Programming Assignment 3 - Implementing Semantical Analysis

Collaborators

- Michael Maitland, mtm68
- Scott Bass, sb2383
- Michael Tobin, mat292

Running the Program

Main class: `mtm68.Main.java`

Run `xic-build` (this requires having Maven installed). Then `xic [options...] <source-files>` becomes available.

Summary

The most interesting part of our implementation is the leveraging of the Visitor design pattern. We spent a lot of time reasoning about how it should work as well as understanding how the polyglot visitor worked (as inspiration). This is tremendously important as not only will our type checker completely rely on it but also any other passes in the future. The most challenging part about it was reasoning when and how nodes should be copied in order to preserve tree immutability. After solidfying our structure, the type checking rules themselves were not too difficult to implement as the type spec was very explicit as to what is or is not valid. We also had to make a number of decisions as to how different errors should be handled and reported to the user.

Specification

The following are choices we made regarding specification...

One choice is that indexing into an empty array initializer will not pass type checking (i.e. {}[0]). We made this choice as allowing it would cause a decent deal of redesign in our current type implementation and it semantically does not make sense anyway. Such an access would always lead to an out of bounds exception down the line, therefore, having it fail during semantic analysis should not have any bearing on user convenience.

Another choice we made is how and when to report errors for inconsistencies in context bindings (whether strictly within the source file or between a source file and interface). First, if a library interface is not found or not lexed/parsed correctly, this is considered a Semantic error in type checking. We also decided that mismatches between function declarations in source files or interfaces should throw an error immediately at the location of either the Use statement (if it is an interface - interface problem) or at the function definition that has the mismatch

(if it is a source file to interface problem). We decided that type checking should end immediately as it cannot continue if there is any ambiguity in the context. We also end the process if any variable is declared under a name that is already in scope. Otherwise, we report non-fatal errors and continue type-checking.

Another choice related to error handling is how to handle the case "_ = 2". The provided typecheck tests have this fail as a semantic error, however, we handled this at the parsing stage. Therefore, we throw a syntax error rather than semantic. This is sensible as the "_" can only be used in the left hand side of a function expression which we can detect during syntax analysis

Another thing we would like to mention is related to a single failing test case from pa1. We are failing the test case `"""` because we report line:column pair `1:3` while the tests expect `1:1`. We are lexing where the first two quotes make up the empty string and then a dangling single quote. The dangling single quote is character `1:3` which we are correctly reporting. The test case expects `1:1` because its interpreting the input as a string with an unescaped double quote in the middle. Regardless, both report this correctly as an error.

Design and Implementation

Architecture

The key classes and packages we created or updated for this assignment are the following...

- mtm68.Main.java
  - Our Main functions very similarly to the previous assignment. We added new options to the command line as described in the spec. We also added type checking to the source file pipeline.
- mtm68.SymbolTableManager
  - This class organizes interface imports and merges all relevant interface function declarations into a symbol table. It maintains a map of "use ids" to maps of "function names" to "function declarations". By organizing it this way, we do not have to lex and parse the same interface files repeatedly if multiple .xi files use them. The info is stored in this map and can be accessed multiple times. Here, there are also checks to ensure that functions with the same name are consistent across multiple interface files. Upon given a program, the SymbolTableManager accesses each Use object and either looks up the .ixi file in the libpath or pulls the info from .ixi files that were passed as source files to the compiler.
- mtm68.visit.Visitor.java
  - We decided to use the Visitor pattern for AST passes. This class is the very basic parent of all visitors we create. For each visitor we create, we add a corresponding method to the Node superclass which requires all nodes to implement their own version of it. Our visitor does an immutable traversal (inspired by polyglot).

- mtm68.visit.Visitor.TypeChecker
  - This is the visitor that actually carries out the type checking of the AST. The TypeChecker "enters" a node, visits its childrens, and then typechecks as it performs "leave". We typecheck in the leave method as, at this point, all of the children will have been typechecked and tagged with their corresponding types. Therefore, we have all the info we need to typecheck the current node. The TypeChecker is given the initial symbol table generated by SymbolTableManager (above) and FunctionCollector (below). From there the TypeChecker constructs and maintains a TypingContext (below) to access as outlined in the type spec. As the TypeChecker visits nodes, it maintains a list of errors that is added to whenever there is a non-fatal type error. This allows us to continue checking even if there are errors. Fatal errors are errors generally where bindings are ambiguous in the context.
- mtm68.visit.FunctionCollector
  - As mentioned in the type spec, type checking requires two passes as functions can be mutually recursive. The FunctionCollector is a Visitor who's purpose is to simply collect function declarations and add them to a symbol table prior to typechecking. This collection happens after SymbolTableManager merges interface tables. The FunctionCollector collects errors (similar to TypeChecker) if a function is declared multiple times in the source file or there is a mismatch between a source file function declaration and an interface function declaration. The FunctionCollector then merges the table it creates with the table from SymbolTableManager which creates the starting typing context.
- mtm68.types.TypingContext
  - This class maintains the typing context for type checking. It is instantiated with the symbol table generated by FunctionCollector. It maintains a stack of Maps from ids to ContextTypes (below). It has a number of documented helper methods to interface with it. It supports entering and leaving scopes by pushing and popping off the stack respectively.
- mtm68.types.ContextType
  - A ContextType stores type information as it is used in the TypingContext. A ContextType can fall into two categories. The first is a simple variable to type binding. The other is a function name to arg types and return types.

Code Design

- For this assignment, we highly leveraged the Visitor design pattern in order to create AST passes. We were inspired by the polyglot visitor and created a similar Visitor that allowed for lazy tree creation. We believe this will be extremely useful in later assignments when we have many passes to implement (whether it be representation translation or optimization). Having an immutable copy of the tree after each pass will allow for easier

debugging.
- As discussed in lecture, a key data structure we used in this assignment is a stack (or techincally a Deque) of maps for our typing context. We decided to go this route for its ease of implementation despite it not being the most efficient method discussed in class. Maps, in general, were very important in this section due to the number of cases they were suited for (discussed in architecture section).

Programming

- Luckily this time, we did not run into any build errors. The most challenging aspect was organizing our code and AST in order to accomodate the new Visitor pattern we were using. This required analyzing each node and how its children should be visited. We also had to concern ourselves with copying nodes in order to preserve the immutability of our AST (which we got wrong a few times at first).
- The following is the team coding/responsibility breakdown for this assignment...
  - **Tobin:**
    - parser fixes + tests
    - SymbolTableManager
    - FunctionCollector + tests
    - TypingContext
    - Function TypeChecking + tests
  - **Maitland:**
    - lexer fixes + tests
    - Visitor + TypeChecker
    - Expr typechecking + tests
  - **Bass:**
    - TypeChecker
    - Statement typechecking + tests
    - Error output/handling
- We leveraged our lexer and parser code from previous assignments. We did have to make a few fixes but most were due to pretty print differences or error location reporting.

Testing

In this assignment, we generally followed a test-driven development cycle for implementing type checking. This involves making failing test cases and then adding the code to make them succeed. This allowed us to ensure our implementation is correct from the bottom-up. It also is a lot easier to consider edge cases as you build from the ground up rather than having to consider an entire type checking system as a whole. This also gives us great confidence that we have good coverage of the input space as each test was designed to cover all possible cases under which a type checking rule could be evaluated. To design this tests,

we looked closely at the type spec and made sure that the TypeChecker would pass under all appropriate situations and fail otherwise.

In order to carry out this plan, we leveraged a lot of JUnit tests that carried out type checking and allowed us to assert results of the TypeChecker visitor or assert errors if appropriate.

We also added tests to our Lexer and Parser PP suites to address the errors that came up during the grading of the last assignment.

This test plan helped us catch small bugs in implementation. For the most part, since the code was test-driven, we were able to catch most errors very quickly which prevented them from turning into a confusing headache later on. It was also a lot easier to reason about bugs since we were designing the code incrementally.

Work plan

For this assignment, the main work was implementing all the type checking rules. We ended up splitting it up by types of Nodes (Exprs, Stmts, and Functions). In hindsight, this might not have been the most optimal plan as some nodes in one person's jurisdiction depended on other nodes outside of their responsibility. For the most part, the rules were able to be implemented in isolation but some tests were held up due to our parallelization. Outside of the typechecking rules, other tasks were generally isolated (fixing parser/lexer, error output, symbol table generation, etc.) and were easy to split up.

Known Problems

We currently are not aware of any issues with our compiler.

Comments

NA