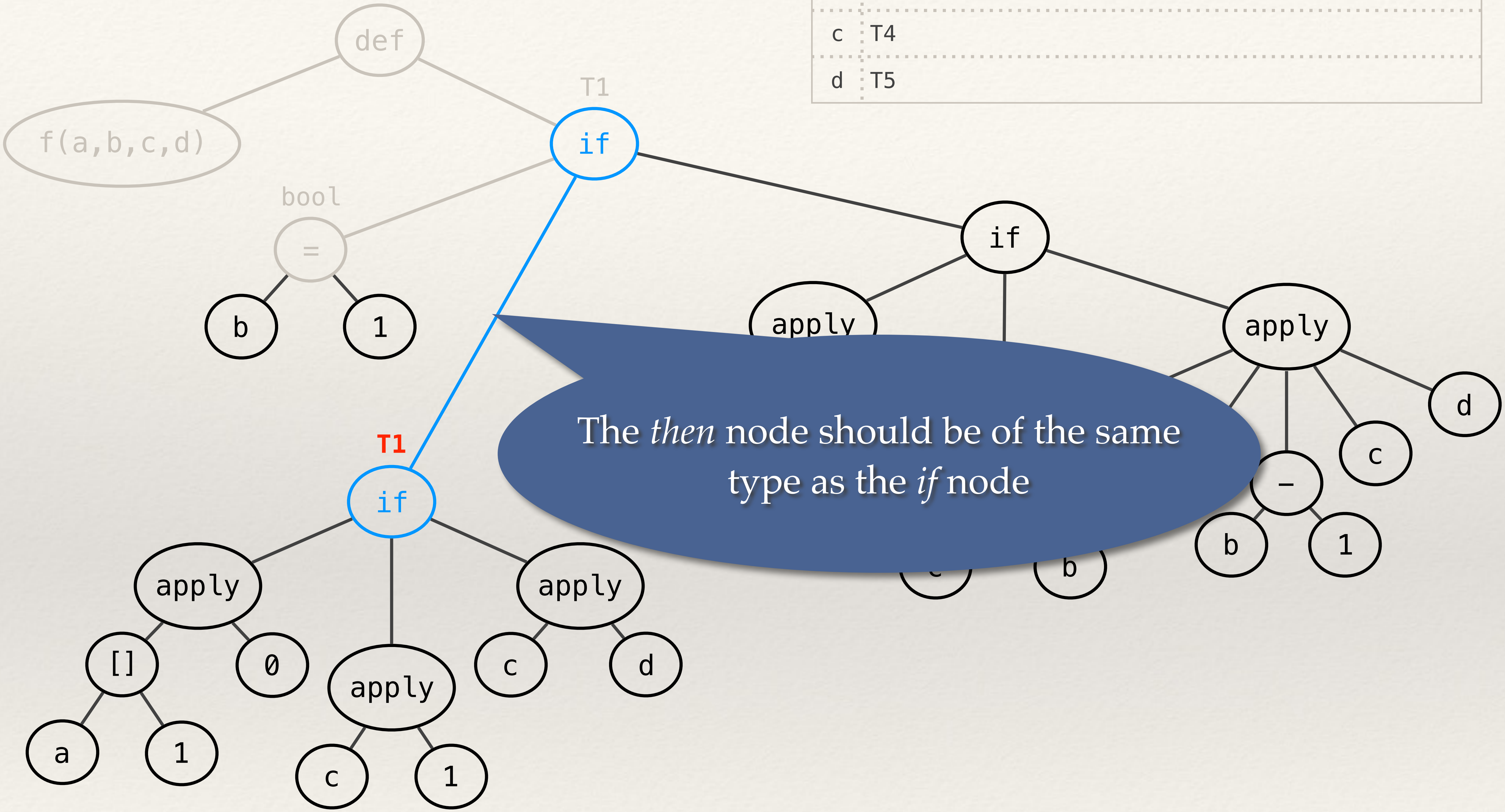
The return type of the *function* should be the same as the type of the *if* node

f	T1(*)(T2,T3,T4,T5)
a	T2
b	T3
c	T4
d	T5



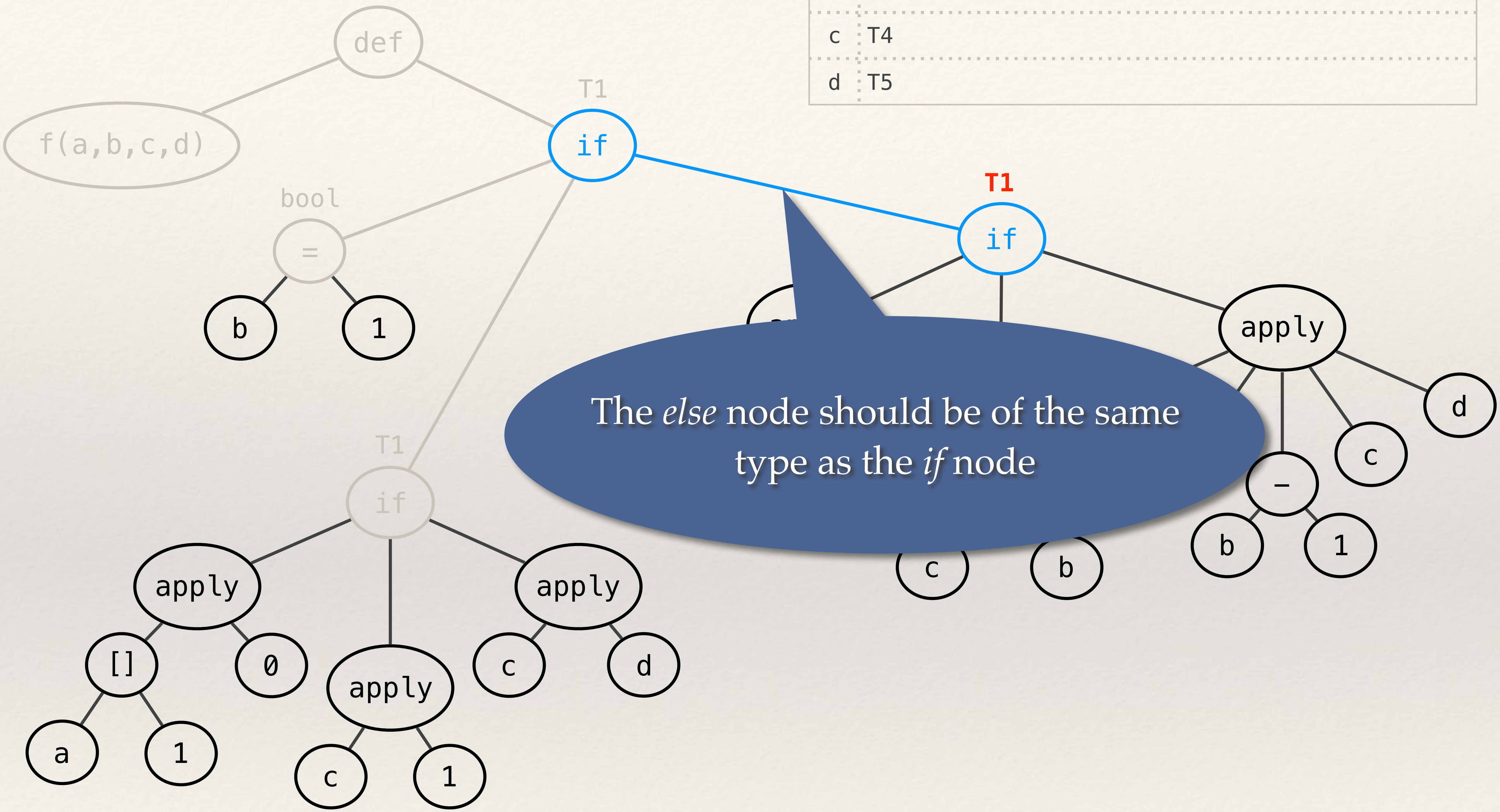
The *condition* should be of type boolean

f	T1(*)(T2,T3,T4,T5)
a	T2
b	T3
c	T4
d	T5



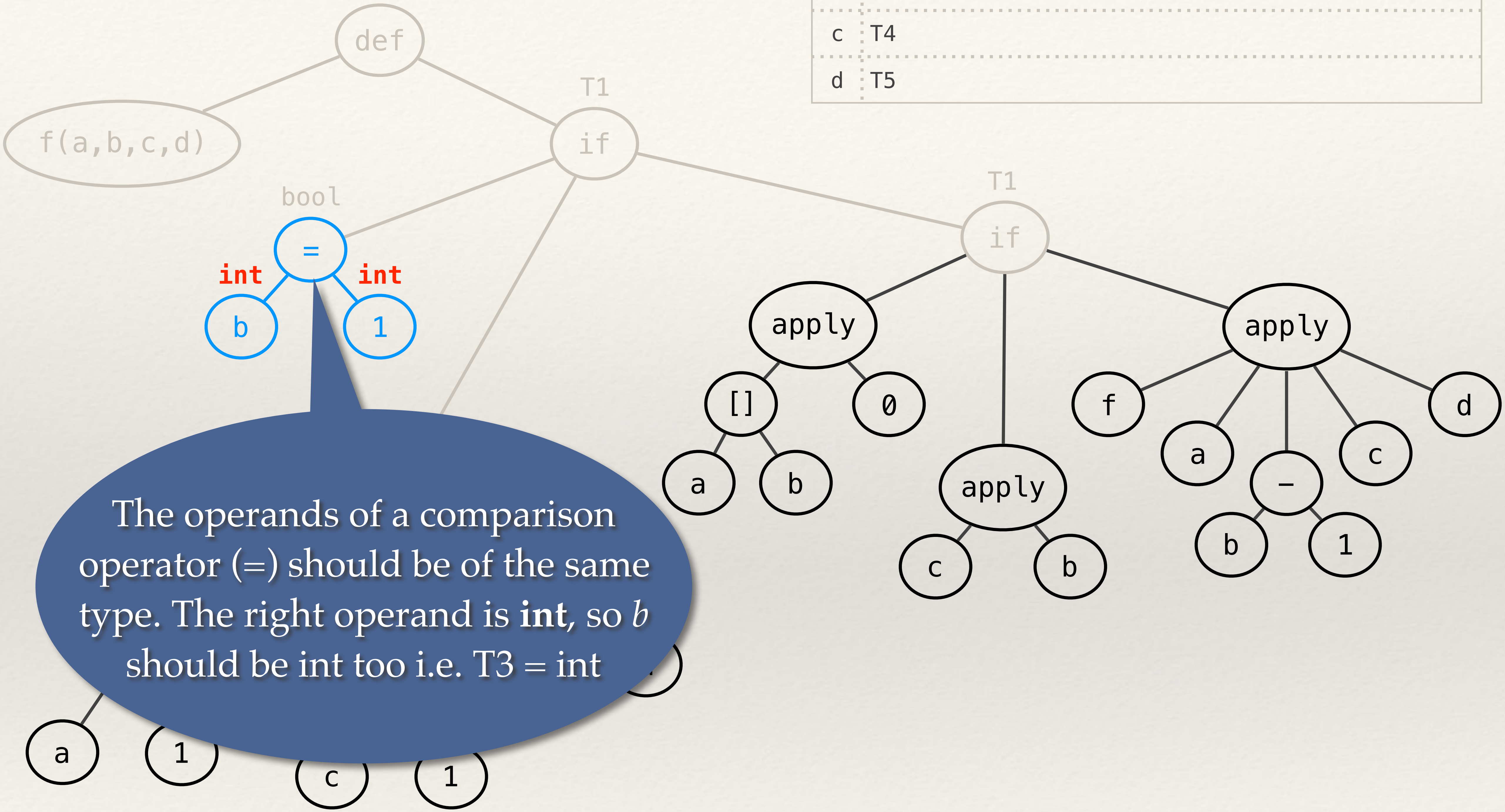
The *then* node should be of the same type as the *if* node

f	T1(*)(T2,T3,T4,T5)
a	T2
b	T3
c	T4
d	T5



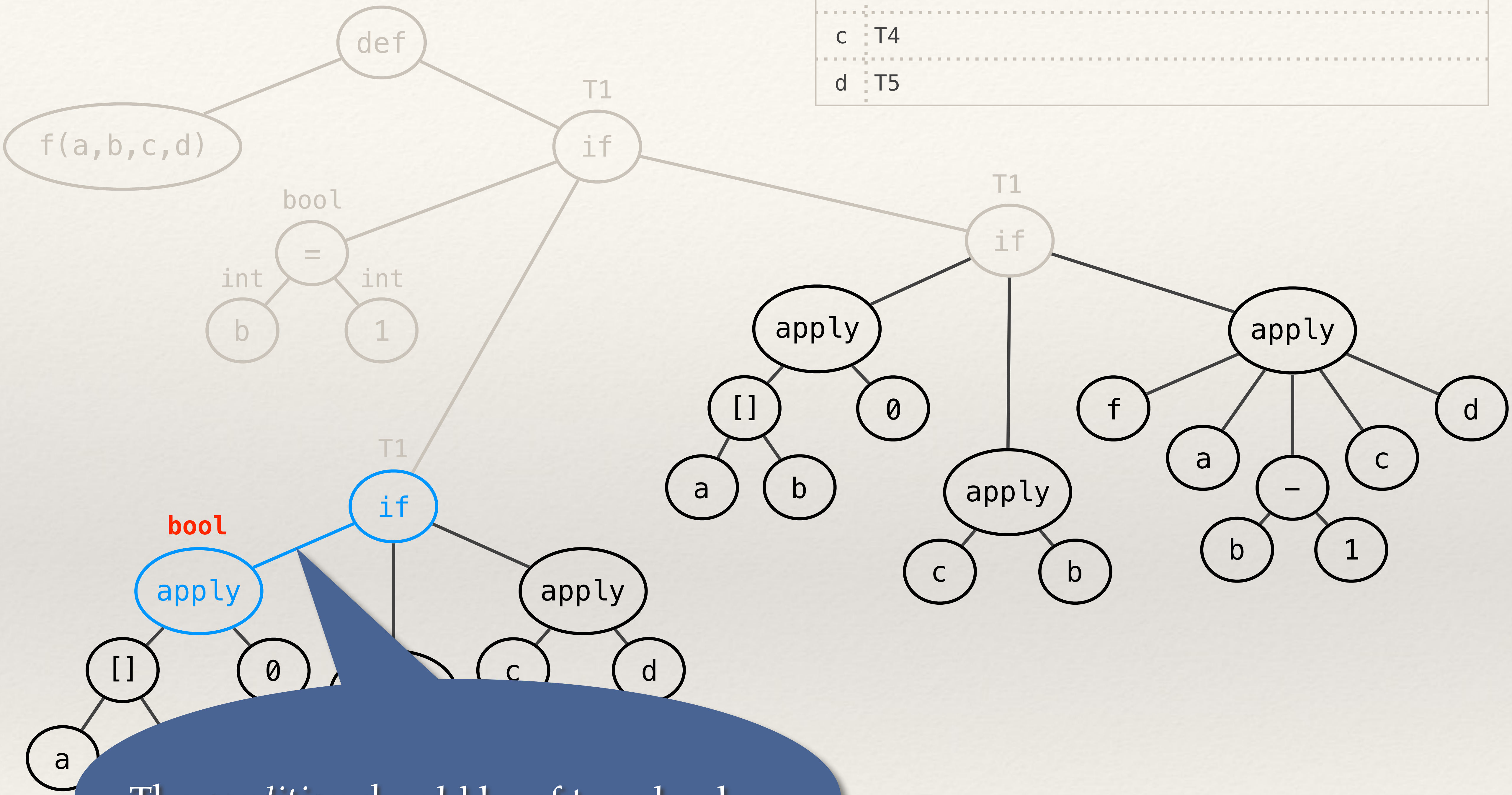
The *else* node should be of the same type as the *if* node

f	T1(*)(T2,**int**,T4,T5)
a	T2
b	**int**
c	T4
d	T5



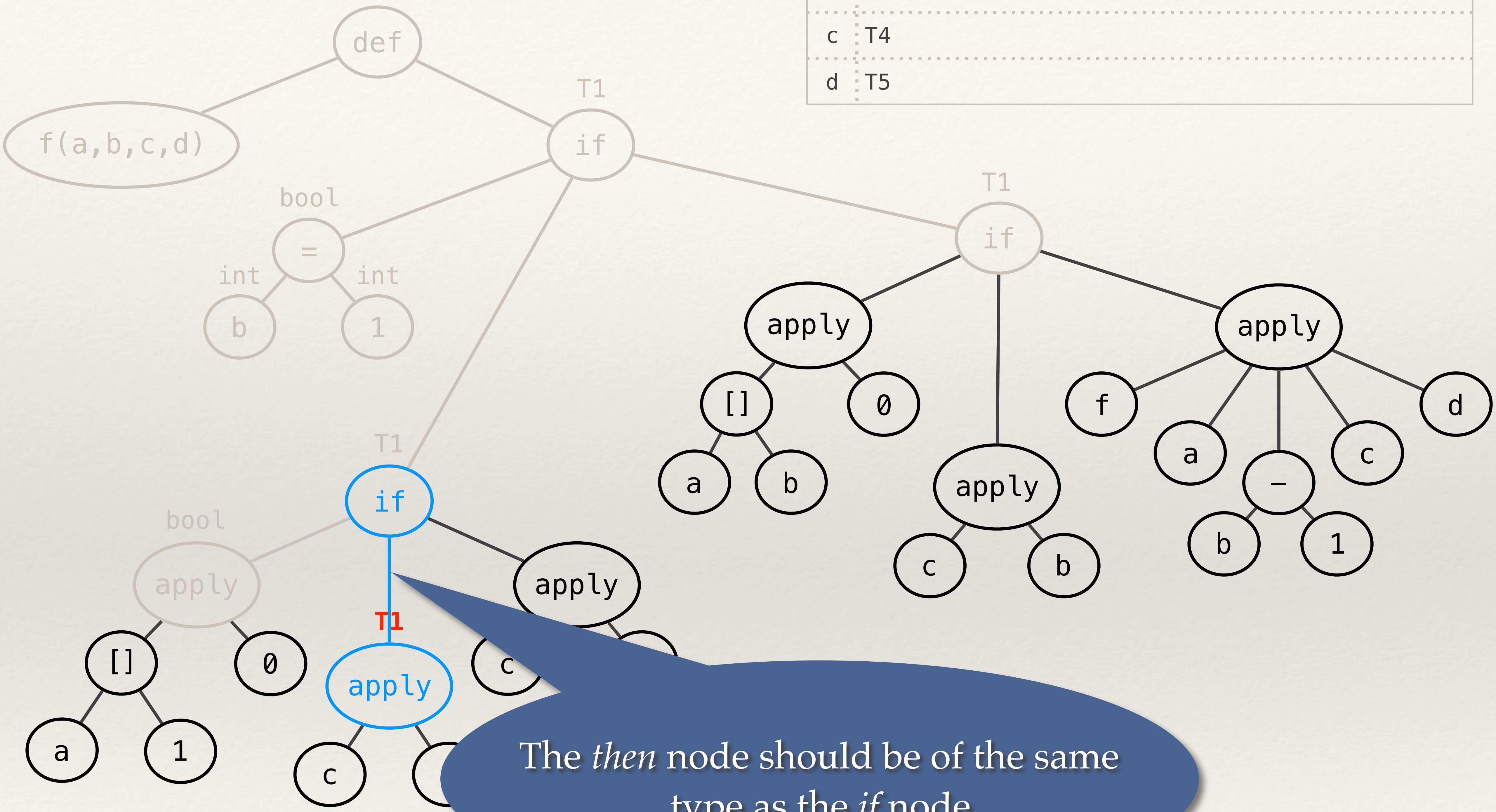
The operands of a comparison operator (=) should be of the same type. The right operand is **int**, so *b* should be int too i.e. T3 = int

f	T1(*)(T2,int,T4,T5)
a	T2
b	int
c	T4
d	T5



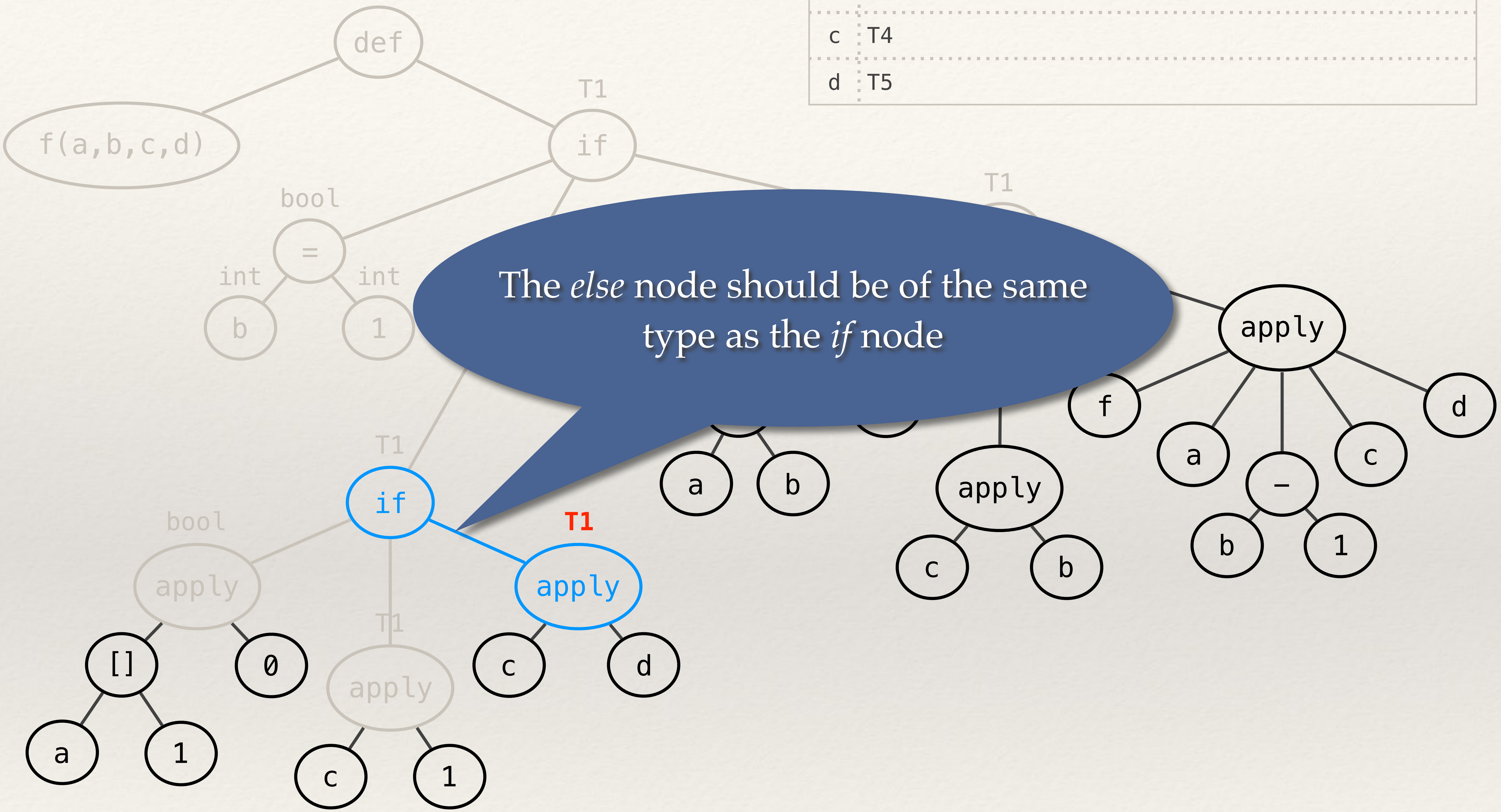
The *condition* should be of type boolean

f	T1(*)(T2,int,T4,T5)
a	T2
b	int
c	T4
d	T5

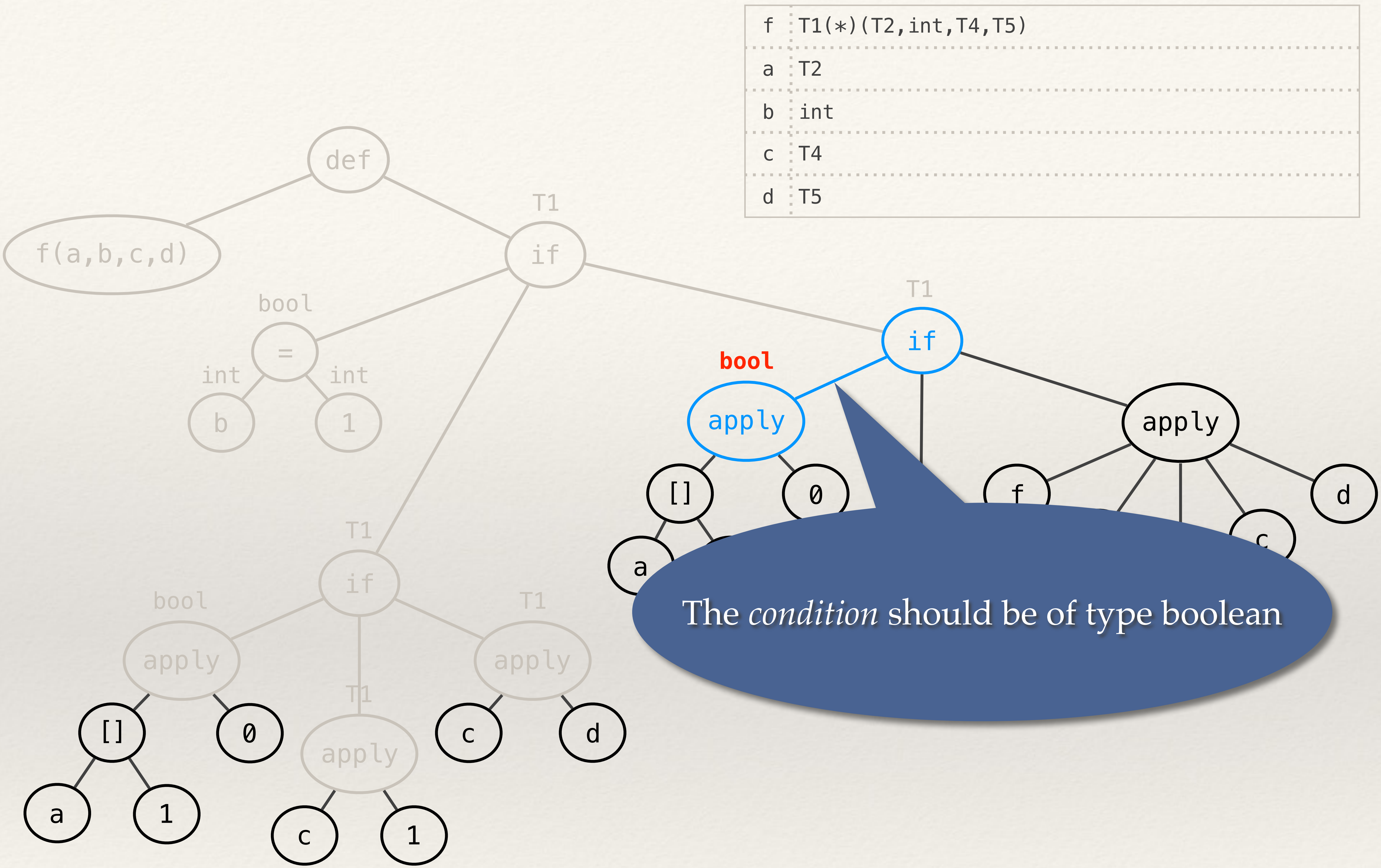


The *then* node should be of the same type as the *if* node

f	T1(*)(T2,int,T4,T5)
a	T2
b	int
c	T4
d	T5



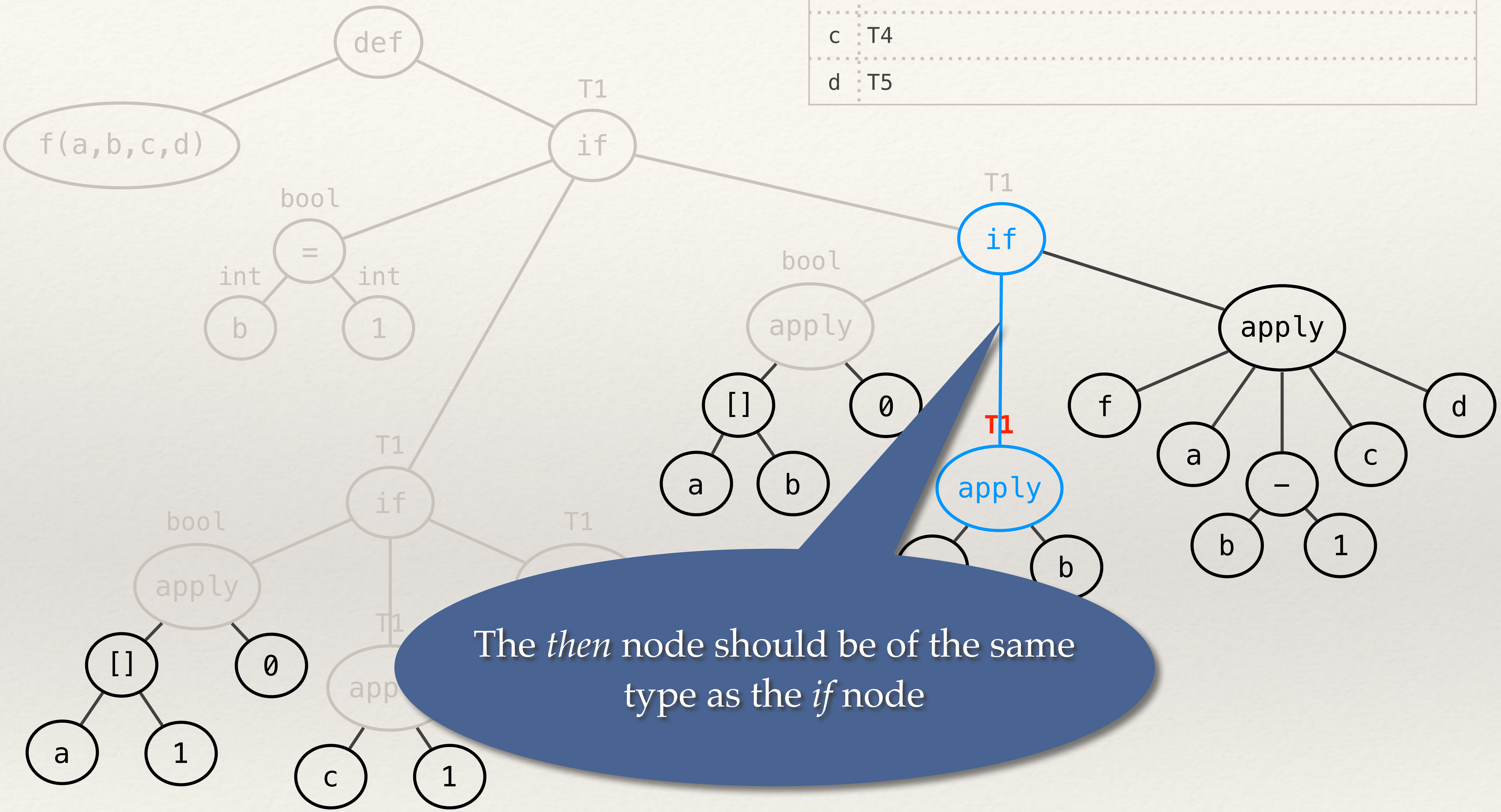
The *else* node should be of the same type as the *if* node



f	T1(∗)(T2,int,T4,T5)
a	T2
b	int
c	T4
d	T5

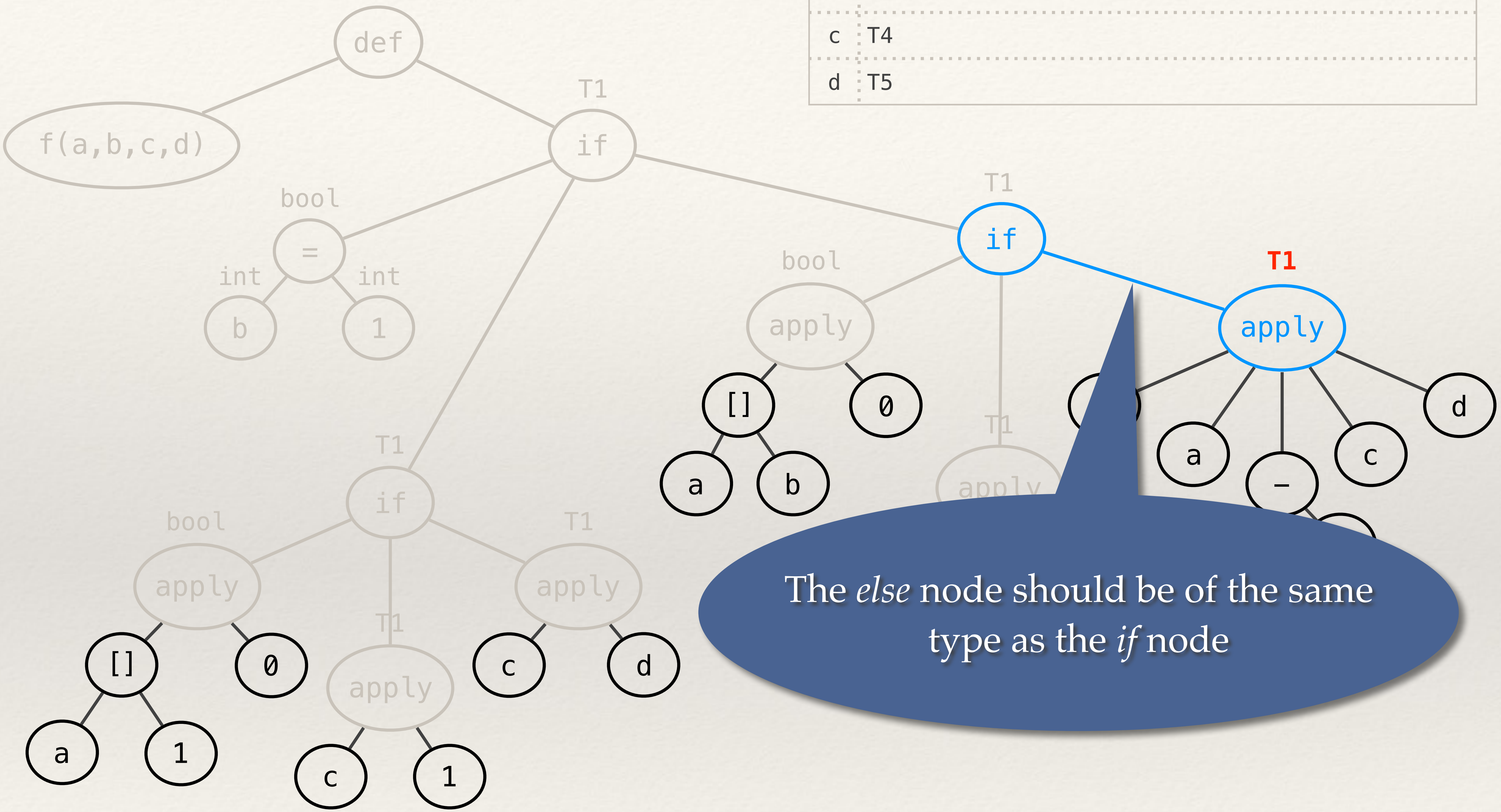
The *condition* should be of type boolean

f	T1(*)(T2,int,T4,T5)
a	T2
b	int
c	T4
d	T5



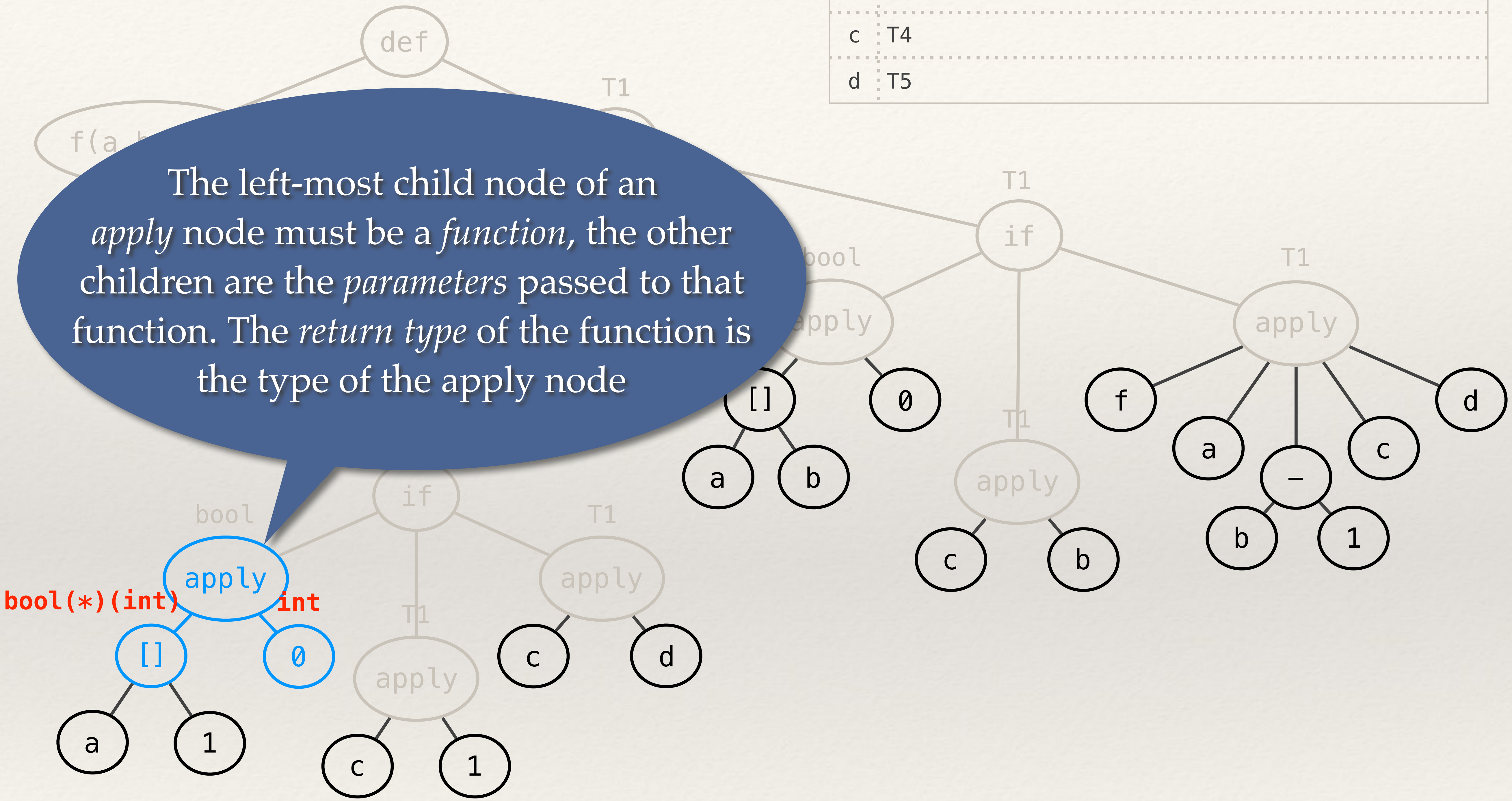
The *then* node should be of the same type as the *if* node

f	T1(*)(T2,int,T4,T5)
a	T2
b	int
c	T4
d	T5



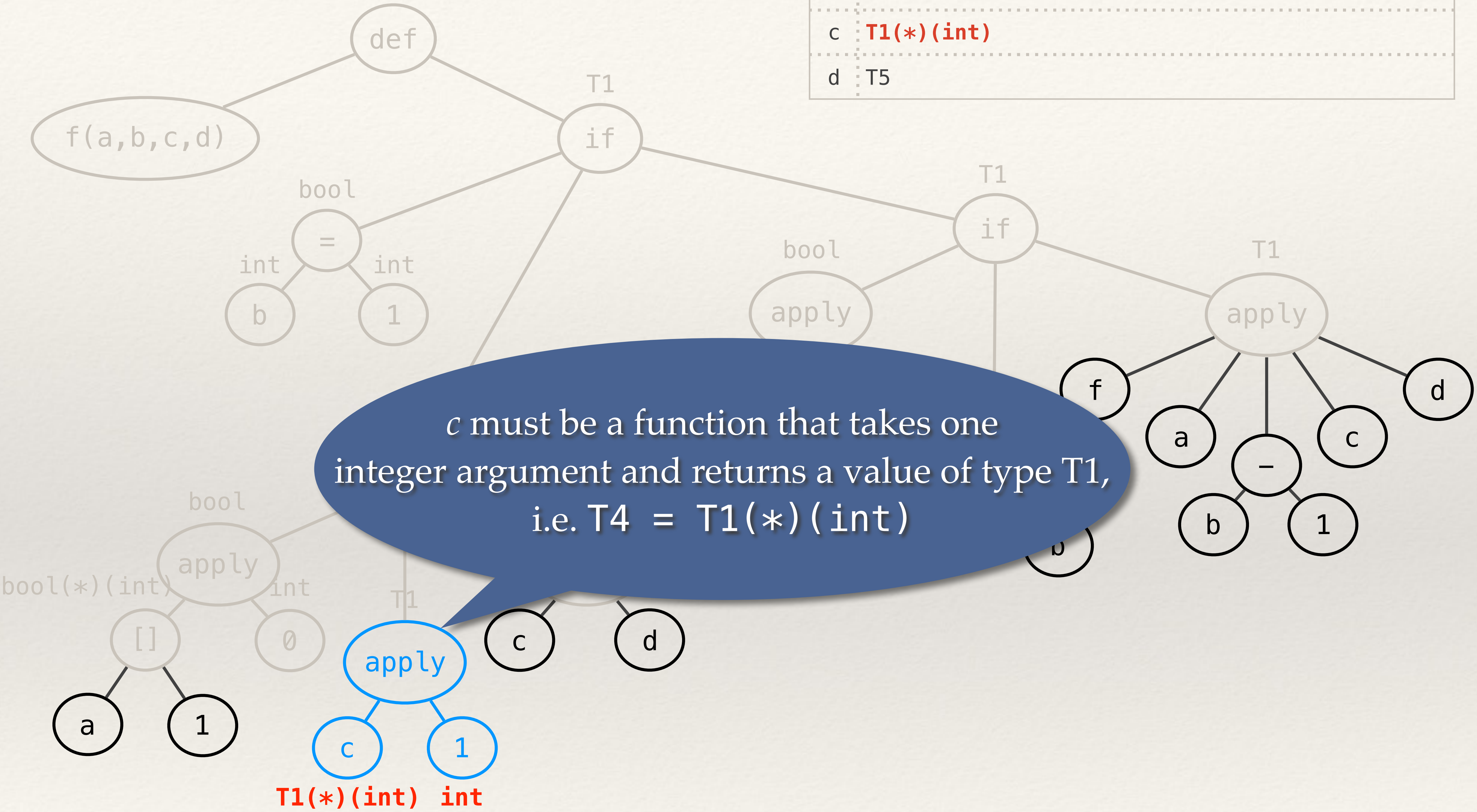
The *else* node should be of the same type as the *if* node

f	T1(*)(T2,int,T4,T5)
a	T2
b	int
c	T4
d	T5



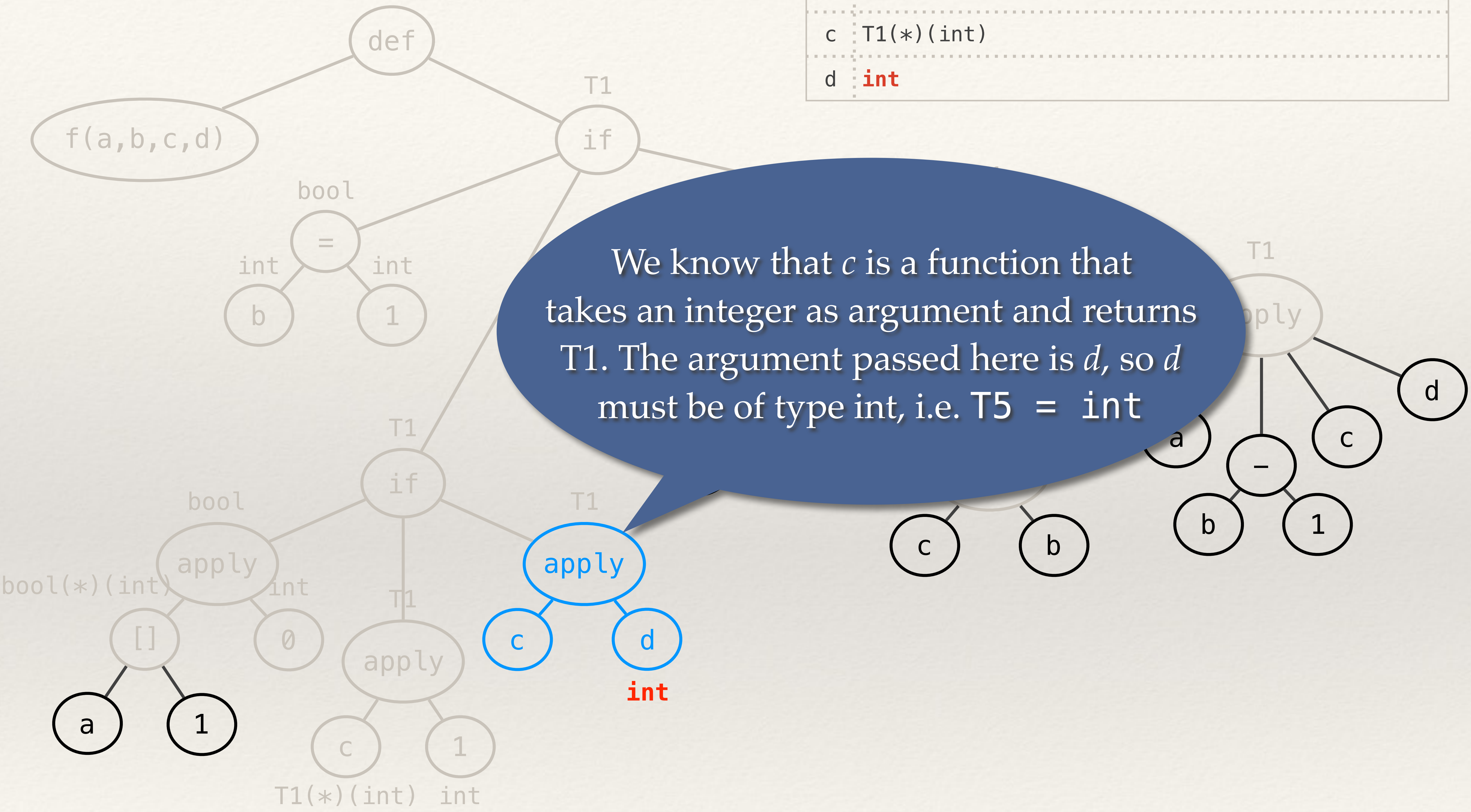
The left-most child node of an *apply* node must be a *function*, the other children are the *parameters* passed to that function. The *return type* of the function is the type of the apply node

f	T1(∗)(T2,int,**T1(∗)(int)**,T5)
a	T2
b	int
c	**T1(∗)(int)**
d	T5



c must be a function that takes one integer argument and returns a value of type T1, i.e. `T4 = T1(∗)(int)`

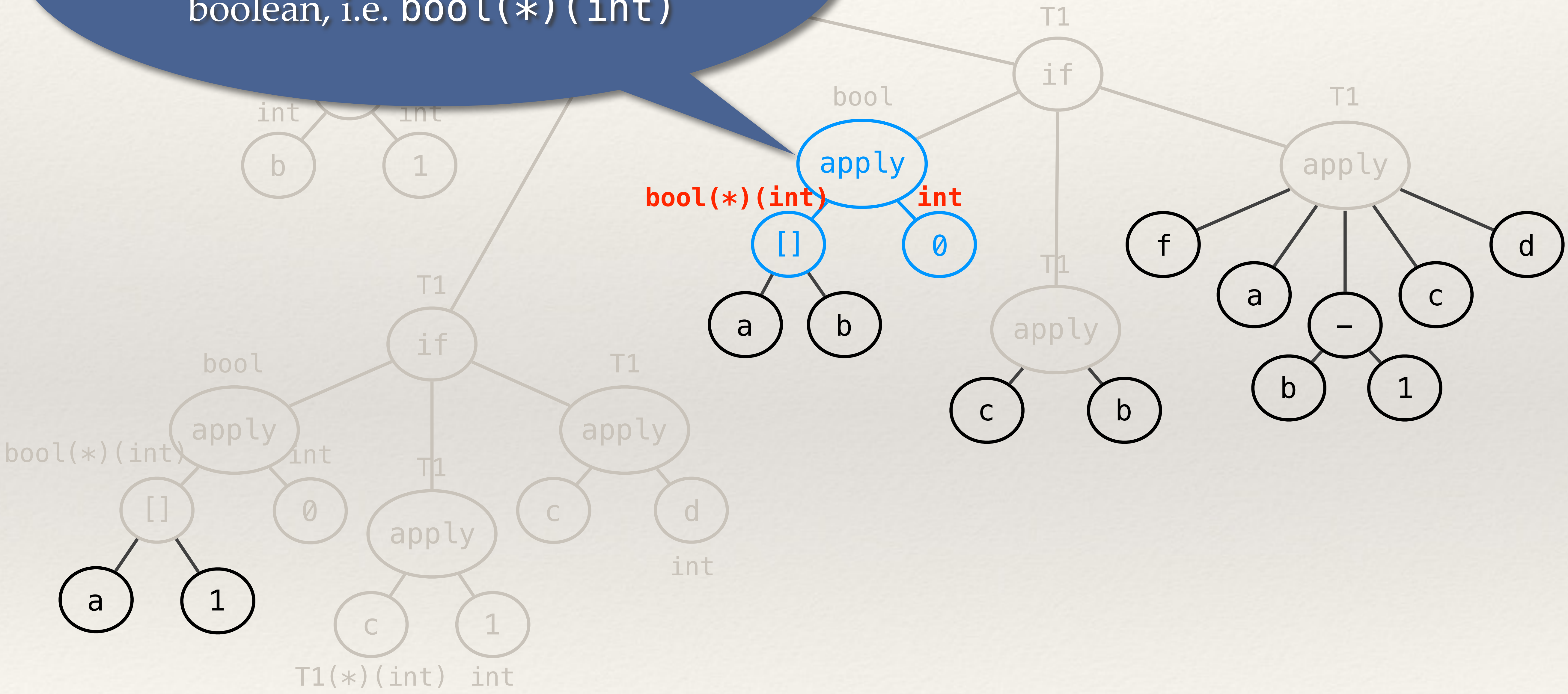
f	T1(∗)(T2,int,T1(∗)(int),**int**)
a	T2
b	int
c	T1(∗)(int)
d	**int**



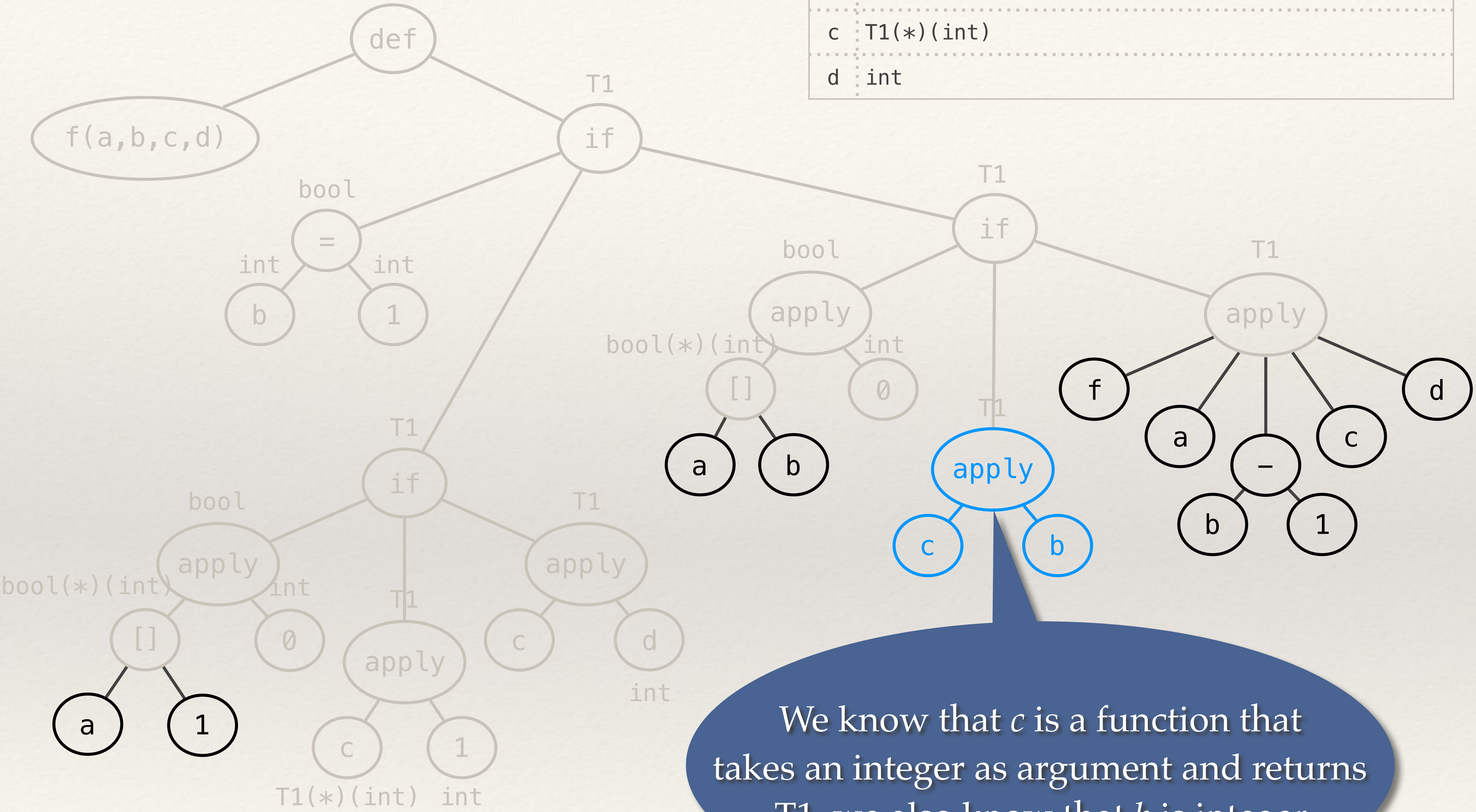
We know that *c* is a function that takes an integer as argument and returns T1. The argument passed here is *d*, so *d* must be of type int, i.e. `T5 = int`

f	T1(∗)(T2,int,T1(∗)(int),int)
a	T2
b	int
c	T1(∗)(int)
d	int

The type of [] node must be a function that takes an integer as argument and returns boolean, i.e. `bool(∗)(int)`

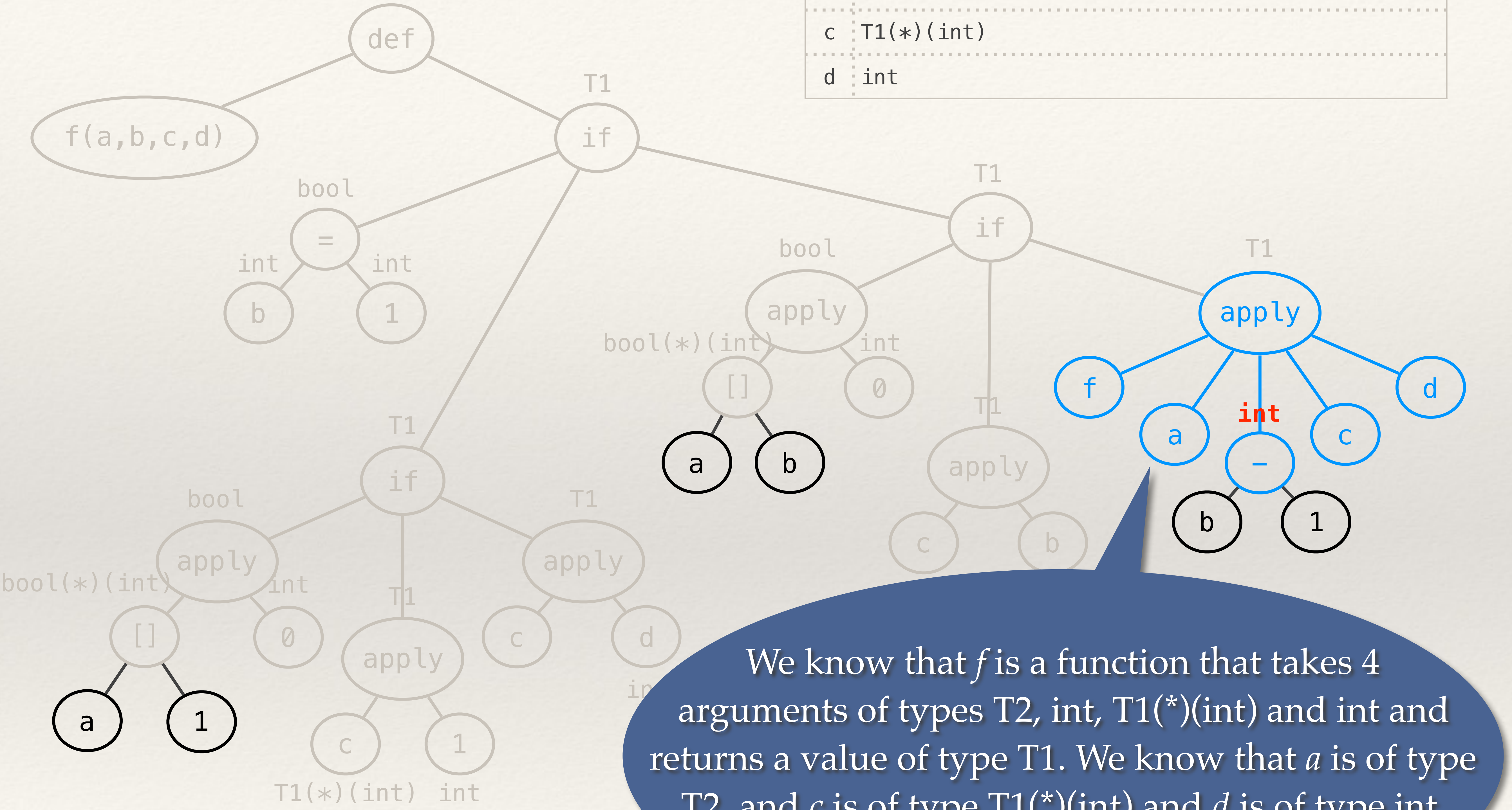


f	T1(∗)(T2,int,T1(∗)(int),int)
a	T2
b	int
c	T1(∗)(int)
d	int



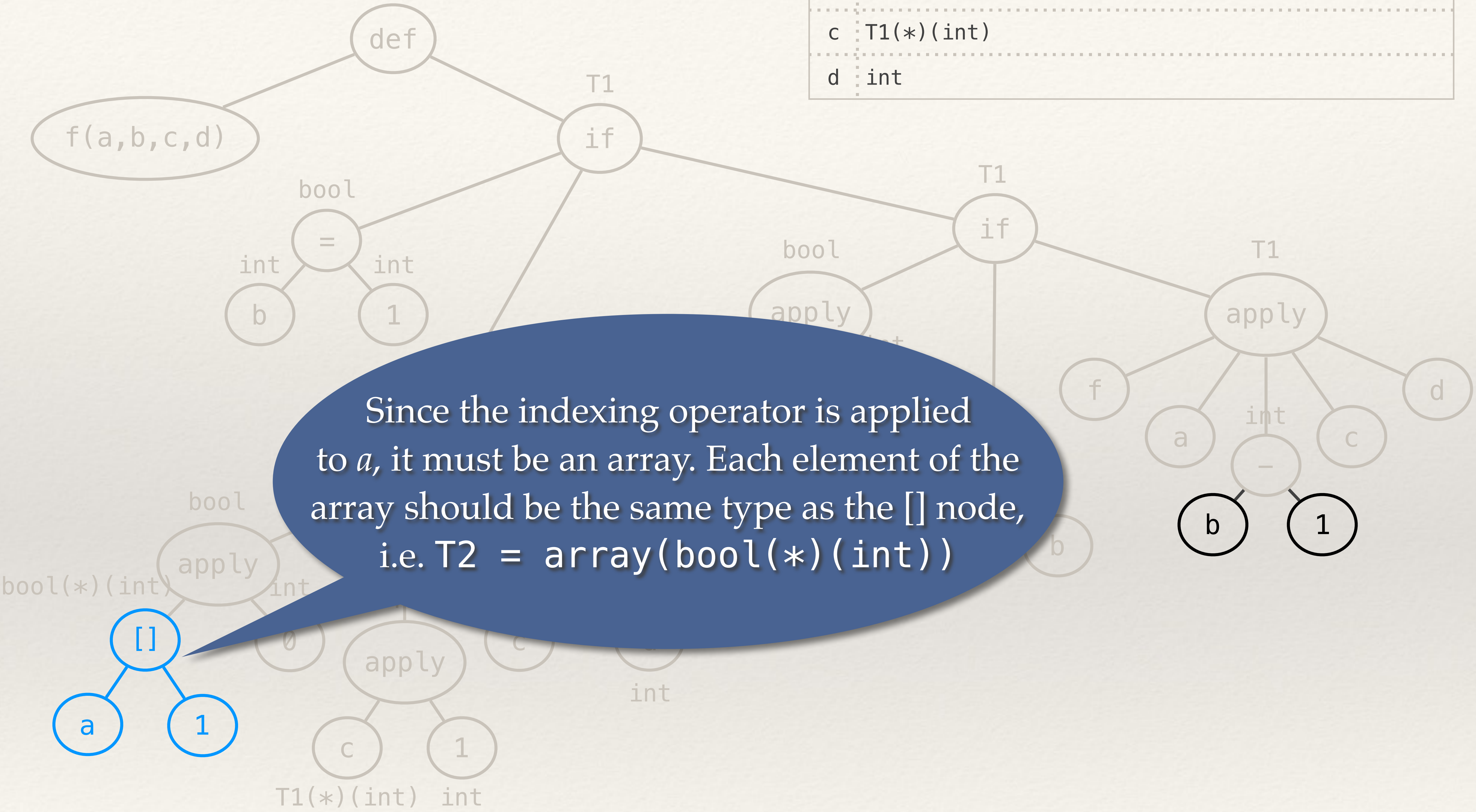
We know that *c* is a function that takes an integer as argument and returns T1, we also know that *b* is integer

f	T1(*)(T2,int,T1(*)(int),int)
a	T2
b	int
c	T1(*)(int)
d	int



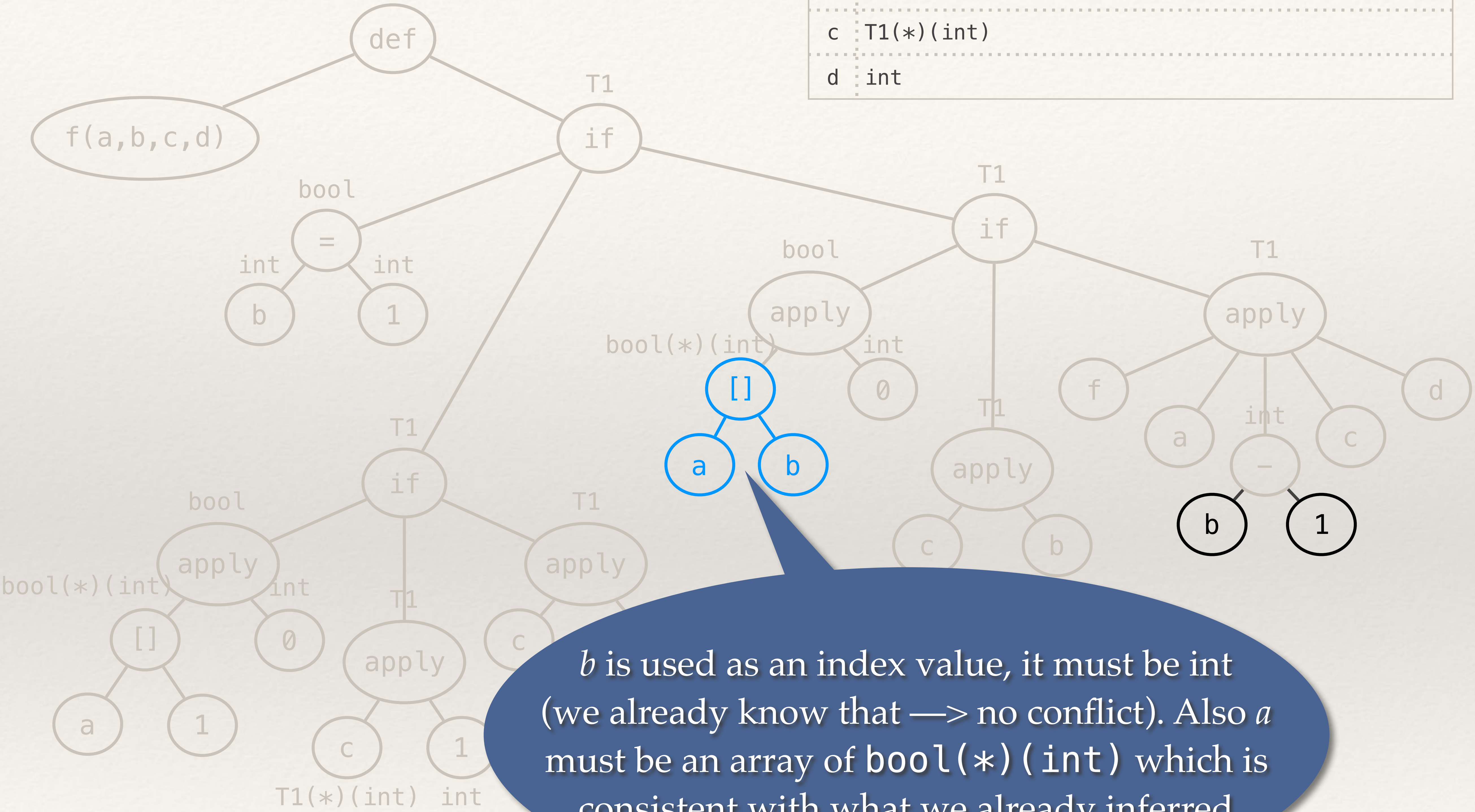
We know that *f* is a function that takes 4 arguments of types T2, int, T1(*)(int) and int and returns a value of type T1. We know that *a* is of type T2, and *c* is of type T1(*)(int) and *d* is of type int. The - node should be of type int

f	T1(∗)(array(bool(∗)(int)),int,T1(∗)(int),int)
a	array(bool(∗)(int))
b	int
c	T1(∗)(int)
d	int



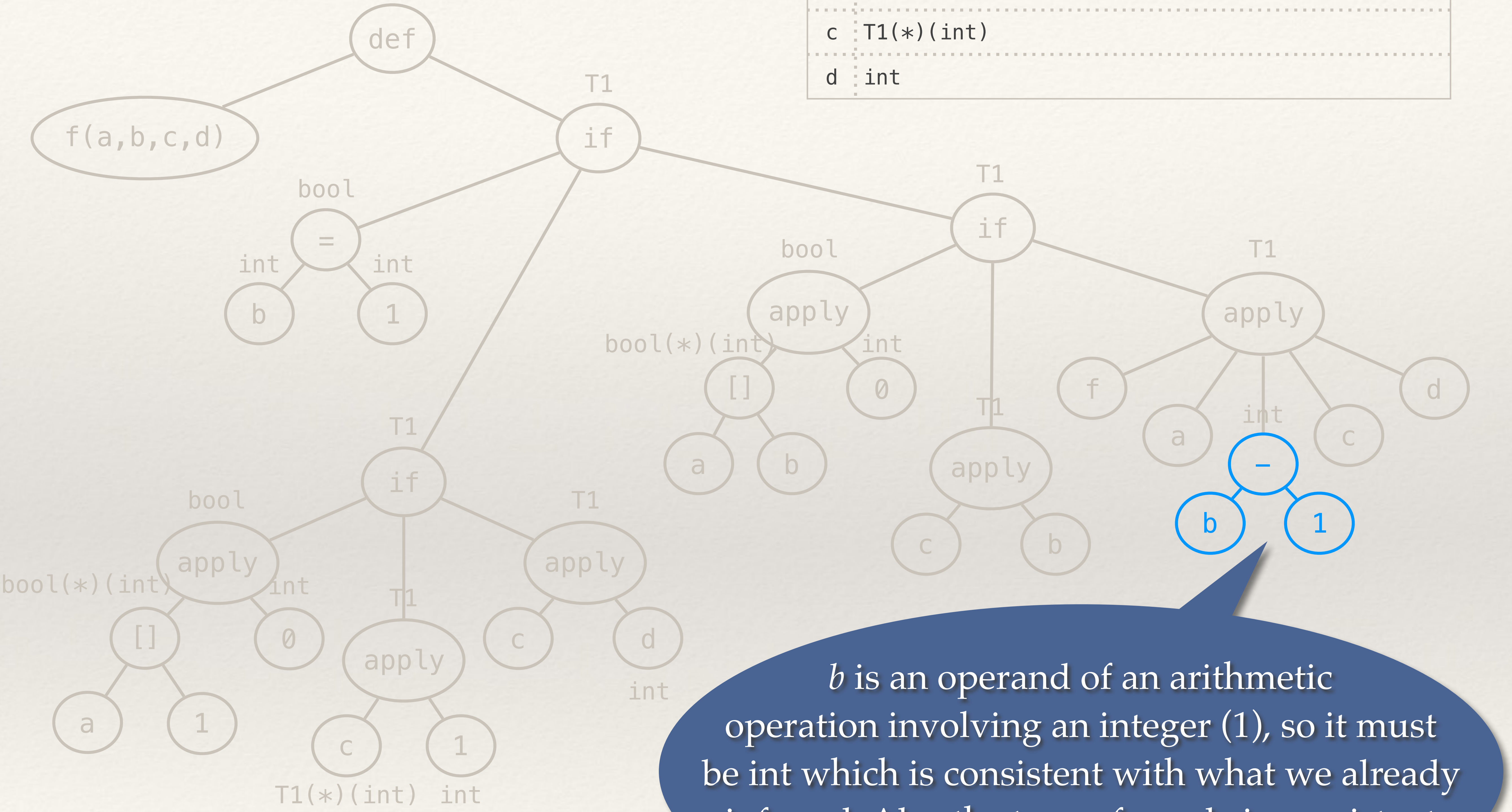
Since the indexing operator is applied to *a*, it must be an array. Each element of the array should be the same type as the [] node, i.e. `T2 = array(bool(∗)(int))`

f	T1(∗)(array(bool(∗)(int)),int,T1(∗)(int),int)
a	array(bool(∗)(int))
b	int
c	T1(∗)(int)
d	int



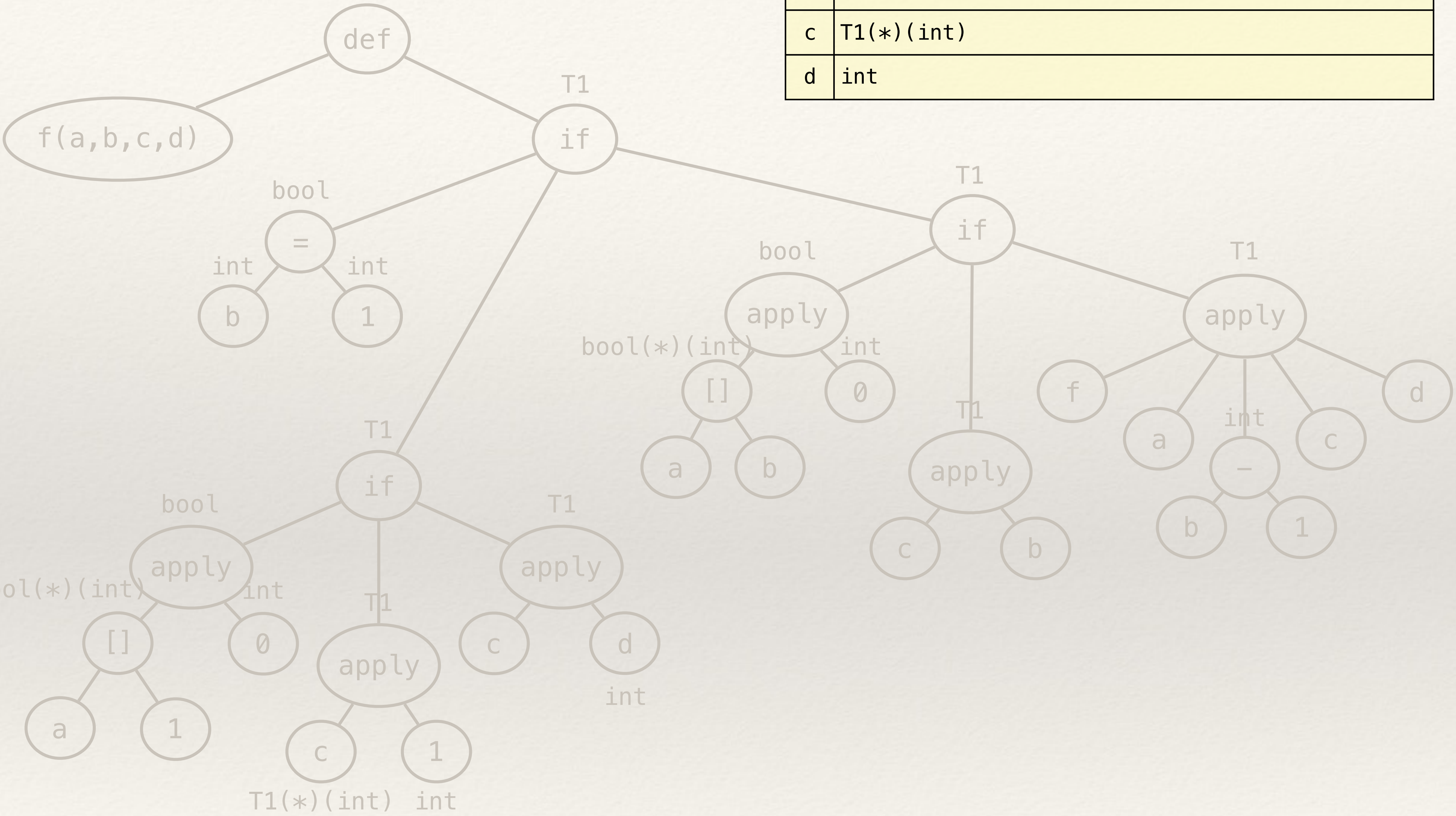
b is used as an index value, it must be int (we already know that —> no conflict). Also *a* must be an array of `bool(∗)(int)` which is consistent with what we already inferred

f	T1(*)(array(bool(*)(int)),int,T1(*)(int),int)
a	array(bool(*)(int))
b	int
c	T1(*)(int)
d	int



b is an operand of an arithmetic operation involving an integer (1), so it must be int which is consistent with what we already inferred. Also the type of - node is consistent with the types of operands

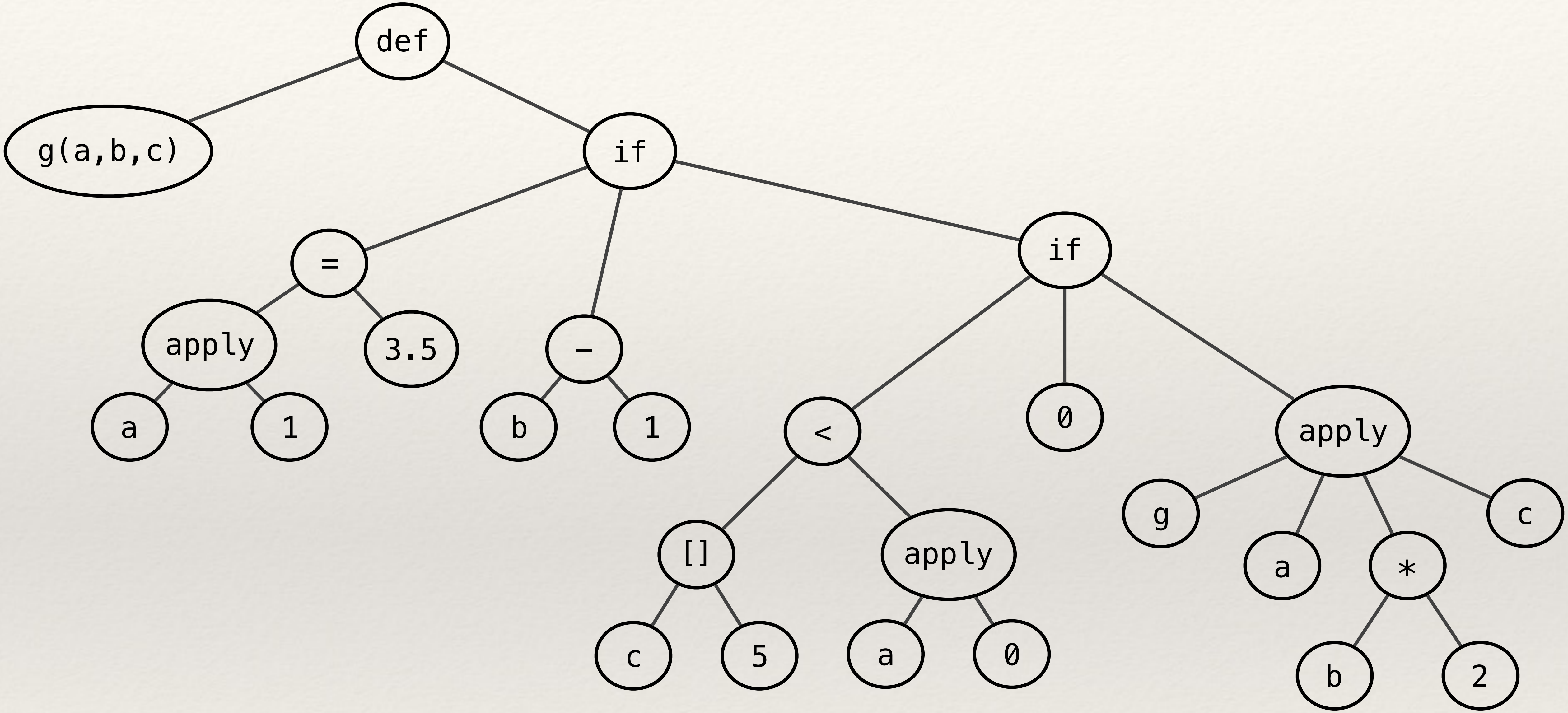
f	T1(∗)(array(bool(∗)(int)),int,T1(∗)(int),int)
a	array(bool(∗)(int))
b	int
c	T1(∗)(int)
d	int



Example #2

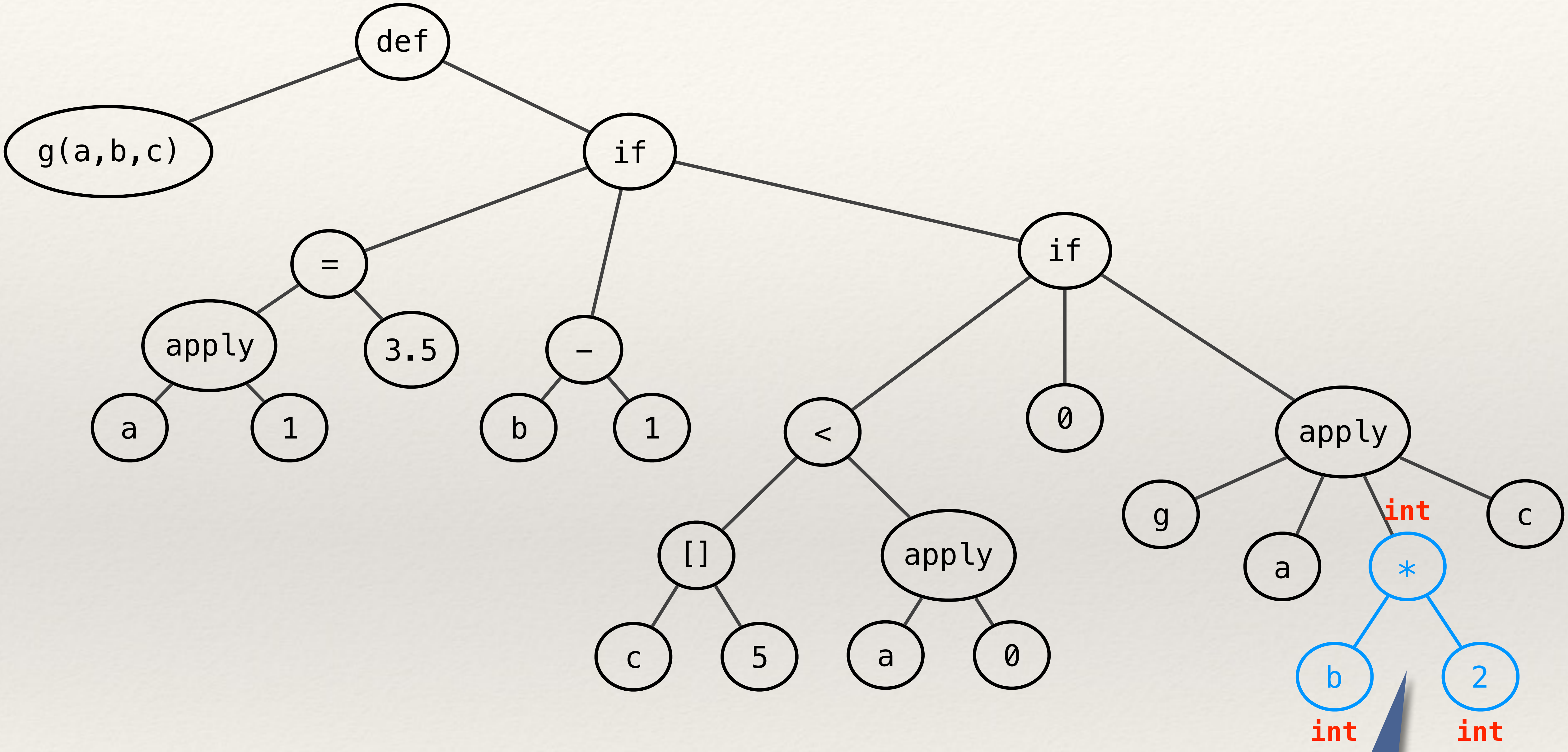
```
g(a,b,c) = if a(1) = 3.5 then
               b - 1
           else
               if c[5] < a(0) then
                   0
               else
                   g(a, b * 2, c)
```

g	T1(∗)(T2,T3,T4)
a	T2
b	T3
c	T4



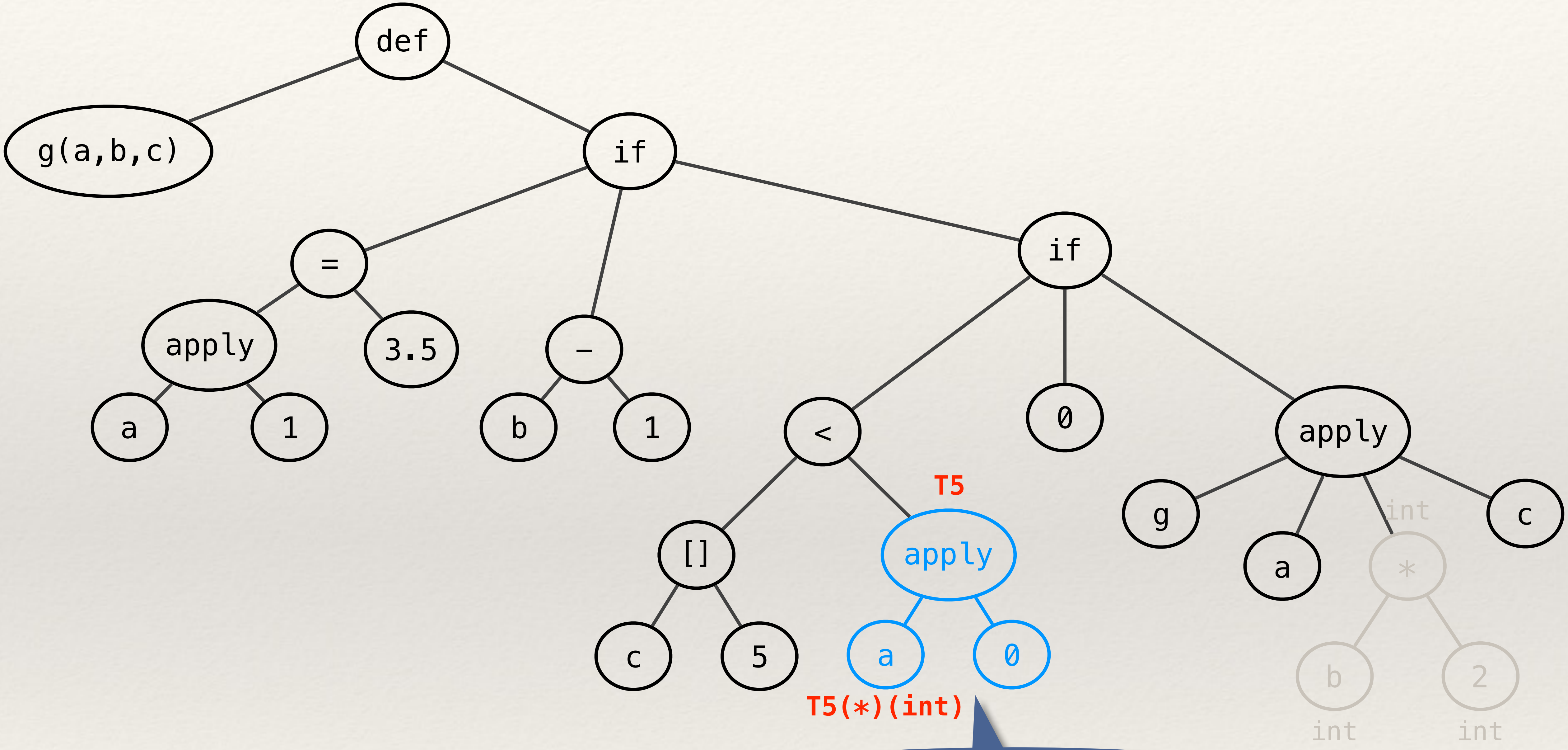
Bottom-Up order

g	T1(*)(T2,**int**,T4)
a	T2
b	**int**
c	T4



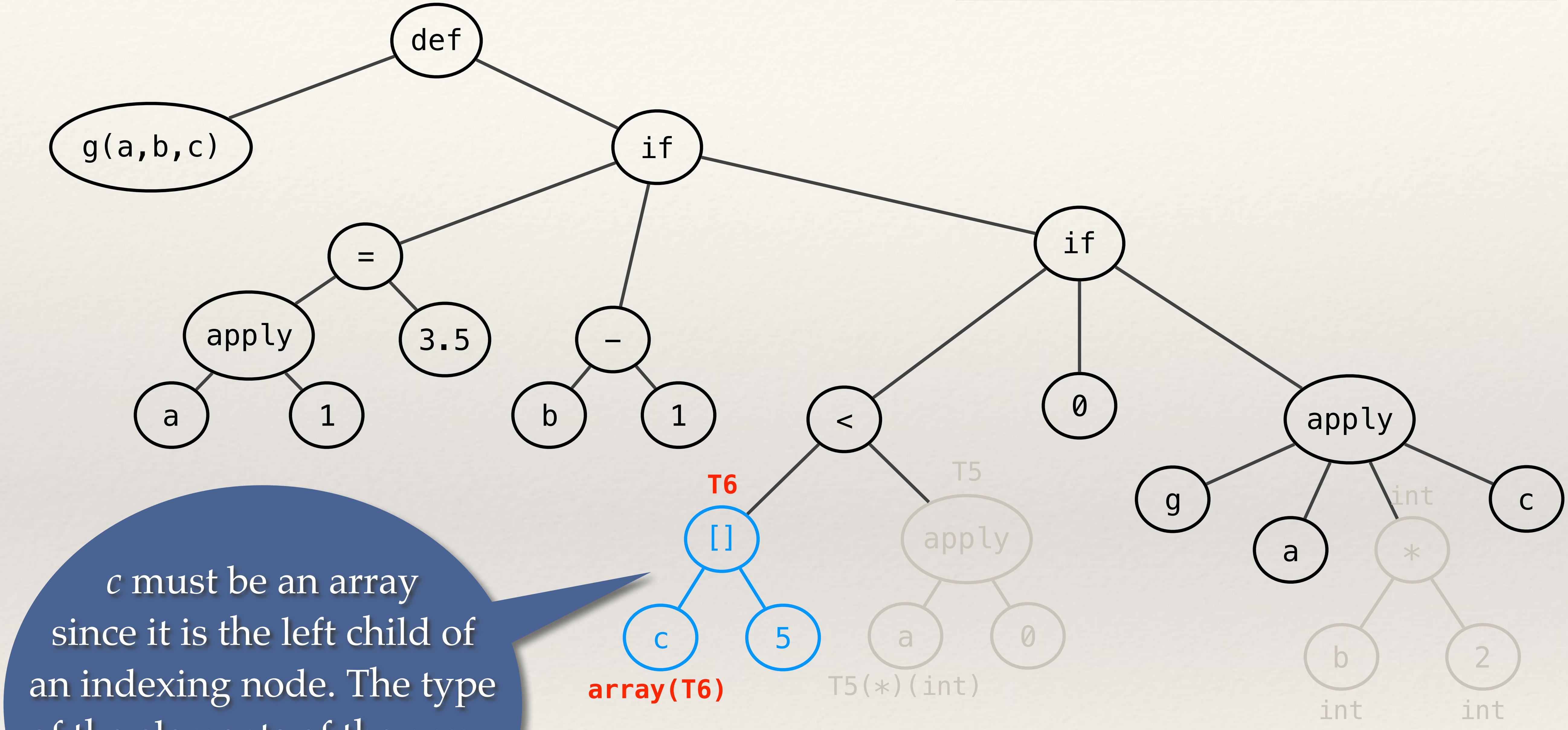
b must be an integer,
i.e. `T3 = int`

g	T1(*)(**T5(*)(int)**,int,T4)
a	**T5(*)(int)**
b	int
c	T4



a must be a function that takes an integer as argument and we don't know its return type yet, i.e. `T2 = T5(*)(int)`

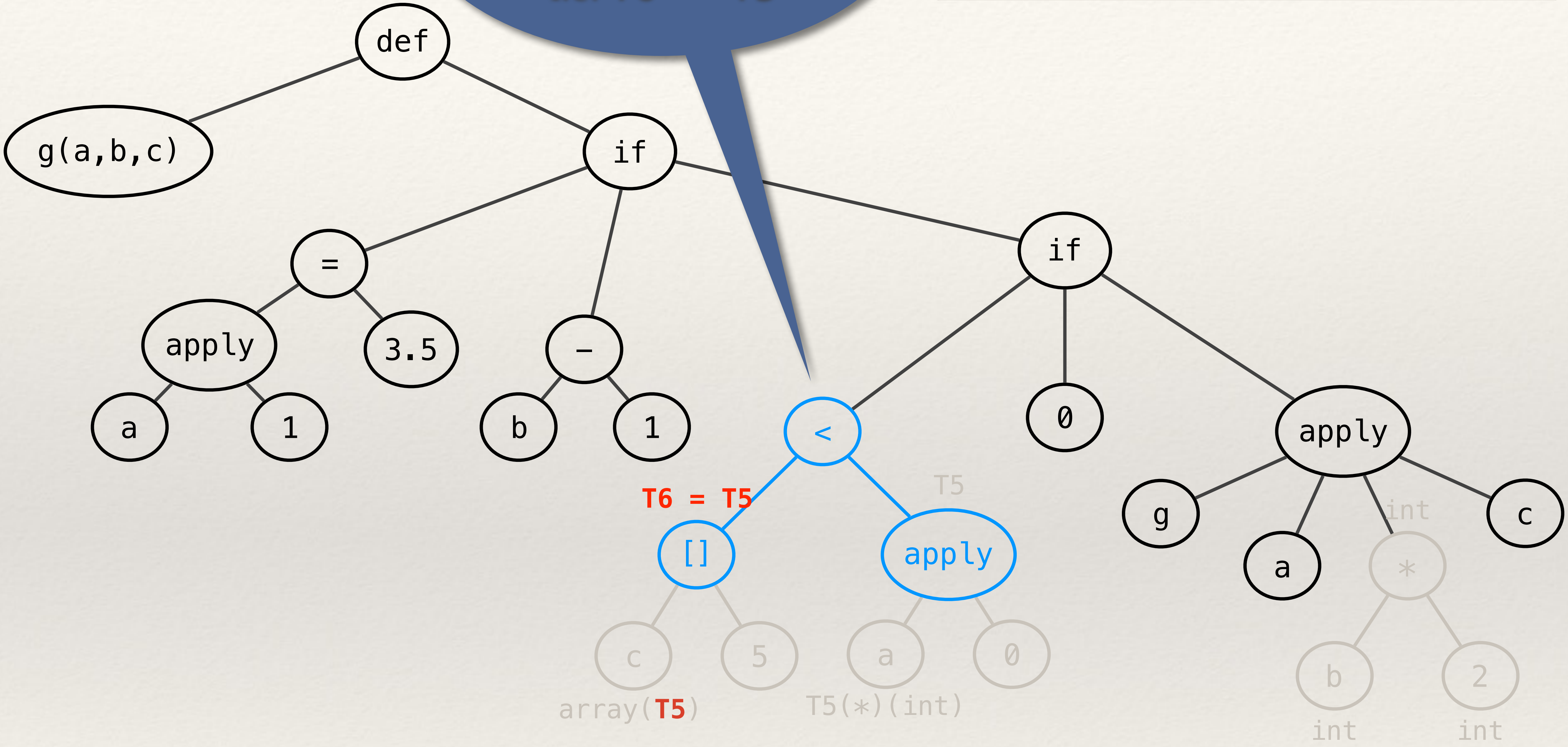
g	T1(*)(T5(*)(int),int,array(T6))
a	T5(*)(int)
b	int
c	array(T6)



c must be an array since it is the left child of an indexing node. The type of the elements of the array are not known yet, i.e. `T4 = array(T6)`

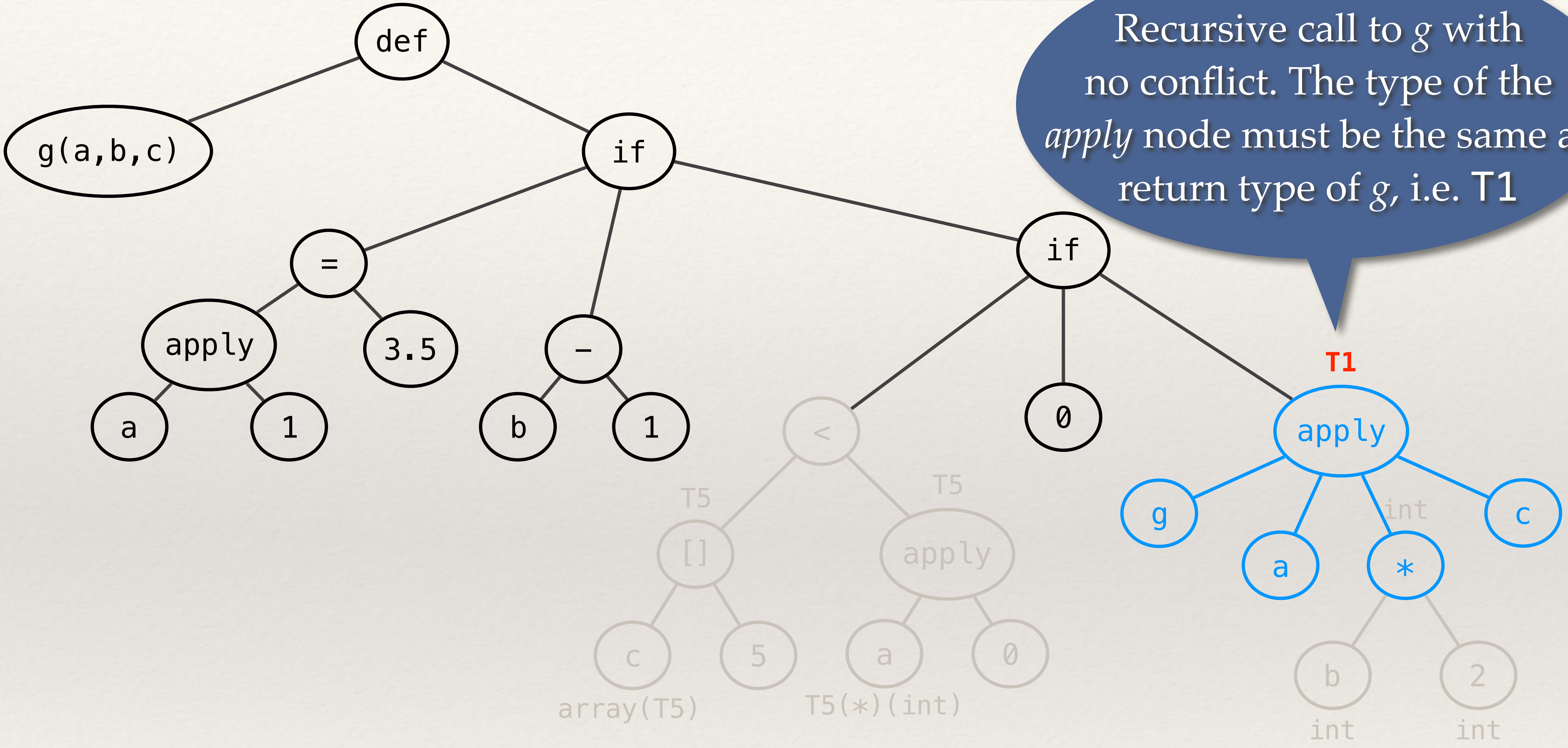
The operands of a comparison operator must be of the same type, i.e. T6 = T5

g	T1(*)(T5(*)(int),int,array(T5))
a	T5(*)(int)
b	int
c	array(T5)

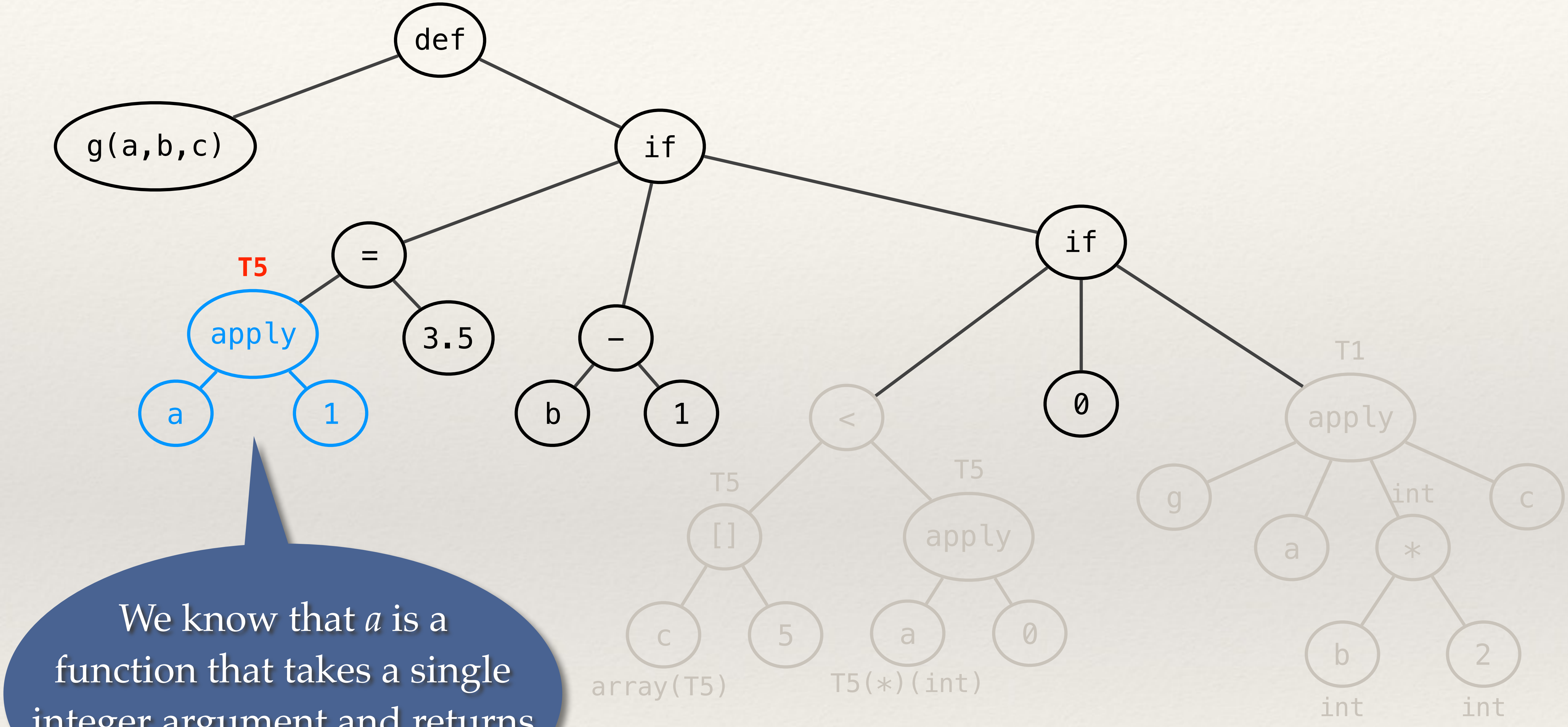


g	T1(∗)(T5(∗)(int),int,array(T5))
a	T5(∗)(int)
b	int
c	array(T5)

Recursive call to *g* with no conflict. The type of the *apply* node must be the same as return type of *g*, i.e. T1

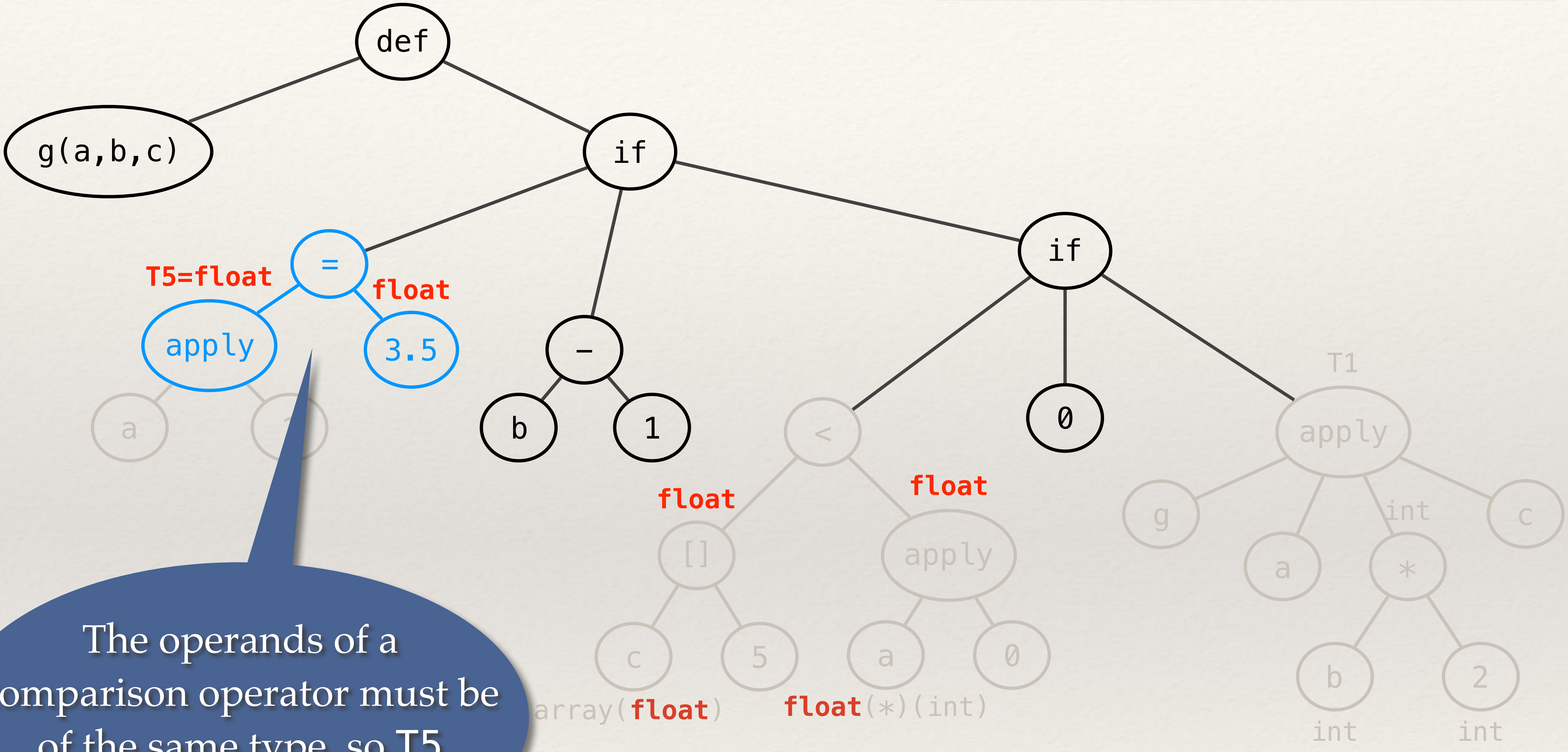


g	T1(∗)(T5(∗)(int),int,array(T5))
a	T5(∗)(int)
b	int
c	array(T5)



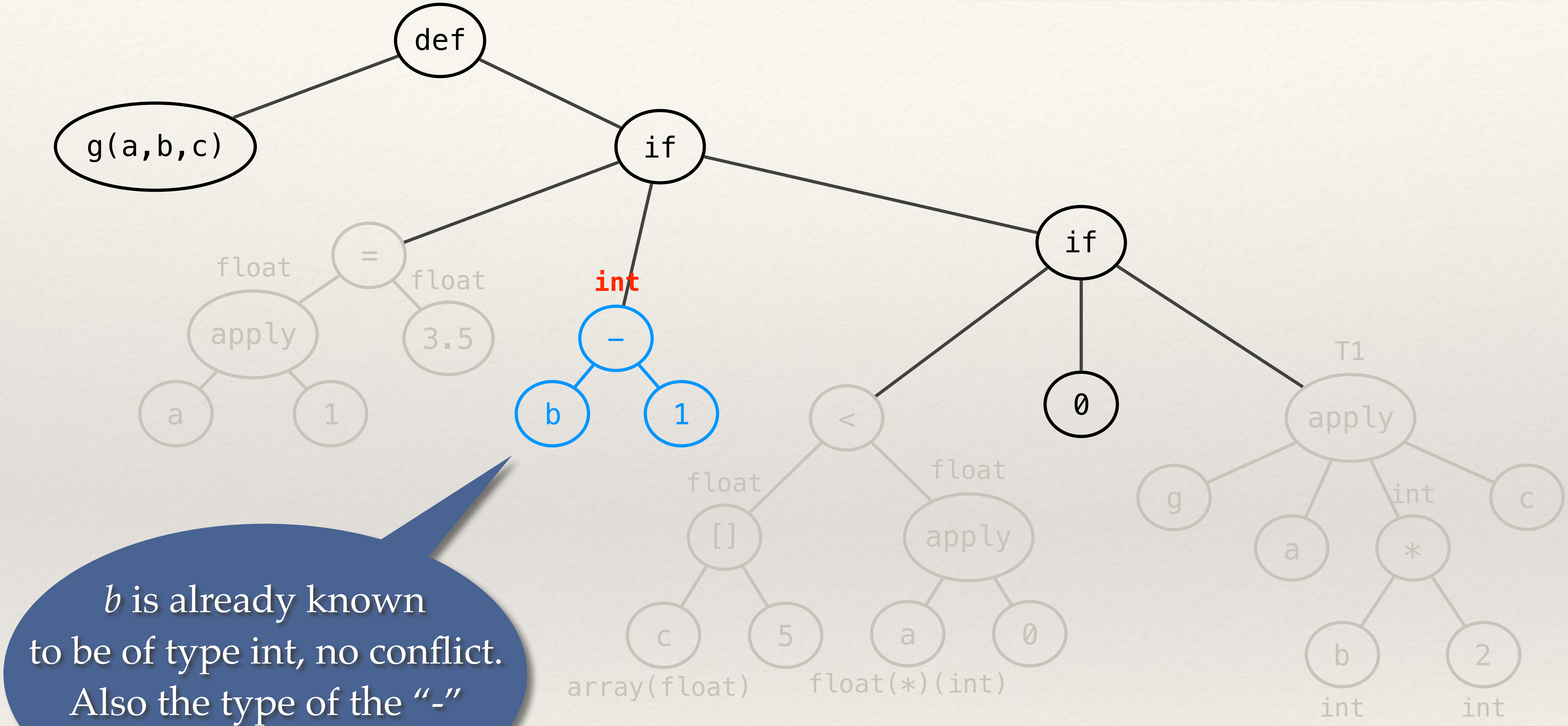
We know that *a* is a function that takes a single integer argument and returns a value of type T5

g	T1(*)(float(*)(int),int,array(float))
a	float(*)(int)
b	int
c	array(float)



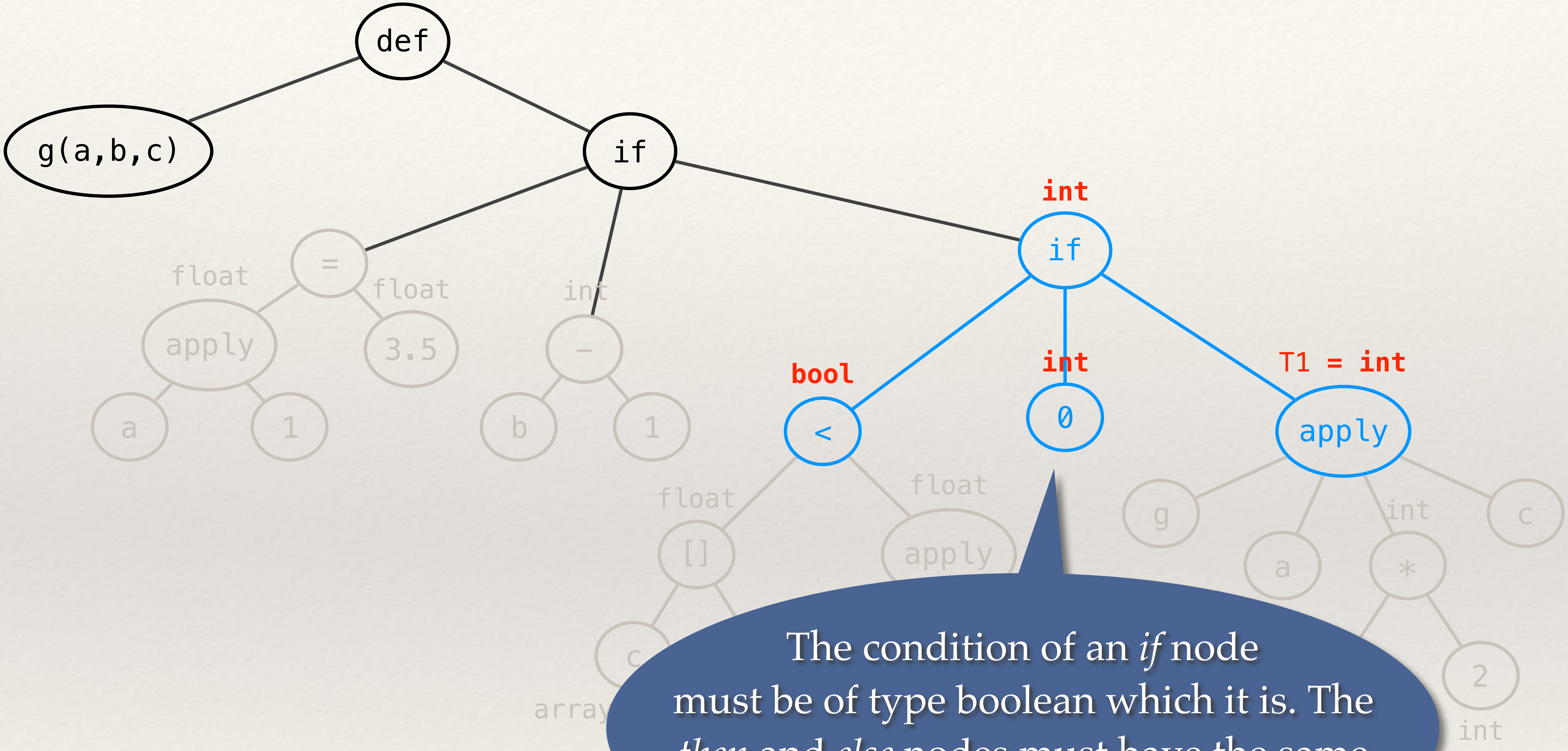
The operands of a comparison operator must be of the same type, so T5 should be float

g	T1(*)(float(*)(int),int,array(float))
a	float(*)(int)
b	int
c	array(float)



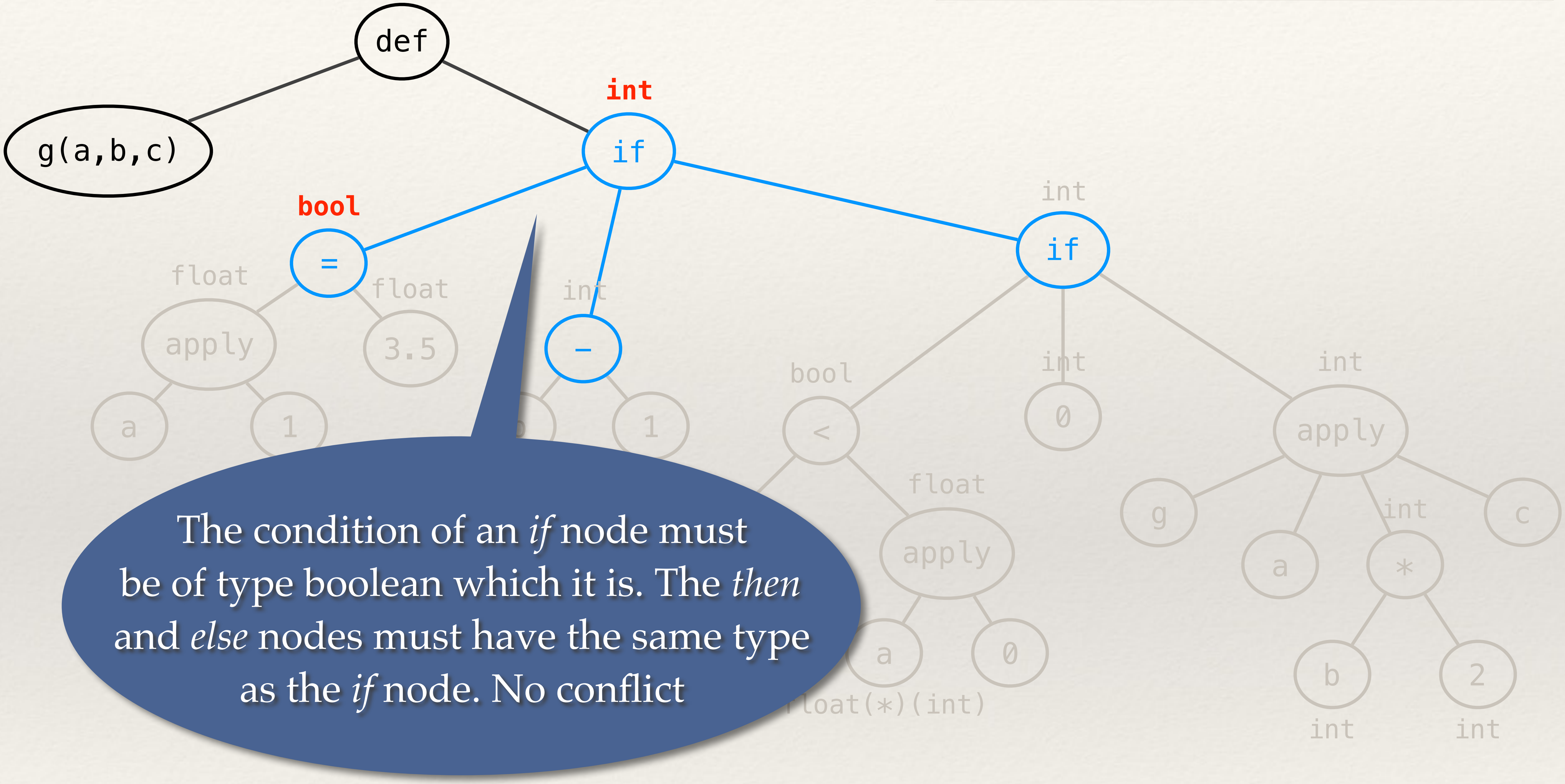
b is already known to be of type int, no conflict. Also the type of the "-" node must be int

g	int(*)(float(*)(int),int,array(float))
a	float(*)(int)
b	int
c	array(float)



The condition of an *if* node must be of type boolean which it is. The *then* and *else* nodes must have the same type as the *if* node. So `T1 = int`

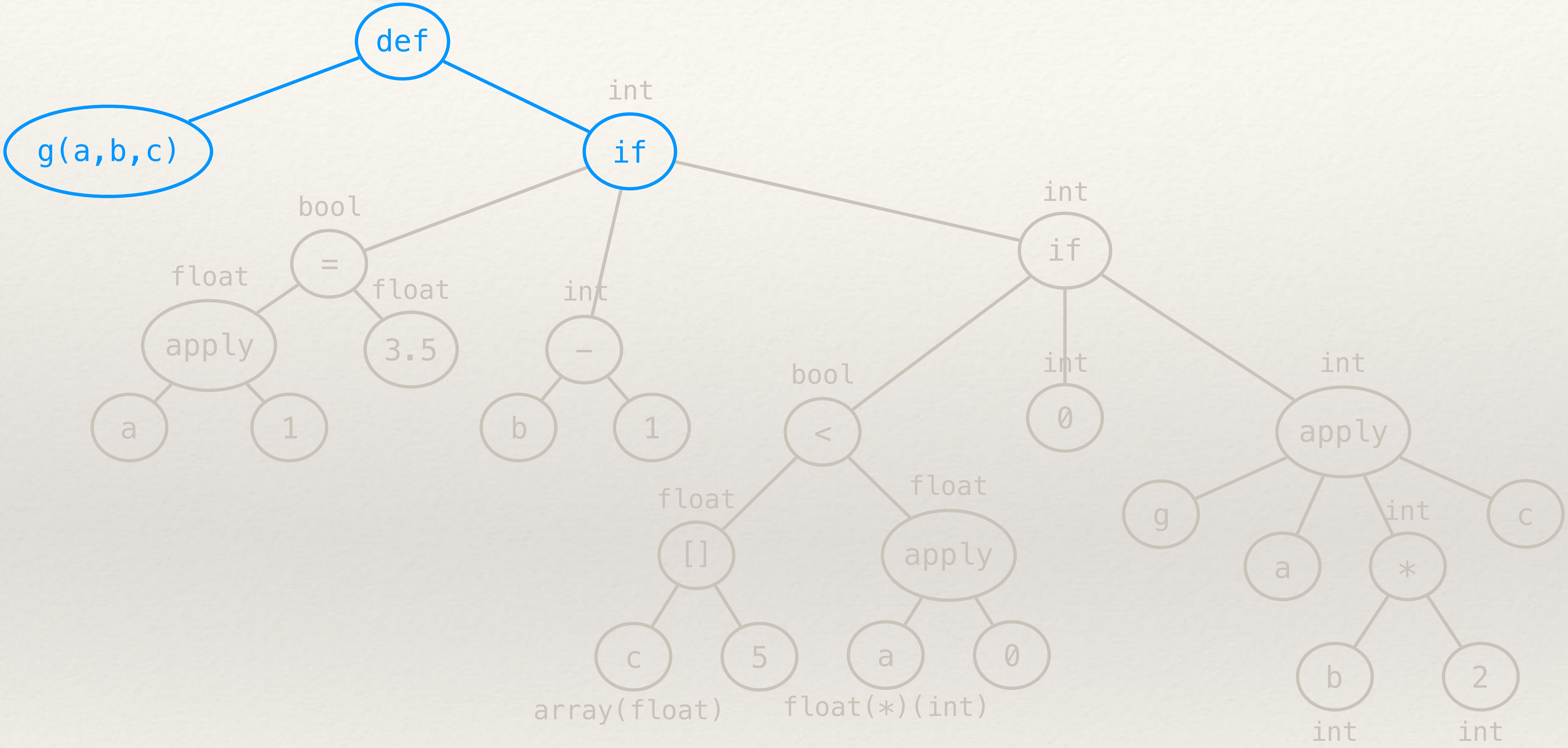
g	int(*)(float(*)(int),int,array(float))
a	float(*)(int)
b	int
c	array(float)



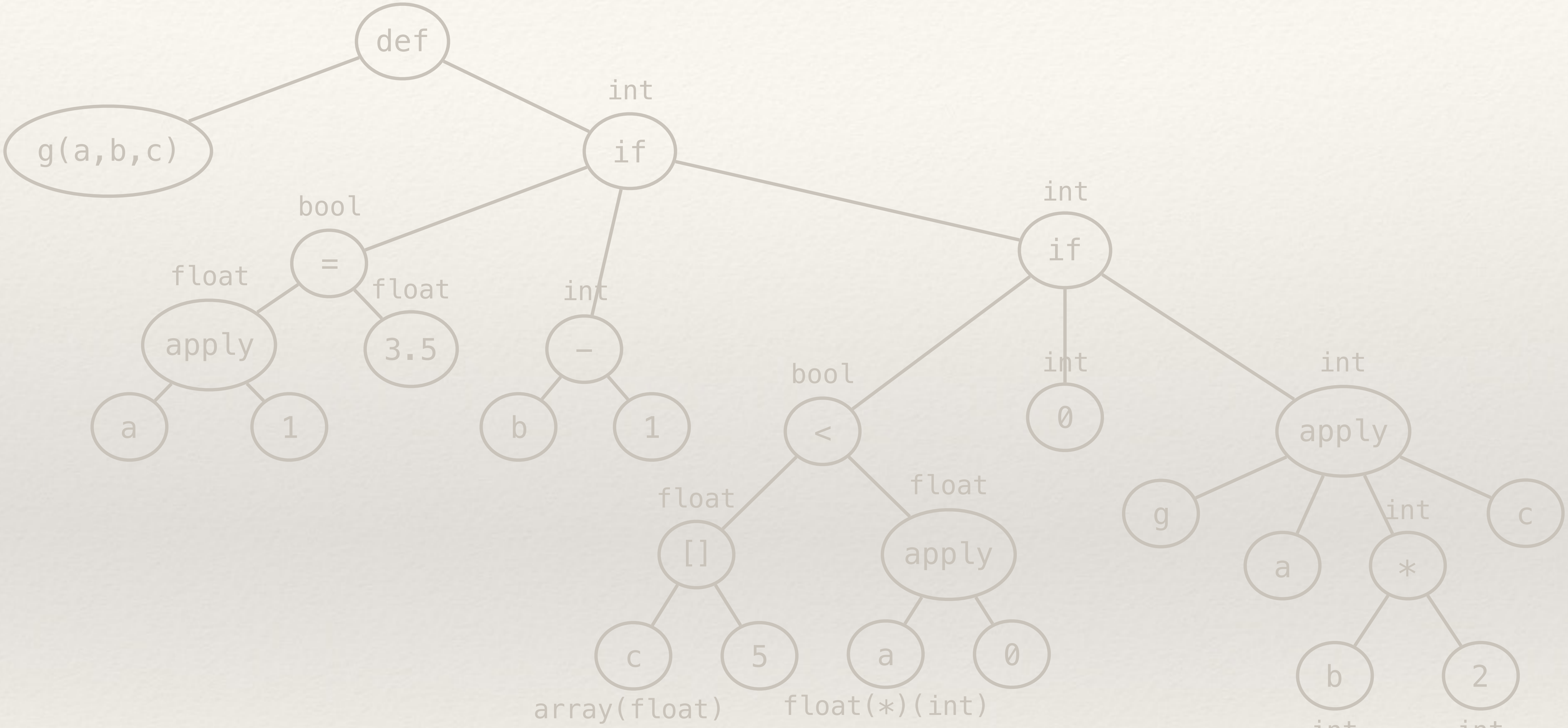
The condition of an *if* node must be of type boolean which it is. The *then* and *else* nodes must have the same type as the *if* node. No conflict

The return type of the function *g* must be the same as the type of the *if* node which it is —> No conflict

g	int(*)(float(*)(int),int,array(float))
a	float(*)(int)
b	int
c	array(float)



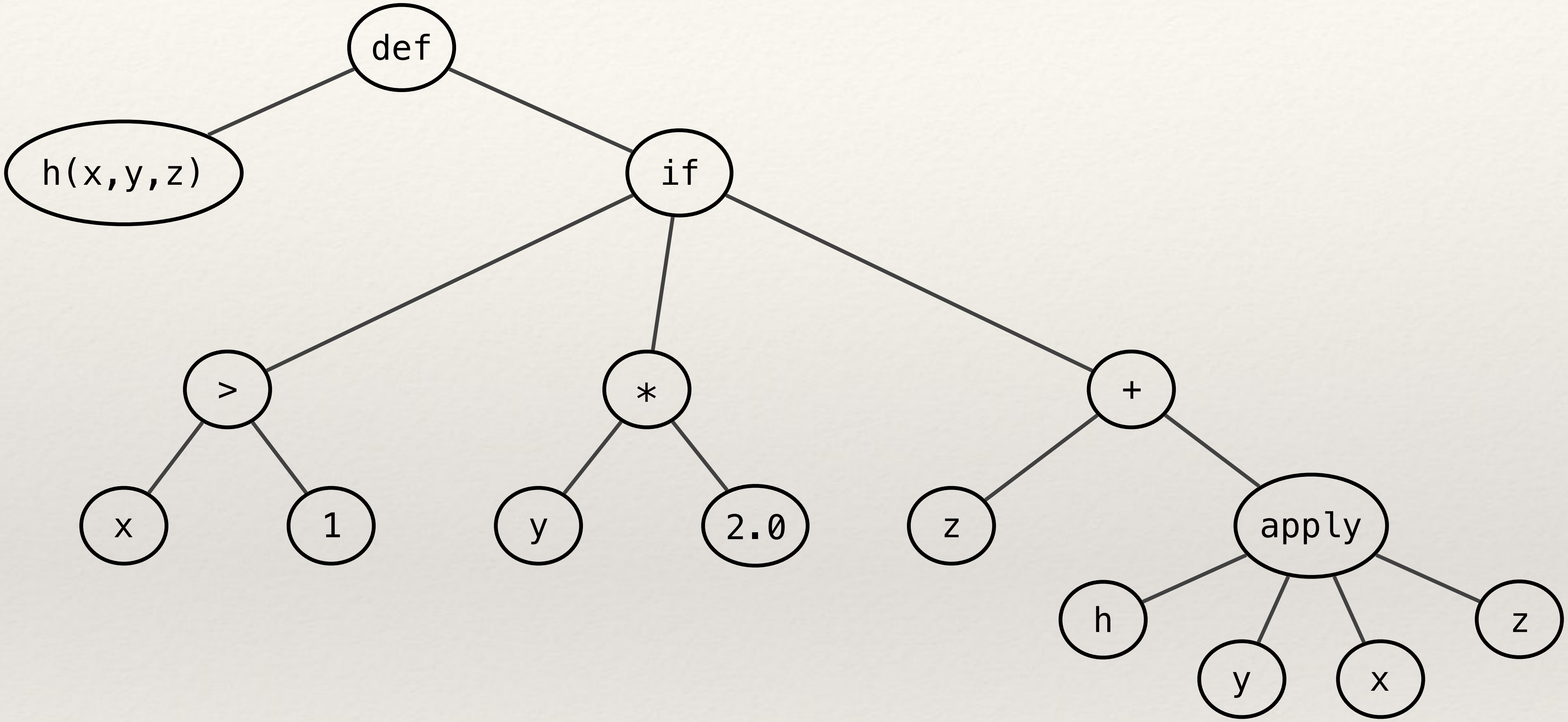
g	int(∗)(float(∗)(int),int,array(float))
a	float(∗)(int)
b	int
c	array(float)



Example #3

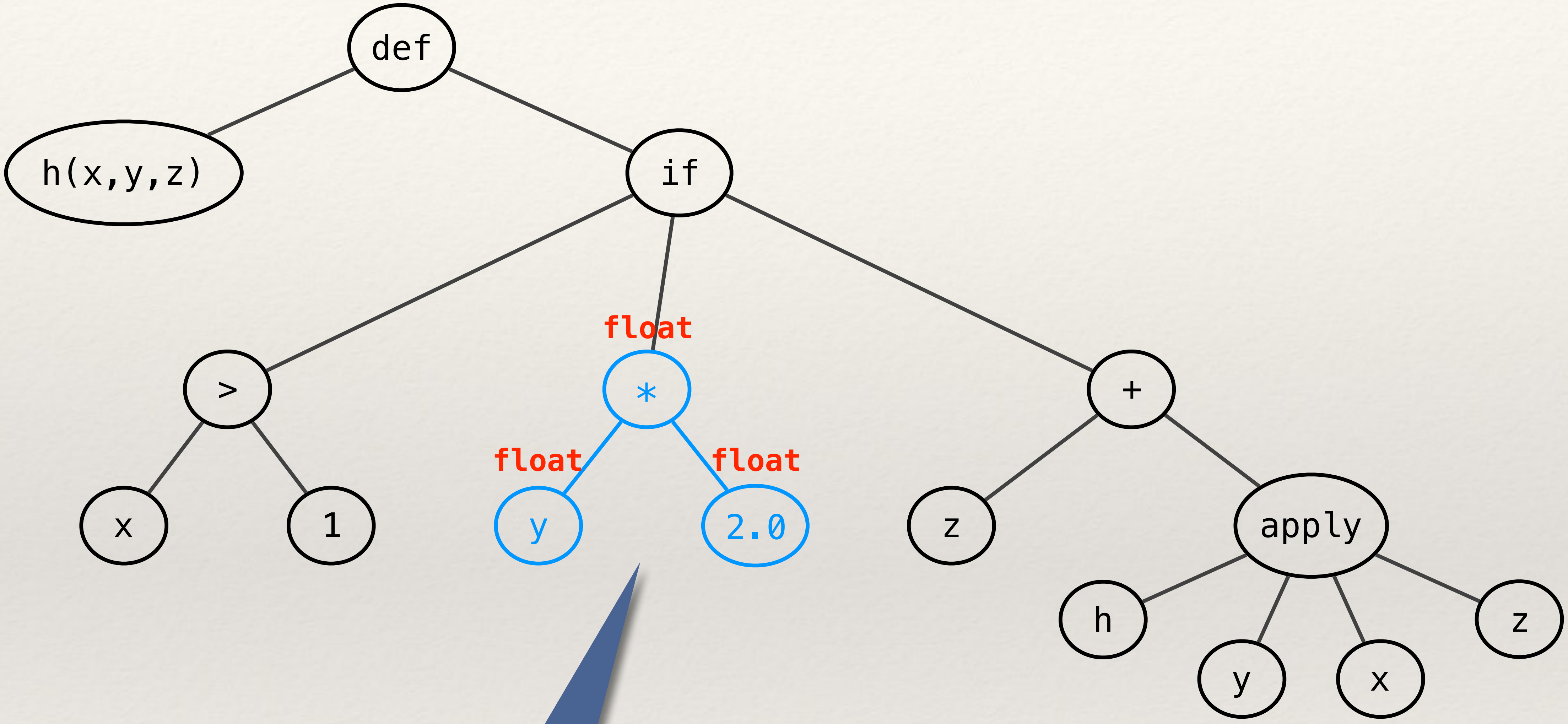
```
h(x,y,z) = if x > 1 then
               y * 2.0
           else
               z + h(y, x, z)
```

h	T1(*)(T2,T3,T4)
x	T2
y	T3
z	T4



Random order

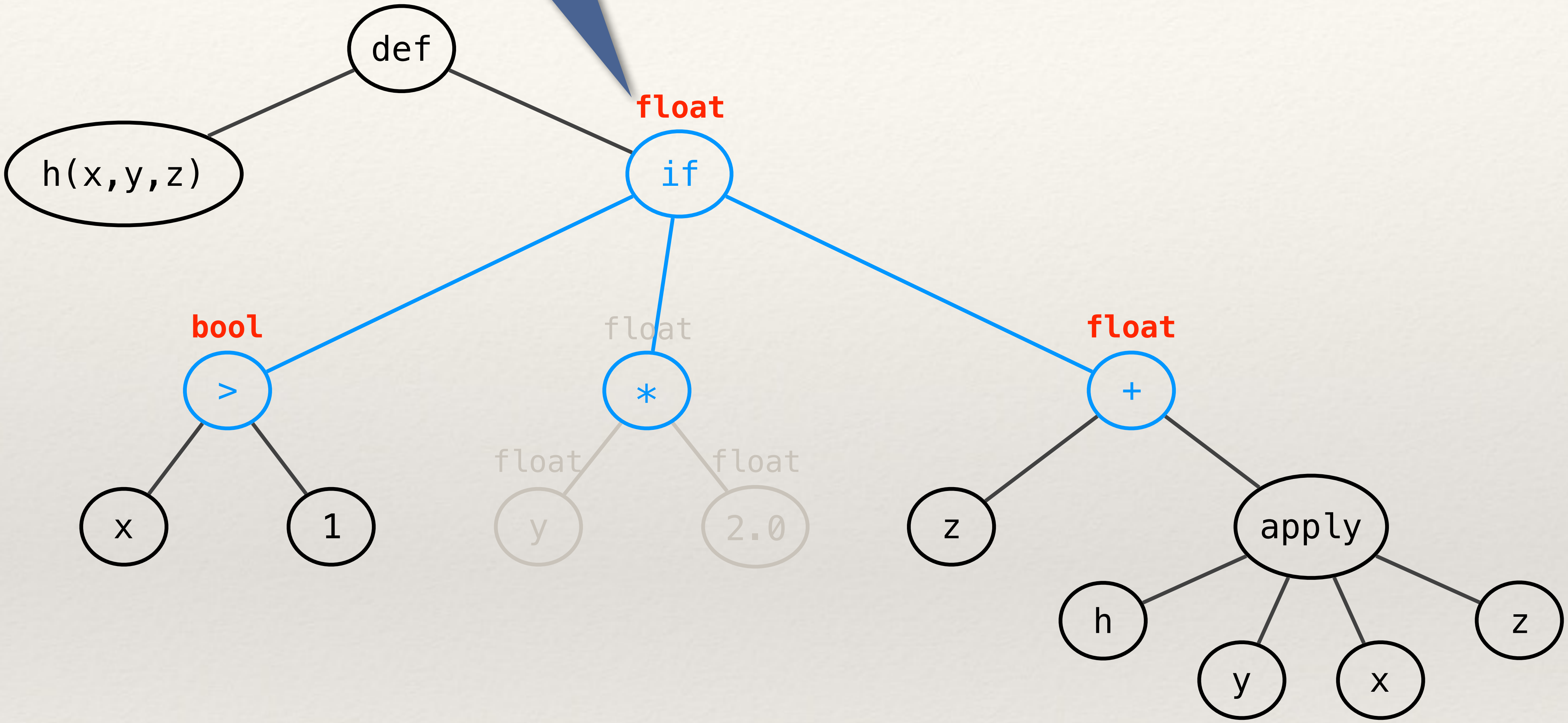
h	T1(∗)(T2,**float**,T4)
x	T2
y	**float**
z	T4



y must be float since it is used in an arithmetic operation involving 2.0, hence `T3 = float`

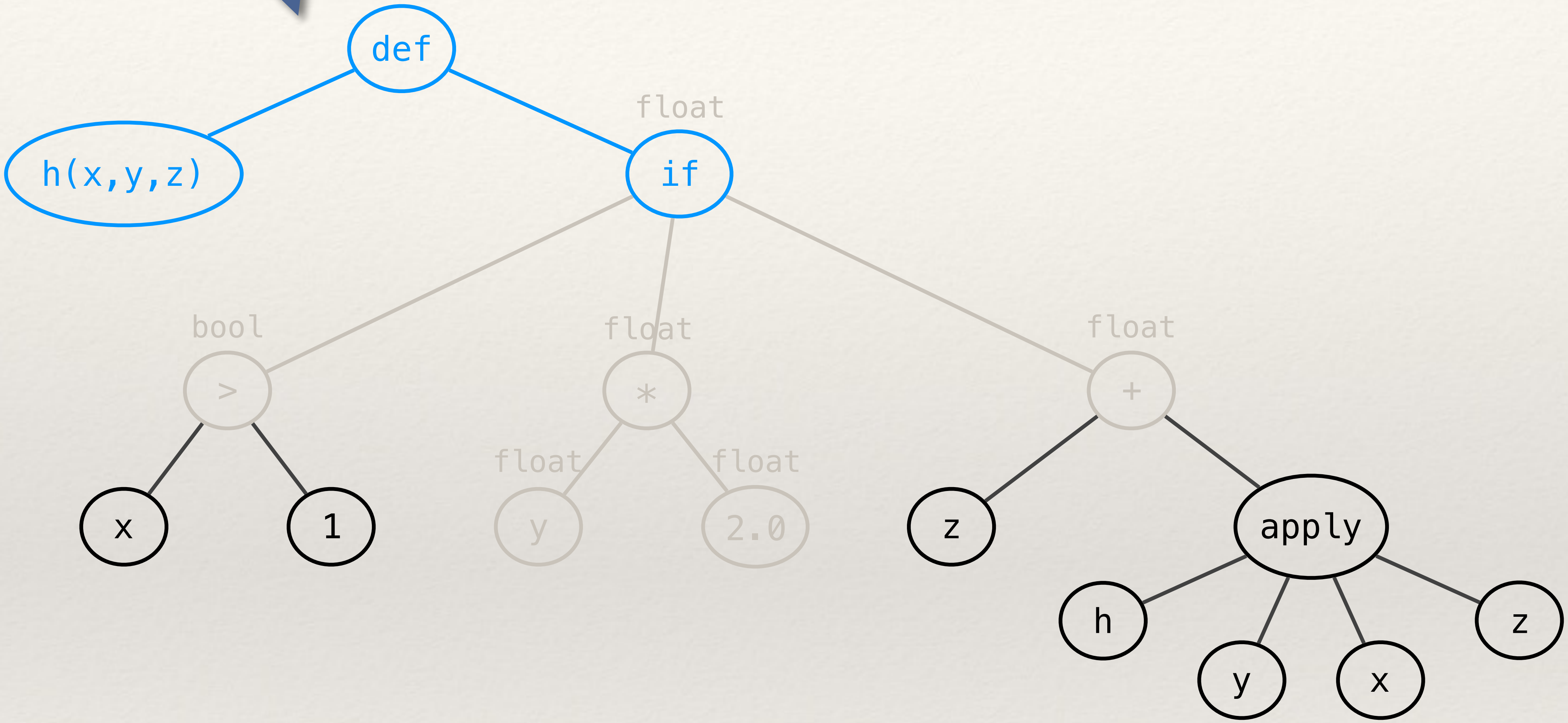
The condition node must be boolean and the *then* and *else* nodes should be of the same type as the *if* node

h	T1(*)(T2,float,T4)
x	T2
y	float
z	T4

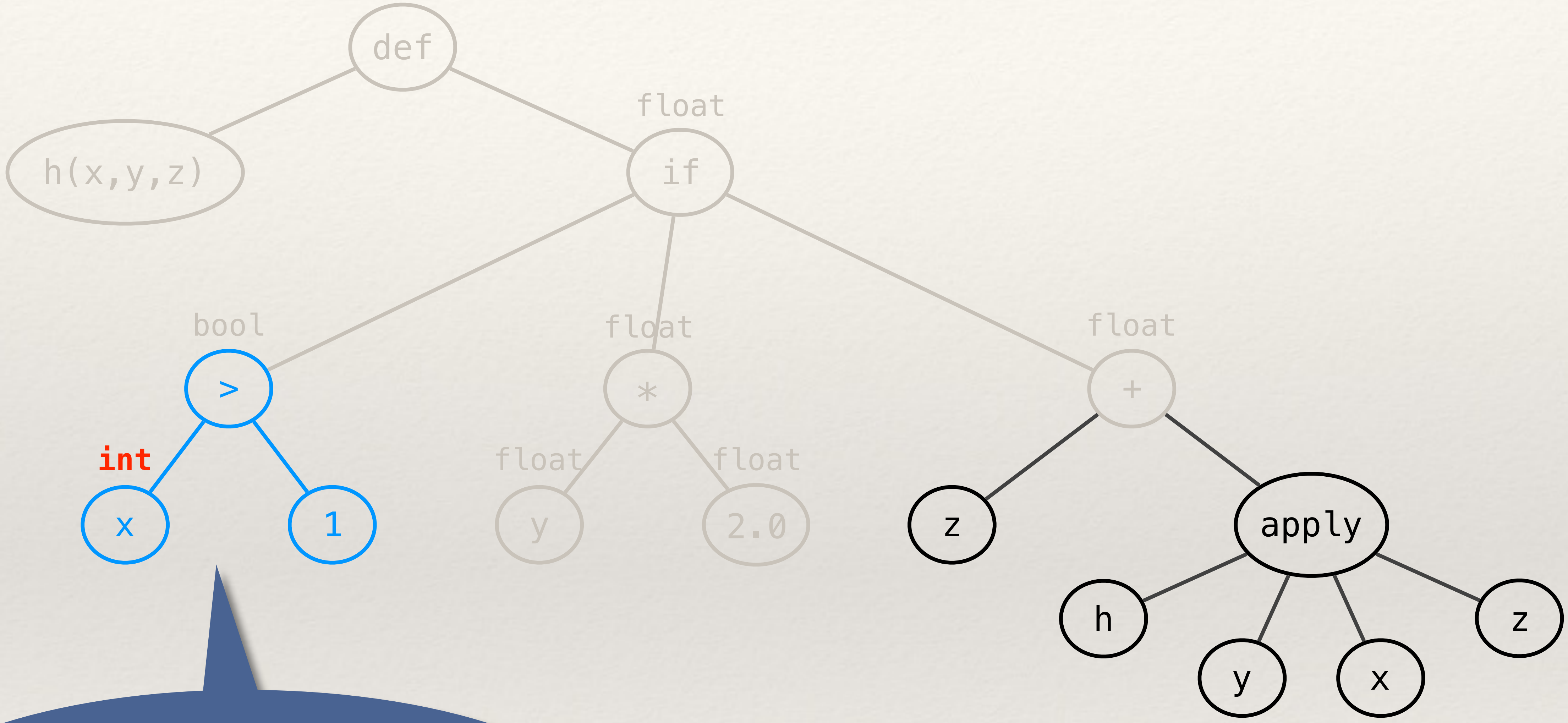


The return type of *h* must be the same as the type of the *if* node, i.e. `T1 = float`

h	**float**(*)(T2,float,T4)
x	T2
y	float
z	T4

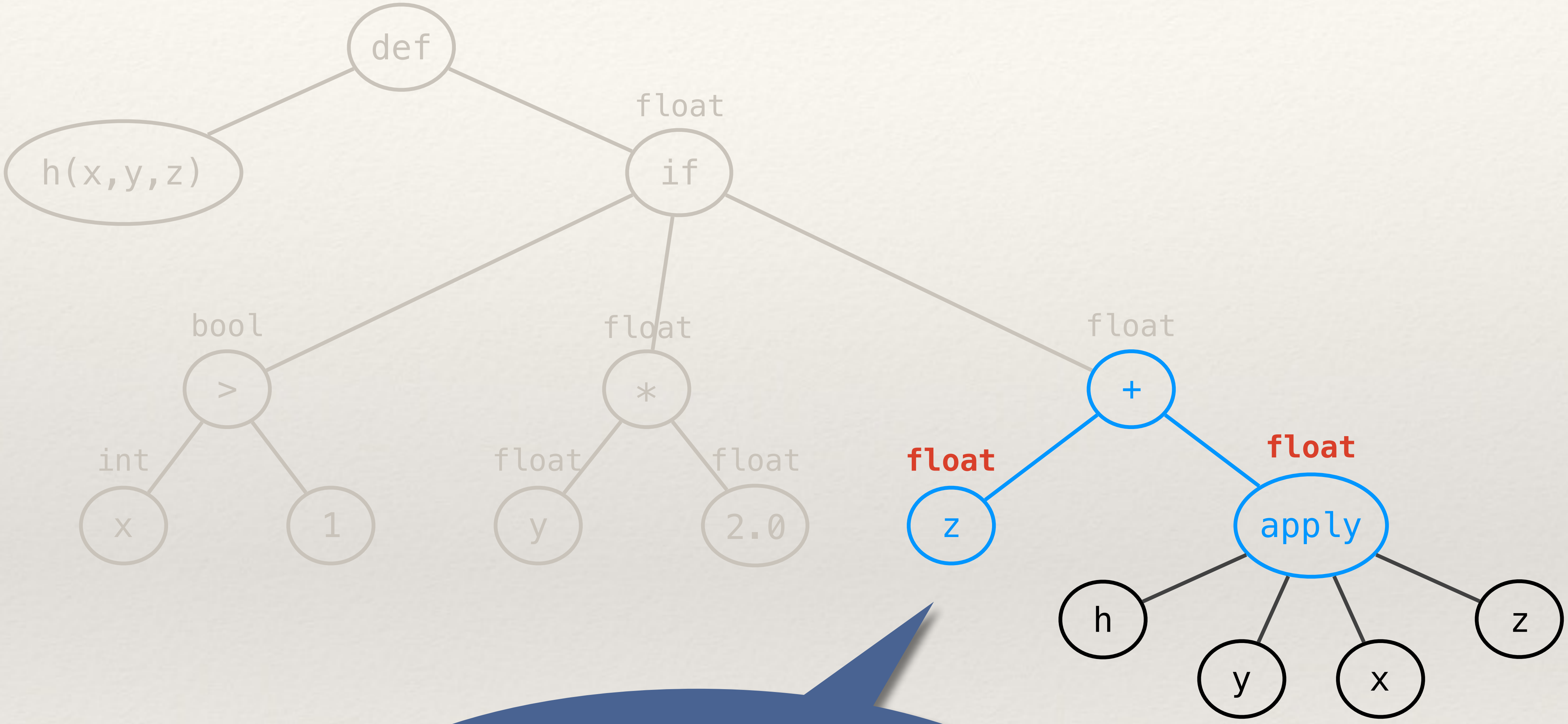


h	float(*)(int,float,T4)
x	int
y	float
z	T4



x must be int since it is compared with integer constant 1, i.e. `T2 = int`

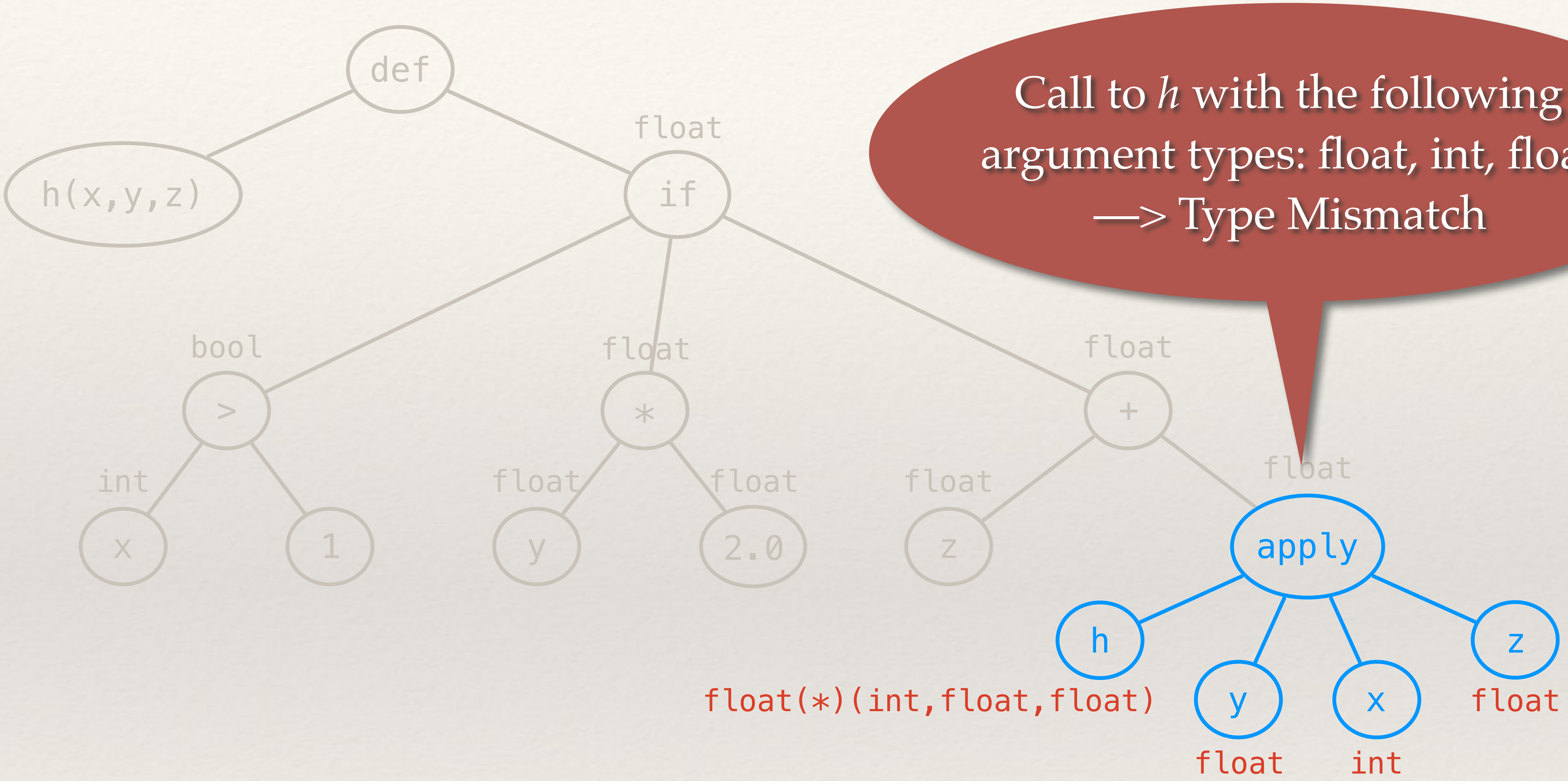
h	float(*)(int,float,**float**)
x	int
y	float
z	**float**



z must be float since the result of the arithmetic operation is of type float. Also the *apply* node must be of type float

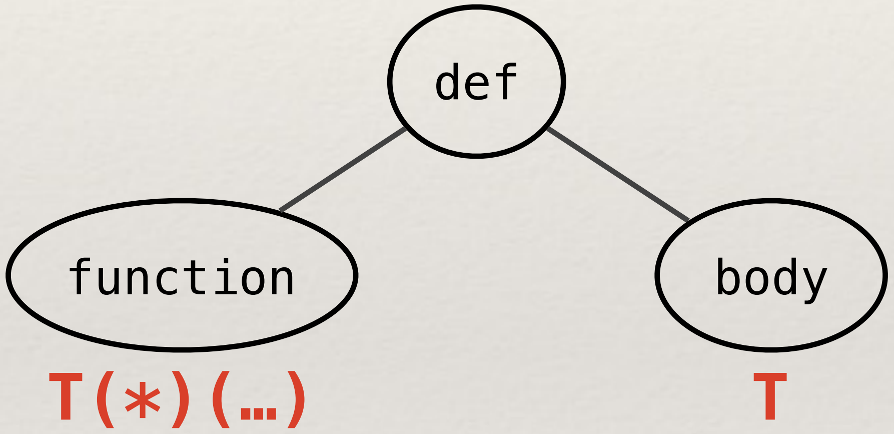
h	float(*)(int,float,float)
x	int
y	float
z	float

Call to *h* with the following argument types: float, int, float —> Type Mismatch

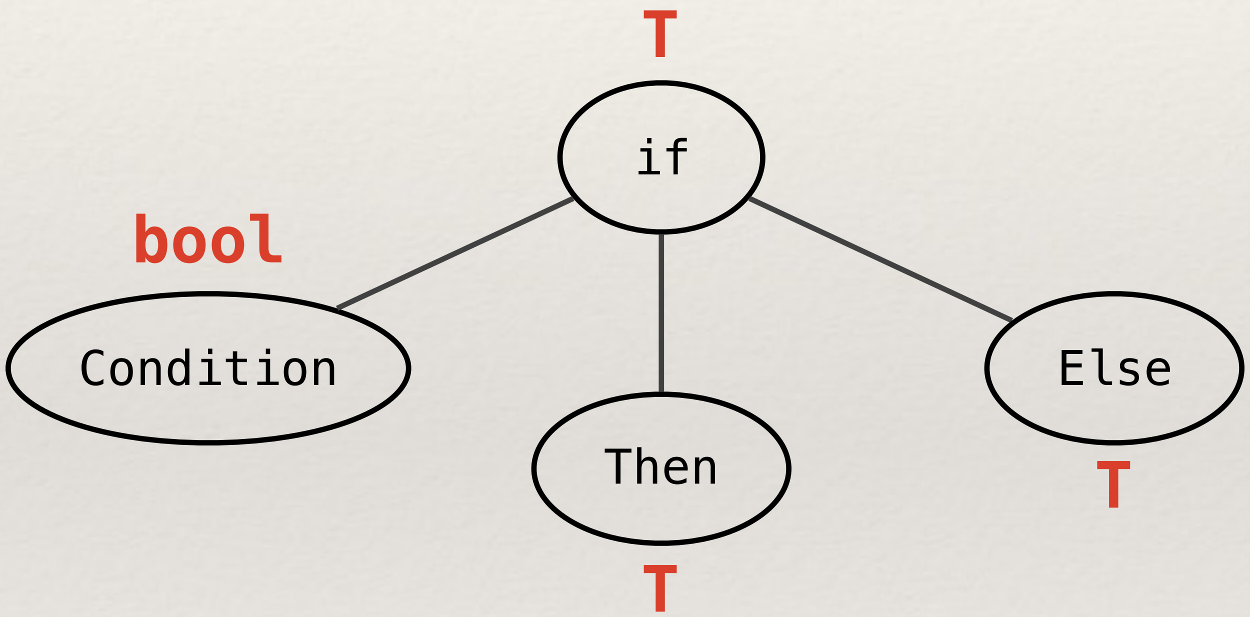


Type Constraints

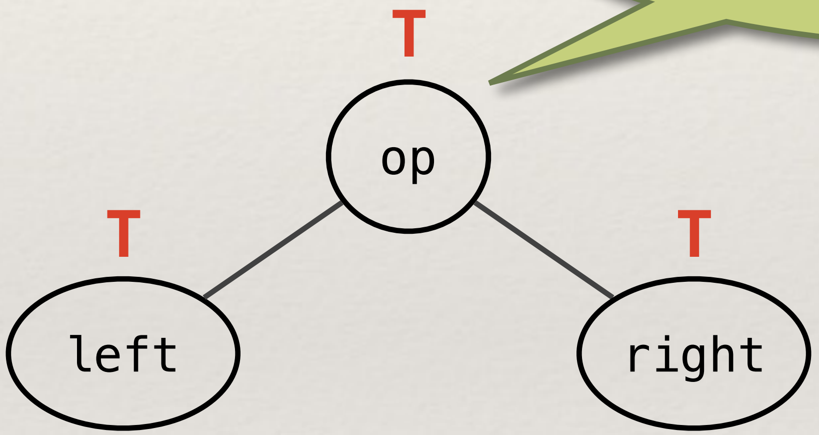
Function Definitions



If-Then-Else

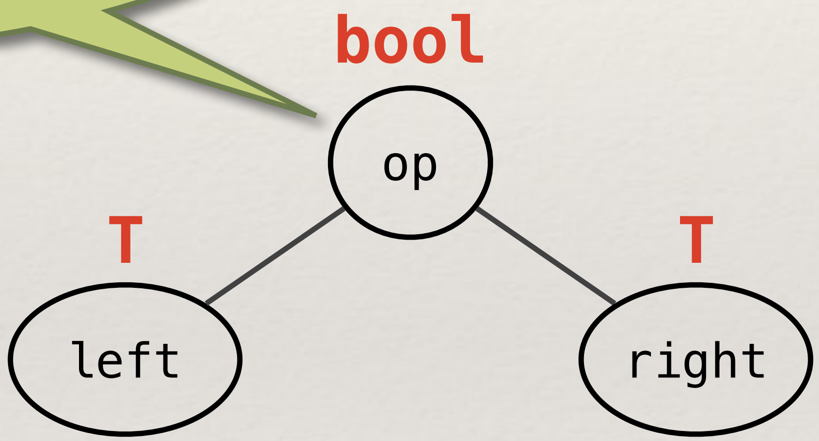


Arithmetic Expressions



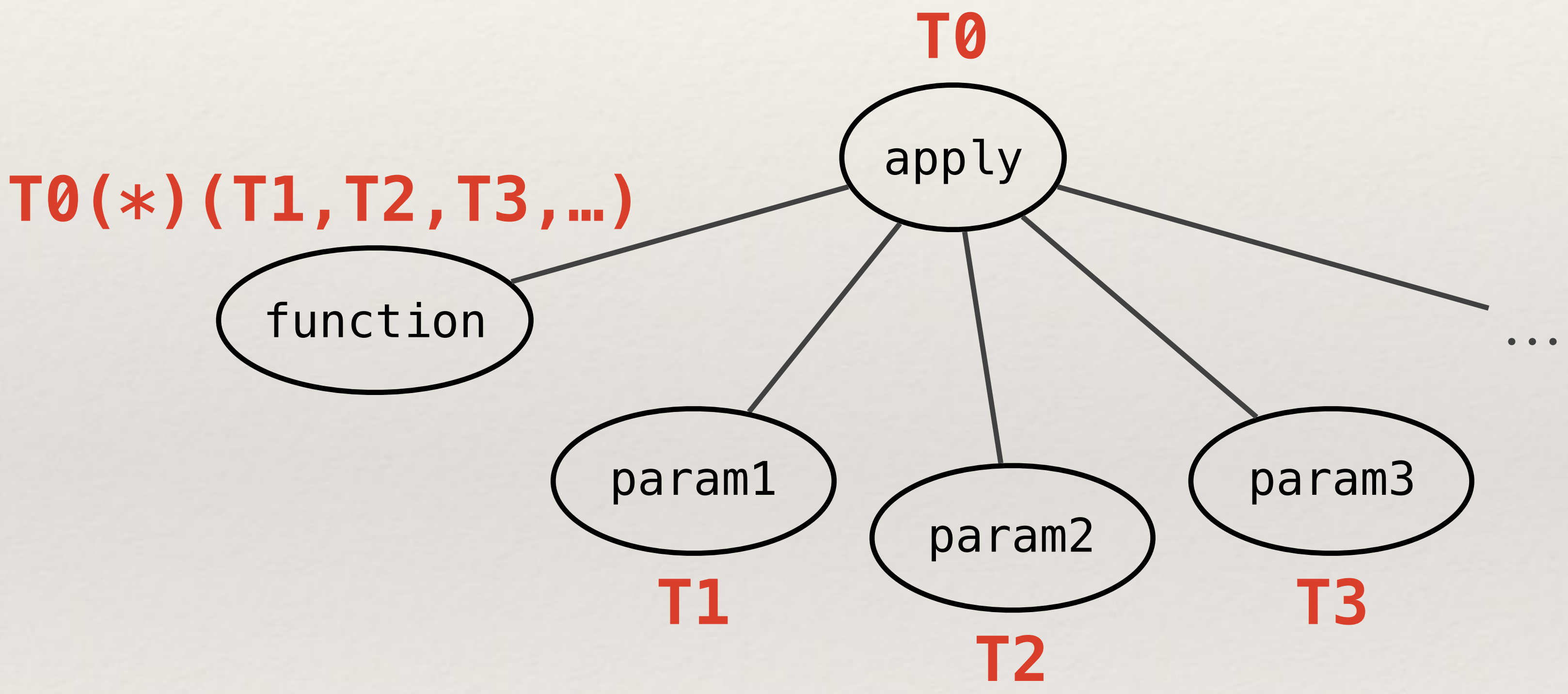
op must be an arithmetic operator: + − * /

Comparisons

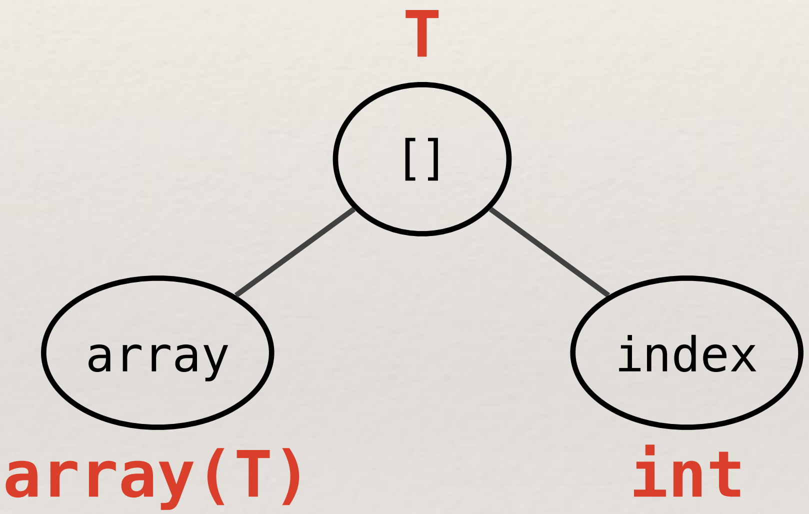


op must be a comparison operator: `<` `>` `=` `<=` etc.

Function Calls



Array Indexing


T
[]
array
index
array(T)
int