

Compiler Construction

~ Further with binder ~

Problem 1: simultaneous symbol definition

How to deal with simultaneous and identical identifiers?

Simultaneous symbol definition

Many different `t`, including several
“variables”. **t-time**

```
let
  type      t = { h: int,
                 t: t }
  function t (h: int,
             t: t) : t =
    t { h = h, t = t }
  var      t := t (12, nil)
  var      t := t (12, t)
in
  t.t = t
end
```

Solution

Maintain several active environment
at once

- One symbol table for variables
- One symbol table for types
- One symbol table for functions
- ...

Problem 2: Memory Management (1/2)

When do you deallocate data associated to a scope?

For statically scoped languages

- many traversals check **uses** against **definitions**
- most traversals need a form of memory (binding, type, escapes, inlining, translation, etc.)
- this memory is related to scopes

Problem 2: Memory Management (2/2)

`scope end` deallocate everything since
the latest `scope_begin`

`pass end` deallocate auxiliary data
after the traversal is
completed

`ast` bind the data to the AST and
delegate deallocation

`by hand` thanks God for Valgrind and
Paracetamol

`never`

Problem 2: Memory Management (2/2)

`scope end` deallocate everything since
the latest `scope_begin`

`pass end` deallocate auxiliary data
after the traversal is
completed

`ast` bind the data to the AST and
delegate deallocation

`by hand` thanks God for Valgrind and
Paracetamol

`never` `--`

Memory Management: Deallocate on scope exit

```
let var foo := 42
    var foo := 51
in foo end
```

```
let var foo := 42 in
let var foo := 51
in foo end end
```

But then...


```
let type rec = {}
in rec {} end <> nil
```

Segmentation violation...
Courtesy of Arnaud Fabre.

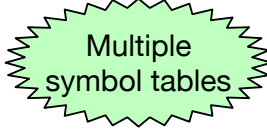
Memory Management: Deallocate with the AST

- annotate each node of AST
- annotate each scoping node with a symbol table and link them
- leave tables outside

Summary



Memory
management



Multiple
symbol tables