UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

TELECOM ParisTech

Sequence 3.2 – The binder

P. de Oliveira Castro S. Tardieu

The binder

- The lexer and the parser, together, analyze the source code and check that it adheres to the grammar of the Tiger language.
- The parser then builds an abstract syntactic tree corresponding to the source code. This phase is named the *syntactic analysis*.
- At no point so far has anyone checked that variables used in the program were properly declared.

Checking that variables have been properly declared and linking their usages to their declaration is the role of the *binder*. This is part of the *semantic analysis* phase, which assigns meaning to the program.

Using variables in the AST

- In the AST, a `VarDecl` node is used to represent a variable declaration.
- Similarly, a `Identifier` node is used to represent a variable usage.
- The binder is a visitor walking the tree to build scopes as `let` expressions and function declarations are encountered, and register `VarDecl` nodes in the innermost scope.
- At the end of a `let` expression or a function declaration, the innermost scope will be discarded.
- The binder will look up variables used in `Identifier` nodes by looking first in the innermost scope and going up as needed. It then stores a reference to the right `VarDecl` node in every `Identifier`, or give an error if the variable cannot be found.

Representing scopes

- The binder maintains a stack of scopes.
- Each scope contains a mapping of variables names to their corresponding `VarDecl` node.
- When a new scope is created, it is pushed at the top of the stack.
- When a scope terminates, it is popped from the top of the stack.
- The stack is searched from the top (innermost scope) to the bottom (outermost scope).

This construct is also called a *chain map*, as mapping are chained and searched in order.

A simple example

The current scope stack is represented in comments (top of the stack is on the right side). Stack is shown as [], and mappings as {}. We represent declarations as their line numbers for simplicity.

```
1  /* [] */
2  let /* [{}] */
3    var a := 1 /* [{a => L.3}] */
4    var b := 2 /* [{a => L.3, b => L.4}] */
5  in
6    let /* [{a => L.3, b => L.4}, {}] */
7      var a := 10 /* [{a => L.3, b => L.4}, {a => L.7}] */
8    in
9      ... /* If a is looked up, result will be L.7 */
10   end /* [{a => L.3, b => L.4}] */
11 end /* [] */
```

What about functions?

- The binder binds every `Identifier` to its corresponding `VarDecl`.
- Functions are given a similar treatment: the binder binds every `FunCall` (a function call) to the corresponding `FunDecl` (a function declaration) by looking up the name of the function in the scopes stack.

After the binder pass

- After the binder has walked the tree, the AST is *decorated* with extra information.
- In subsequent passes, we will no longer use variable names as those can be overloaded when a variable is masked by another with the same name.
- We will instead use the `VarDecl` nodes as the identifier of every variable: if two identifiers point onto the same `VarDecl`, they denote the same variables, Otherwise, they denote different variables, possibly with different types.
- Similarly, we will no longer use the function names, but the `FunDecl` nodes which uniquely denote a function.

Conclusion

- The binder binds variable uses and function calls to their current declaration according to the language semantics.
- In doing so, it detects uses of non-existing or non-visible variables and functions and can report the error to the user.
- After the binder pass, we use the declarations as identifiers for the variables and functions.