

## 5 Semantic Analysis

( 5. Semantic Analysis , p. 103)

- Starting with semantic analysis, the compiler phases may be characterized by (abstract syntax) **tree walkers**, each fulfilling a specific purpose.
- The **semantic analysis** phase walks the abstract syntax tree (AST) to test for two principal semantic properties of a correct Tiger program:
  - ① proper variable (and function) **scoping**,  
[ Record any variable (function) declaration, check if such a declaration is visible when a variable (function) is used. ]
  - ② **type correctness**.  
[ Record all user-defined types, check if functions are applied to correctly typed arguments and, if so, what is the type of the returned function result? ]

## 5.1 Symbol Tables

- Clearly, both tasks are not independent of each other (in Tiger, for example, each variable declaration also assigns a type to that variable).
- To keep track of the variable (type) declarations encountered so far in a program, the compiler uses **variable (type) environments** while it walks the AST:

- In a **variable environment**, an entry of the form

$$v \mapsto \tau$$

indicates: “variable  $v$  is declared and visible, the type of  $v$  is  $\tau$ ” (e.g., in Tiger:  
`let var  $v$  :  $\tau$  =  $e'$  in  $e$ ).`

- In a **type environment**, an entry of the form

$$t \mapsto \tau$$

indicates: “a type named  $t$  has been declared, and  $t$ ’s definition is  $\tau$ ” (e.g., in Tiger:  
`let type  $t$  =  $\tau$  in  $e$ ).`

- The **scoping rules** of the target language determine how semantic analysis updates the environments while it walks the AST.
- **Example:** trace the **variable environment**  $\sigma$  while the compiler walks the AST for the Tiger program fragment below:

**Variable environment trace**

1	$\sigma_0$	function f (a:int, b:int, c:int) =
2	$\sigma_1$	(print_int (a + c);
3	$\sigma_1$	let var j := a + b
4	$\sigma_2$	var a := "hello"
5	$\sigma_3$	in print (a); print_int (j);
6	$\sigma_3$	end;
7	$\sigma_1$	print_int (b);
8	$\sigma_1$	)
9	$\sigma_0$	

- $\sigma_0 = \{\}$  (empty environment)
- $\sigma_1 = \sigma_0 + \{a \mapsto \text{int}, b \mapsto \text{int}, c \mapsto \text{int}\}$  (body of f, **add** formal parameters)
- $\sigma_2 = \sigma_1 + \{j \mapsto \text{int}\}$  (variable declaration (let))
- $\sigma_3 = \sigma_2 + \{a \mapsto \text{string}\}$  ( $a \mapsto \text{int}$  **shadowed**)
- **undo** additions when leaving scope (end of let body, end of function body).

- The trace suggests that we require an environment implementation to offer the following functionality:
  - ① Given a (variable/type) name  $x$ , perform **lookup** in the environment for an entry of the form  $x \mapsto y$ ; yields either  $y$  or an error.
  - ② **Add** (+) a binding  $x \mapsto y$  to an existing environment, a prior binding  $x \mapsto z$  needs to be shadowed (after addition, a lookup for  $x$  yields  $y$ )<sup>18</sup>
  - ③ **Undo** a series of additions to restore the state of the environment (e.g., after a scope has been left).
- Two viable alternatives:
  - ① **Imperative** (or **destructive**) style:  $\sigma_{i+1} = \sigma_i + \sigma'$  destroys  $\sigma_i$ , we need additional undo information to recover  $\sigma_i$  later on.
  - ② **Functional** style:  $\sigma_{i+1} = \sigma_i + \sigma'$  leaves  $\sigma_i$  intact, undo simply means “*forget*  $\sigma_{i+1}$ ”.

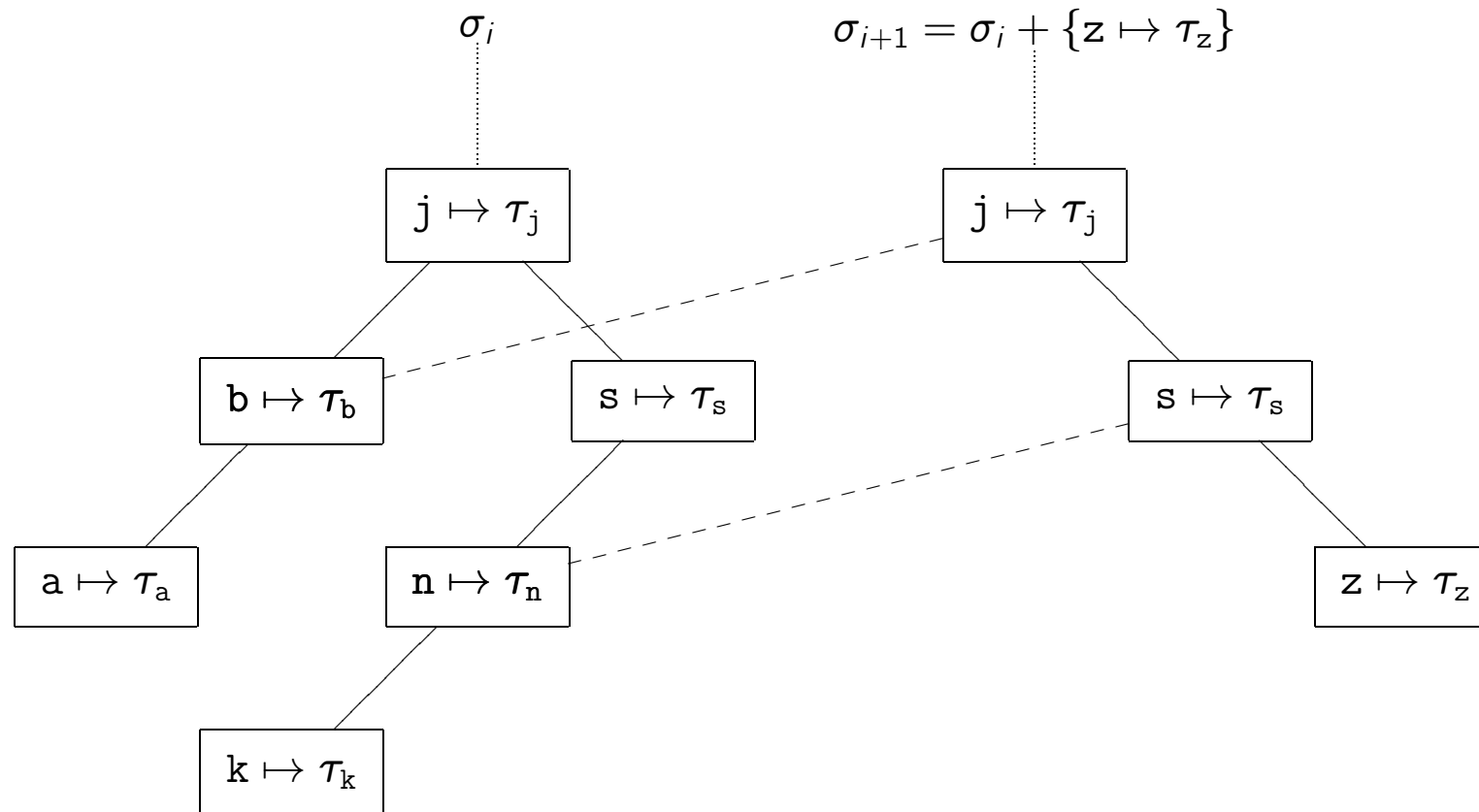
---

<sup>18</sup> $\sigma_1 + \sigma_2 \neq \sigma_2 + \sigma_1$

- **General idea to implement functional style:** maintain **binary search tree**:

- Add binding: copy only as many nodes as needed, otherwise **re-use** existing tree.  
Adding a binding works in  $O(\log n)$  time and space for a balanced tree of  $n$  bindings.

-  Does the functional style support shadowing well?



- **General idea to implement imperative (destructive) style:** maintain **hash table**:
  - Hash variable/type name and use hash table with *external chaining* to resolve collisions, during lookup walk chain from first to last to make shadowing work.
  - To implement undo, maintain an additional **undo stack** with *scope marks*.
- **Example:** let the hash function<sup>19</sup> simply be  $h(x) = (x - 'a') \bmod 3$ . We trace the undo stack and hash table for the Tiger program

**Tiger program fragment**

1	$\sigma_0$	let var a := 0
2	$\sigma_1$	var c := "foo"
3	$\sigma_2$	in
4	$\sigma_2$	let var c := 1
5	$\sigma_3$	var d := 2
6	$\sigma_4$	in
7	$\sigma_4$	a + c
8	$\sigma_4$	end
9	$\sigma_2$	end
10	$\sigma_0$	

<sup>19</sup>This hash function only supports single-letter identifiers.

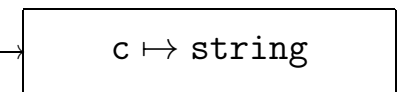
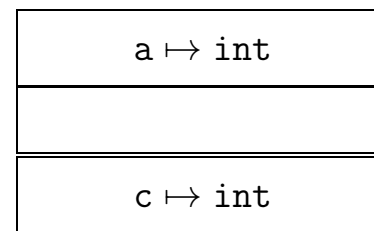
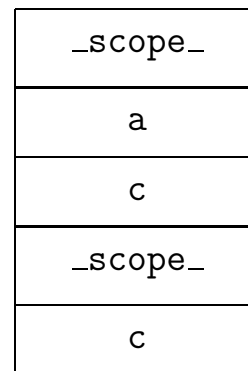
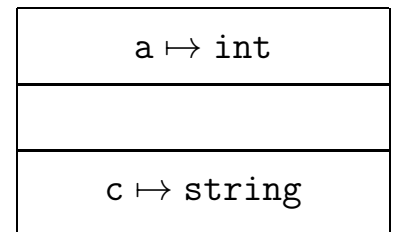
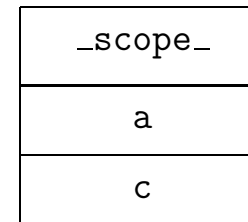
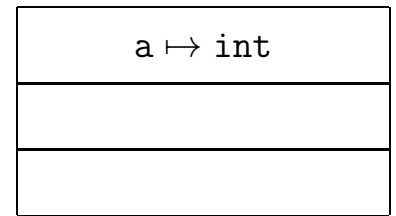
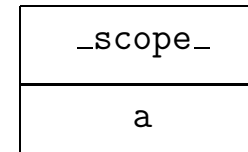
–  $\sigma_0$

–  $\sigma_1 = \sigma_0 + \{a \mapsto \text{int}\}$

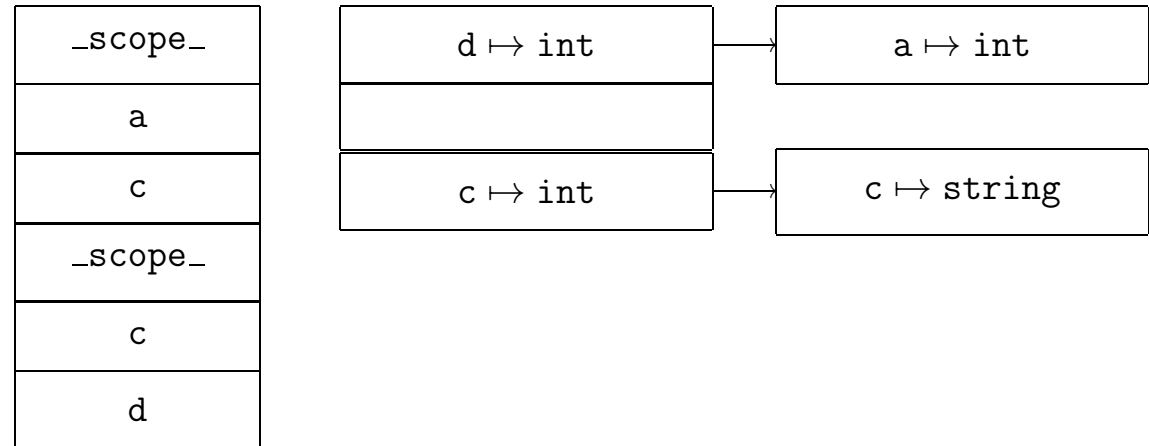
–  $\sigma_2 = \sigma_1 + \{c \mapsto \text{string}\}$

–  $\sigma_3 = \sigma_2 + \{c \mapsto \text{int}\}$

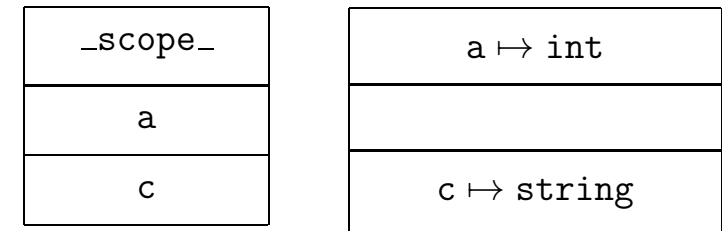
(undo stack empty)



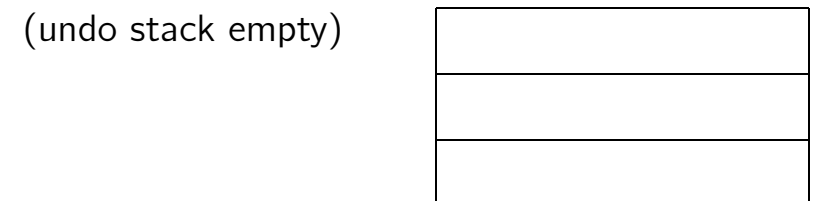
–  $\sigma_4 = \sigma_3 + \{d \mapsto \text{int}\}$



– first end (line 8, leave scope)



– second end (line 9, leave scope)



## • Remarks:

- To save the extra effort of maintaining the undo stack, simply link all bindings inserted into the hash table. To implement undo, start at binding last inserted, then follow links to find all bindings that need to be removed.



## 5.1.1 Symbols vs. Strings

- Although the previous discussion might suggest to, it is *not* efficient to use character strings (variable/type identifiers) as keys to implement symbol tables (environments):
  - Comparing strings for **equality** (e.g., during lookup) is inefficient (character-by-character),
  - **ordering** strings is inefficient (e.g., during binary search tree construction),
  - **calculating hash keys** may be costly, too.
- Instead, we map identifiers into **symbols** and make sure that the operation cost discussed above are cheap for symbols.

**Idea:** provide a function `Symbol(x)` that inserts identifier `x` into a hash table. Then simply use the **memory address** of `x`'s hash bucket as the symbol for `x`. On collisions, simply return the address of the bucket that is already in the table.

-  Cost for symbol equality tests, ordering, hashing (destructive environment impl.)?

## 5.2 Representation of Types in Tiger

- Now that we know how to represent identifiers internally, i.e., the  $x$  in the binding

$$x \mapsto \tau ,$$

we turn to the internal representation of the types  $\tau$ .

- The Tiger types  $\tau$  are

Type $\tau$	Remark
int	
string	
nil	nil is of any record() type
void	the type of statements returning no value
array( $\tau$ )	$\tau$ is the element type, array bounds not represented
record( $f_1 : \tau_1, \dots, f_n : \tau_n$ )	the $f_i$ are the record field names (symbols)

- A representation of this type structure in C:

#### Internal representation of Tiger types

```
1  typedef struct Ty_ty_ *Ty_ty;
2
3  struct Ty_ty_ {
4      enum { Ty_int, Ty_string, Ty_nil, Ty_void,
5              Ty_array, Ty_record, Ty_name }      kind;
6      union {
7          Ty_fieldList      record;
8          Ty_ty              array;
9          struct { S_symbol sym, Ty_ty ty; }      name;
10     } u;
11 };
```

- **Remarks:**

- Type kind `Ty_name` and union variant `u.name` is used to represent *named types*, e.g., the Tiger declaration `type length = int` introduces a new type `length` that is completely equivalent to `int`.
- `Ty_fieldList` is a linked list of `S_symbol/Ty_ty` ( $f_i/\tau_i$ ) pairs.

- According to the Tiger semantics, two record types are *distinct* even if they have identical field names and types (the same applies to array types).
- The following Tiger program fragment contains a type error:

**Tiger type error**

```
1  let type a = { x : int, y : int }
2      type b = { x : int, y : int }
3      var i : a := ...
4      var j : b := ...
5  in
6      i := j      ⚡
7  end
```

while the following program should compile fine (types a and b are equivalent):

**Correct Tiger program**

```
1  let type a = { x : int, y : int }
2      type b = a
3      var i : a := ...
4      var j : b := ...
5  in
6      i := j
7  end
```

- To implement these semantics, apply a technique similar to symbol representation:

**Check the identity of two types** simply by comparing the **memory addresses** of the struct `Ty_ty_` that represent the two types (i.e., compare the `Ty_ty` pointer values).

- According code snippet in Tiger's `types.c`:

```
types.c
1  struct Ty_ty_ tyint    = { .kind = Ty_int };
2  struct Ty_ty_ tystring = { .kind = Ty_string };
3
4  Ty_ty Ty_Int (void)
5  {
6      return &tyint;
7  }
8
9  Ty_ty Ty_String (void)
10 {
11     return &tystring;
12 }
13
14 Ty_ty Ty_Record (Ty_fieldList fields)
15 {
16     Ty_ty p = checked_malloc (sizeof (*p));    /* create new identity for this record type */
17
18     p->kind = Ty_record;
19     p->u.fields = fields;
20     return p;
21 }
22
```

- To reveal the actual identity of a named type (i.e., to detect that types a and b in the last Tiger fragment are actually identical), use a function like `actual_ty`:

```
                                actual_ty
1  Ty_ty actual_ty (Ty_ty ty)
2  {
3      if (ty->kind == Ty_name)
4          return actual_ty (ty->u.name.ty);
5      else
6          return ty;
7  }
```

- The type checking code of the semantical analysis phase will then contain idioms like

#### **Tiger type checker (excerpt)**

```
1      :
2      /* make sure that the two types t1 and t2 match */
3      if (actual_ty (t1) == actual_ty (t2)) {
4          ...
5      }
6      else {
7          /* type mismatch */
8          EM_error (...);
9      }
10     :
```

## 5.3 Type-Checking Expressions

- **Type-checking** actually comprises of two closely related tasks:
  - ① **Type-inference:** given an expression  $e$ , which type does  $e$  have?
  - ② **Type-checking:** is the inferred type  $e$  compatible with the typing rules of the target language?
- The actual type-checking process—as pointed out in the introduction to this chapter—is implemented as an AST walker.
- We check types while we walk the AST **bottom-up**: to check the type-correctness of a complex expression we need its subexpressions to be successfully type-checked already.

- The typing rules of a language are often given in the form of so-called **derivations** (“*Given that all premises hold, we can derive the conclusion.*”):

$$\frac{\text{premises}}{\text{conclusion}}$$

- For type-checking purposes, the *premises* make assumptions about the types of the subexpression of an expression  $e$ , while the conclusion infers a type for the complex expressions  $e$  itself.
- **Example:**<sup>20</sup>

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{\underbrace{e_1 + e_2}_e : \text{int}}$$

Read: “*If subexpressions  $e_1$  and  $e_2$  are both of type `int`, then  $e = e_1 + e_2$  is correctly typed and has type `int`, too.*”

-  Note how the usage of derivations implies a bottom-up type-checking discipline.

---

<sup>20</sup>The colon (`:`) is read as *has type*. The correct derivation is actually a bit more complex than shown here, we will fill in the gaps soon.



- The C code implementing this typing rule quite directly follows from the derivation:

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

### **Tiger type checker (excerpt)**

```
1  Ty typecheck (A_exp e)
2  {
3      :
4      switch (e->kind) {
5          :
6          case A_opExp: {
7              Ty t1 = typecheck (e->u.op.left);
8              Ty t2 = typecheck (e->u.op.right);
9
10             if (e->u.op.oper == A_plus) {
11                 if (t1 == Ty_Int () && t2 == Ty_Int ())
12                     return Ty_Int ();
13                 else
14                     EM_error (...);
15             }
16         }
17         :
18     }
19 }
```

- Once the typing rules of a language have been formulated using derivations, coding the type-checker is mostly easy.

– **Example:** Two more Tiger type derivations:

$$\frac{e : \text{int} \quad e_t : \tau \quad e_f : \tau}{\text{if } e \text{ then } e_t \text{ else } e_f : \tau}$$

$$\frac{e : \text{array}(\tau) \quad e_i : \text{int}}{e[e_i] : \tau}$$


-  Do you consider the if-then-else typing rule to be correct as it stands?

Consider

#### Tiger program fragment

```

1      :
2      let type pair = { fst: int, snd: int }
3      :
4      in
5          if x < 1 then nil else pair { fst = x, snd = 2 }
6      end
7      :
```

-  The *premises* may also be empty. When is this useful?

## 5.3.1 Type-Checking Variables

- The type-checking of a **variable reference**,  $v$  say, is dependent on the current variable environment: if a binding  $v \mapsto \tau$  is found in the environment, we can infer type  $\tau$ . Otherwise we need to report an `unknown variable` error or similar.
- We thus need to incorporate environments into our derivation notation. For the variable reference case we get

$$\frac{E + \{v \mapsto \tau\}}{v : \tau}$$

Read: “Given an environment  $E$  that (among others) also contains a binding  $v \mapsto \tau$ , the variable reference  $v$  has type  $\tau$ .”

- Once more, the C code immediately follows from the derivation. We now know that our `typecheck ()` function needs to pass the variable environment, `venv`, around:

$$\frac{E + \{v \mapsto \tau\}}{v : \tau}$$

### Tiger type checker (excerpt)

```

1  Ty typecheck (A_exp e, S_table venv)
2  {
3      :
4      switch (e->kind) {
5      :
6      case A_simpleVar: {
7          E_enentry binding = S_lookup (venv, e->u.simple);
8
9          if (binding && binding->kind == E_var)21 {
10             return binding->u.var.ty;
11         }
12         else EM_error (...);
13     }
14     :
15     }
16 }
```

<sup>21</sup>In the environment we maintain visible variables (`kind == E_var`) as well as functions (`kind == E_fun`).

- Another similar Tiger type derivation:

$$\frac{E + \{v \mapsto \text{record}(\dots, f : \tau, \dots)\}}{v.f : \tau}$$

### 5.3.2 Type-Checking Variable Declarations

- A `let` expression *introduces* a new variable binding. The typing rules is as follows:

$$\frac{E \vdash e : \tau \quad E + \{v \mapsto \tau\} \vdash b : \tau'}{\text{let } v = e \text{ in } b \text{ end} : \tau'}$$

Read: “If  $e$  has type  $\tau$  in environment  $E$ , and  $b$  has type  $\tau'$  in the enriched environment, then the whole expression has type  $\tau'$ .”

Read  $E \vdash e : \tau$  as “ $e$  has type  $\tau$  when the current environment is  $E$ .”

- The implementation of this derivation will call `typecheck ()` for  $e$ , then open a new scope, add  $v \mapsto \tau$  to the environment, call `typecheck ()` for  $b$ , and finally close the scope.

- If the `let-in-end` actually looks like follows:

`let v :  $\tau$  = e in b end`

we additionally need to make sure

- ① that the type of `e` is actually  $\tau$  (may be coerced to  $\tau$ ), and
  - ② if  $\tau$  is a named type, that this type is actually in scope (in the type environment).
- This suggests that we also need to pass the type environment to `typecheck ()`.
    - A type declaration like

`let type t =  $\tau$`

(with  $\tau$  being built from types in scope) simply adds `t` to the current type environment, `tenv`, say.

- Function `typecheck ()` will thus actually look similar to

**Tiger type-checker (excerpt)**

<pre>1 Ty typecheck (S_table venv, S_table tenv, A_exp e) 2 { ... }</pre>
---

## 5.3.3 Type-Checking Recursive Declarations

- To type-check **recursive** groups of function (or type) declarations, we need to exercise some care.

### – Example:

#### Tiger program fragment

```
1 let function f (x : int) : int = g (x / 2)
2   function g (y : int) : int = if y < 1 then 1 else f (y)
3 in
4   f (128)
5 end
```

In this situation, we cannot typecheck the body of `f` (because `g` has not been entered into the variable environment yet) and also cannot typecheck `g` (because `f` has not entered into the variable environment yet).

**Idea:** assume that typing will go alright and simply enter all functions of the recursive group into the environment *before* typechecking their bodies.

- A similar idea is applicable to (mutual) recursive type declaration groups.

– **Example:**

**Tiger program fragment**

```
1 let type tree    = { node: int, children: forest }
2   type forest = { head: tree, tail: forest      }
3   in
4     ...
5 end
```

**Idea:** before checking that all type names (here: `tree`, `forest`) are actually in scope, simply enter all types of the recursive declaration group as *named types with empty (NULL) declarations* into the type environment, then check the right-hand sides of the type declarations.