Project 3: Tiger Language Type Checker

Dong Son Trinh
Maxim Mints

High-Level Type Checker Architecture:

The type checker makes use of the AST created for Project 2. An AST generated by the parser can be passed to the TypeChecker class (p3/src/com/tiger/type/TypeChecker.java), which can then be used to verify that the program is well-typed. The analysis makes use of the SymbolTable class (p3/src/com/tiger/type/SymbolTable.java). The SymbolTable class contains a stack with key-value pairs of names and corresponding types, which are represented with the TigerType class (p3/src/com/tiger/type/TigerType.java).

A TigerType can represent the basic types (boolean, int, float, unit) as well as arrays and functions. The latter two are represented using a list of type parameters inside TigerType: an array type is defined by 1 type parameter, while a function is defined by n type parameters for parameter types and one more for the return type.

The SymbolTable exposes operations to push a name-TigerType pair onto the stack, remove the top entry from the stack, and find the type from the last-added name-TigerType pair with a given name. It is used to represent the type alias map (type names to actual types) and the type context (var names to their types) of the program. Strictly speaking, since there is only one type alias scope, the stack isn't really needed for it in this specific case. Also, for the case of Tiger, the only time that something has to be popped from the type context is when we finish processing a function scope and pop its parameters (which are the only local variables allowed in functions, although they can still "hide" the variables from vardecls).

The type checking process works top-down: it starts by attempting to check if the entire program is well-typed, then goes through the typedecls, the vardecls, the funcdecls and the program statements, recursively attempting to prove they are well-typed as well, and using the results to make decisions about the well-typedness of the program as a whole.

Low-Level Design Decisions:

Type-checking of each program element is performed by a single dedicated method in TypeChecker, which recursively calls itself and/or other methods on subsections of the AST. The method which handles the program as a whole starts by going through the typedecls, the vardecls and the funcdecls in a loop, pushing them onto the type alias map (for the first one) and the type context (for the last two), respectively. In order for the type alias map to be built correctly, it is initialized with the basic types, so that subsequent type declarations (including the recursively-discovered array element types) could find them by name in the type alias map It then calls the method which processes statement lists. That, in turn, goes through a list of statements, tracking the type that a function may return. If, at some point, a statement must return a value of some type, that list of statements is no longer processed. If it is found that a statement may or must return a type which is different from a type that some previous statement may return, the sequence of statements is considered to not be well-typed. For this, the MayReturn class, an inner class of TypeChecker, is used. Also, statements and variable assignments both make use of expressions, so there is a separate method for handling them as well. Recursively calling itself on its children until an operator is reached, it compares the types of its operands with the expected types for the operator. For literal values, another method processing the const nonterminal node is used, which matches literals and identifiers with a type (in the latter case using the type context).

Challenges and Issues:

The main issues had to do with the implementation of the AST. It was not implemented with type-checking in mind, and the data structure had to be heavily modified to allow for a cleaner type checking implementation. The biggest problem was processing lists of elements, such as lists of function parameters or lists of statements. These are given somewhat inconsistently in the grammar, and accessing the necessary terminals and nonterminals can sometimes be tedious. This also makes verification of the type checking code's correctness quite hard.

Testing:

The type checker has been tested against the test cases provided on GitHub. We also chose several of the provided Project 1 and 2 test cases. We deemed them satisfactory as they cover most of the cases. We manually sorted them to tell apart well-typed and non-well-typed programs. Then Dong Son wrote scripts, which can be found in p3/test/, to automate testing of all 3 parts of the project using test cases from among all of those provided for this and the previous projects. This was done as integration testing to ensure that the changes required for the implementation of Project 3 did not break any features from the scanner or the parser.

Unresolved Issues:

None that we could find.