

CS350 Lab 5: Type Checking

Change the definition of `which_cs350_lab` in `tiger.cc` so that it has the value 5; then add type checking to your tiger compiler in two stages:

- A) Add symbol tables that simply record the sets of names declared and available at each node, and identify cases of undeclared identifier or multiply declared functions/types (as per the “local redeclarations” section of Appel’s Appendix A).
- B) Adapt your symbol table to record the type information about each variable (using Appel’s `types.h` and `types.cc` or your own method of representing types), and then synthesize a “type” attribute for each expression and print messages for each type error (such as `"hello" + 3`).

The type checking process will be started with a call to the `typecheck` function in `typecheck.cc`, with the AST root as the parameter. You should modify this function to actually perform type checking, e.g., by calling functions to evaluate the attributes discussed in lecture.

You can send attribute information around the AST in any of several ways — for any of these, you’ll need to modify the variable `OBJ` in the `Makefile` in your `Debug` folder to list any new “.cc” files you create, as well as of course declaring the functions in `AST.h` and writing them in your .cc files. Ways to communicate attribute information include:

- Use my functions for synthesized attributes (already in your tiger project) and my functions for inheriting attributes (copy the files `pinherit.cc` and `pinherit-implementation.h` from the `Lab5Files` folder in the `cs350 Examples` folder). Then implement each attribute as a (possibly public) data member in one or more AST classes (you’ll have to edit `AST.h`), and create a set of virtual functions for the AST classes, to define each attribute (by editing `AST.h` and putting the function bodies in `typecheck.cc`). This approach will receive full credit (assuming you do all the parts of the Tiger language).
- Use my synthesized attribute functions, but write `pinherit` yourself, and then create attributes as in the previous approach. This approach will get bonus credit, but if you choose it, make sure you explain this in the `Design-Documents` folder.
- If you wish, you may use any other system to handle attributes, or even do type checking without them, but please give an extensive description of what you did. Note that we’ll be describing other phases of compilation in terms of attributes, so it will be good if you have some way to make use of this framework. This approach will receive full credit if you get it working for the whole Tiger language, but note that I won’t be able to give you much help, and I won’t give you extra credit for just figuring out something that’s specific to type checking. If you can do something novel and explain to me how it addresses the general problem of attribute grammars, I may award bonus credit for that.

Your symbol table class can be

- your symbol table from CS245,
- the symbol table class defined in `ST.h` and `ST.cc` (this similar to what we used in CS245),
- a more advanced symbol table class from the C++ standard library, or one you download from some other source (which you must cite properly in a comment), or
- a more advanced symbol table data structure you create yourself, such as a hash table or a balanced tree (such as a “2-3 tree” or “red-black tree”) — the balanced tree approach will probably fit better with the “pure functional” approach we’re using in class.

You may represent types using

- Appel’s `types.h` and `types.cc`, or
- your own representation.

Record important design decisions in a file `Lab5-TypeChecking` in the `Design_Documents` subdirectory.

For 75% partial credit, do all features of Tiger except mutually recursive function declarations, arrays, records, type declarations, and implicitly typed variables; for 85%, add mutually recursive functions; for 100%, add arrays, records, and type declarations. If you can get implicitly typed variables working too, that will count for bonus credit (I *believe* it will require yet another kind of tree traversal).

STRATEGY HINTS

- Take this lab in steps — start with 5a, and separate 5a into pieces that you can create and test. For example, you *could* start by writing a set of functions to synthesize the names declared in a set of declarations, and have it use `EM_debug` to print messages as it goes along. Once this compiles, you could test it on a simple example to make sure it is really running and produces the result you expect (remember to use the “-d” option to make your compiler print debugging information). After that, you could think about what would be needed to detect errors involving multiply-defined symbols (e.g. two variables named `x` in the same `let`), implement this, and then test it by running your tiger compiler on a program that contains such an error. After you get working code to synthesize the information you need from declarations, you can at last start to think about sending information about available variables down into the expressions — this will require my `pinherit` functions or some system of your own devising.
- The somewhat “aspect-oriented” approach we’re using is very different from the imperative approach described in Appel’s Chapter 5 — I recommend you read his discussion of symbol table data structures in Chapter 5.1, but stop when he describes his files for Tiger, and from that point on, just skim the chapter to pick out anything about the semantics of the tiger language (such as the fact that it uses “name equivalence” rather than “structural equivalence” for type compatibility, discussed at the start of Chapter 5.2), but ignore the discussion of how to implement this in C.

NUTS AND BOLTS — Remember to...

- have `typecheck.cc` call upon the functions you create to do the type checking
- add the names of all the “.cc” files you create to the `Makefile` (in the `Debug` folder), by adding the file name (with “.cc” turned into “.o”) to the line that starts `OBJ=`
- add a line to the `DECLARE_ALL_SYNTHESIZED` macro in `AST.h` for each new data type (other than `int` and `bool`) that you need to use for a synthesized attribute (and remember that *every line of a macro definition except for the last one must end with a backslash* — be careful about blank spaces after this backslash, since these will mess it up!)
- add the appropriate lines to `synthesize.cc` for each new type you add to `DECLARE_ALL_SYNTHESIZED` (just do what is already done for `int` and `bool`)