Semantic analysis requires:

- checking the **types of all expressions**
- showing the program satisfies
  all non-grammatical requirements

Let's consider the non-grammatical requirements first.

.

Three deal with **function names at the global level**:

- each function name is defined exactly once
- there **is** a user-defined function named `main`
- there **is not** a user-defined function named `print`

.

We can implement each of these checks with a loop over the list of function definitions that make up the program.

For example:

```
for p in list of defs
  if p.name = "print" then
     addError
       ("user-defined function named 'print'")
```

All three of these checks can be done with a single loop:

```
f_named_main = false
list_of_names = []

for f in list of function defs
    if f.name = "print"
       then addError
              ("user-defined function named 'print'")
    if f.name = "main"
       then f_named_main = true
    if f.name in list_of_names
       then addError("duplicate function named", f.name)
       else add f.name to list_of_names

if !f_named_main
   then addError("no user-defined function named 'main'")
```

•

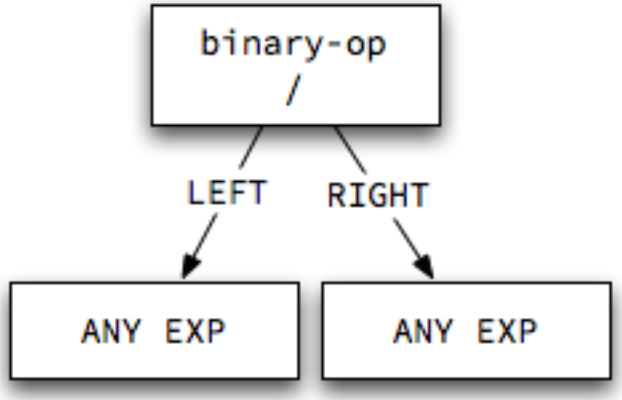
The other two deal with **names used within a function**:

- each parameter name
  is defined exactly once in a given function

- each function
  refers only to its formal parameters and other functions

We can implement the first with a **loop over the parameter list** and the second with a **tree traversal** of the function's body that validates each identifier and function call.

.



To type-check this or any other node, a semantic checker needs to...

1. Type-check its parts.
2. Apply a type rule to the results.

.

The type rule for divide and other binary arithmetic ops is:

```
if type(left operand) == integer
   and
   type(right operand) == integer

   then type of node = integer
   else type of node = error
```

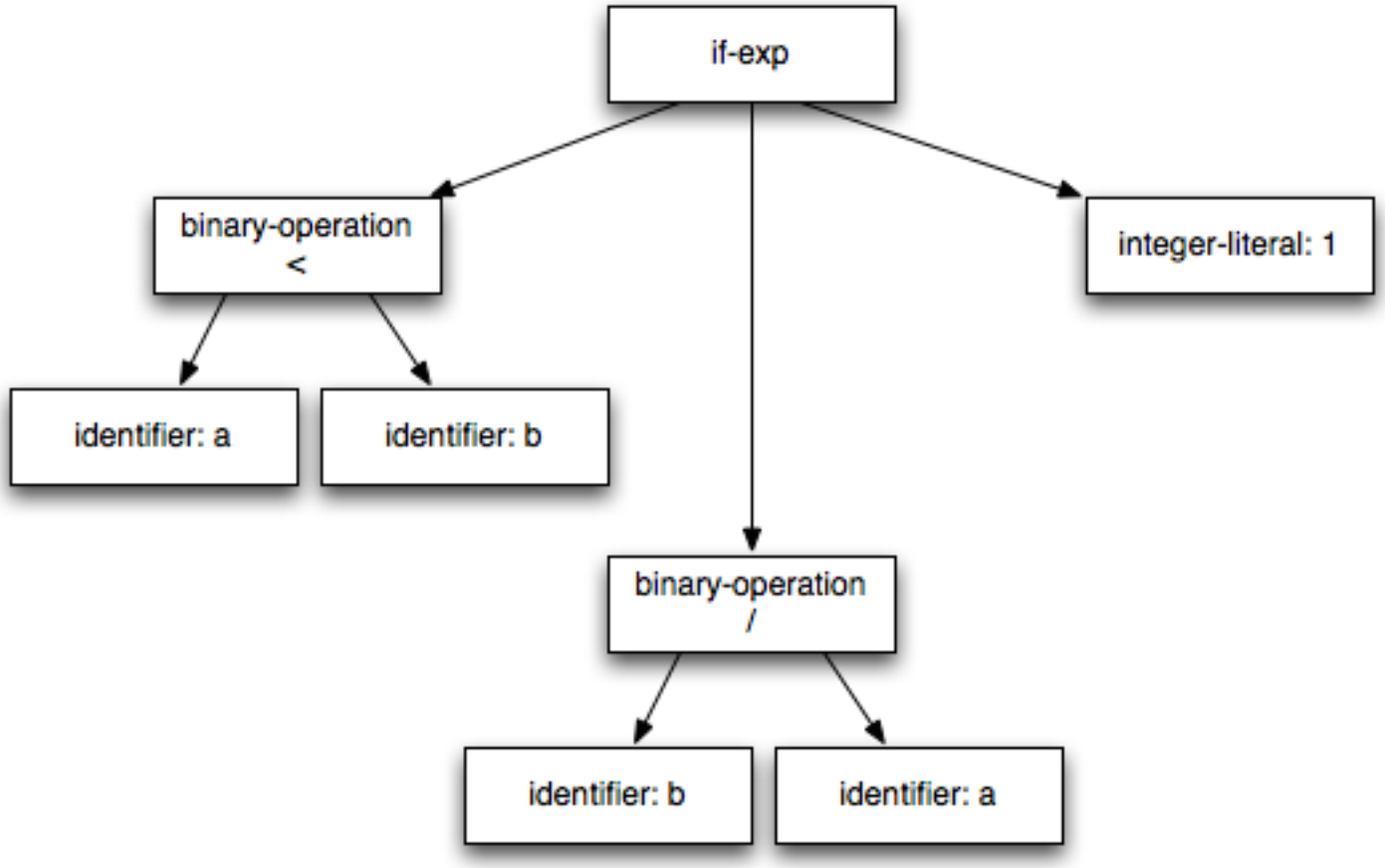
.

Consider this Klein **`if`** expression:

```
if (a < b)
   then b / a
   else 1
```

Its AST looks something like this:

.


if-exp
binary-operation
<
integer-literal: 1
identifier: a
identifier: b
binary-operation
/
identifier: b
identifier: a

Our type checker must:

- type check the condition
- type check the then clause
- type check the else clause

And then apply a rule:

```
if type(condition) == boolean
   and type(then) == type(else)
then
   type(if-exp) = type(then)
else
   type(if-exp) = error
```

Quick exercise:

What type expression might a semantic checker assign to this Klein `if` expression?

```
if (a < b)
   then b / a
   else false
```

What if we change `(a < b)` to `true` ?

.

Applying that type rule, the **`if`** expression from our exercise:

```
if (a < b)
   then b / a
   else false
```

has a type of error. We can never use this expression as the body of a function. It cannot guarantee to return an integer or a boolean.

.

But what of **this** expression?

```
print(
        if (a < b)
           then b / a
           else false
      )
```

.

Our Klein compiler **can** support this feature by assigning a new kind of type to the `if` expression:

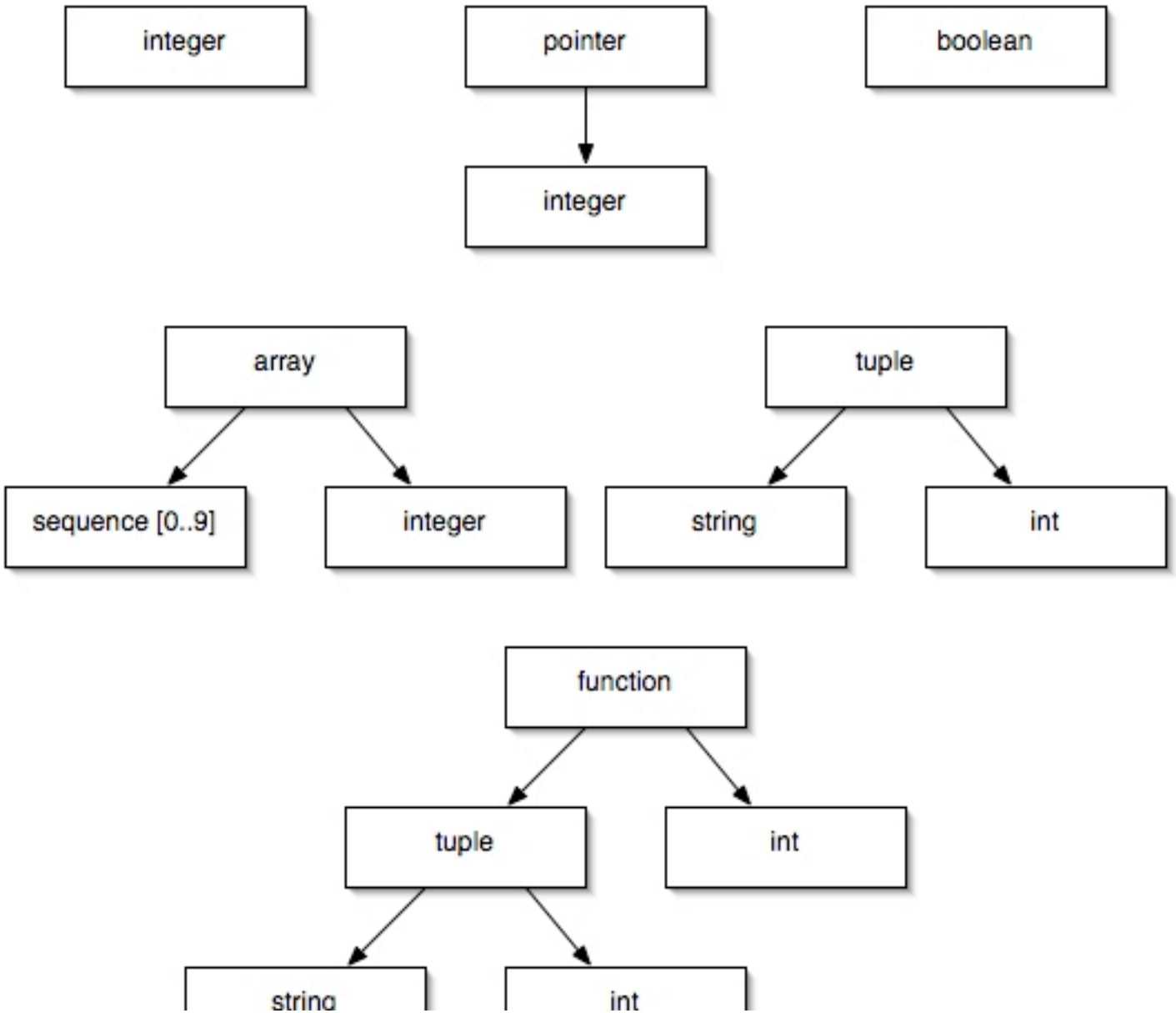
```
OR(type1, type2)
```

This is a new kind of constructed type.

It says an expression is **either** `type1` **or** `type2`.

.

Types have structure.


integer
pointer
boolean
integer
array
tuple
sequence [0..9]
integer
string
int
function
tuple
int
string
int

```
function divides(x: integer, n: integer): boolean
   MOD(n, x) = 0
```

The rule we need to type-check a **function definition** looks like this:

```
if type of body == type of return type
then
   type of function def
      = function(parameters type, return type)

else
   type of function def = error
```

.

```
divides( count(1, binary_n), n )
```

The rule we need to type-check a **function call** looks like this:

```
if type of function def  == function(s, t)
   and
   type of argument list == s
then
   type of function call = t
else
   type of function call = error
```