Compilers Fall 2015 Project 3

Due We 11/11/2015 at 8:30am.

Project Milestone 3: Semantic analyzer

Implement a semantic analyzer for TACK. Your program should parse a TACK file and construct an AST. Then, it should do semantic analysis. If there are no errors, your program should print the symbol table. As an example, given the following input TACK code:

```
# Scopes and symbols
f = fun (p : int, q : int) -> void {
  v = p + q;
  for i in [1,2,3] {
    v := v + 10;
  }
  { v = 0; }
  print(v : string);
}
main = fun () -> int {
  r = (g = 4, h = 5) : (g : int, h : int);
  f(10, 10);
  -> 0;
}
```

Your program should print the symbol table with the following format and indentation:

```
Program {append, bool2int, bool2string, f, int2bool, int2string, length, main, newArray, newRecord, print, range, size, string2bool, string2int, stringEqual}
  FunDef f {p, q, v}
    RecordType {p, q}
    ForStmt i {i}
    BlockStmt {v}
  FunDef main {r}
    RecordType {}
    RecordLit {g, h}
    RecordType {g, h}
```

Each line of output shows one scope. The symbols in each scope are printed in alphabetical order on the same line as the scope. Nested scopes are printed with two extra spaces of indentation under their parent scope. Note that the `Program` scope contains all function symbols, including intrinsics as well as user functions.

In case of semantic errors, you can format the output however you like, but you need to return a non-zero exit code (e.g., with `System.exit(-1)` in Java). Your semantic analyzer should check for all the errors cases in §3 "Scope Rules" and §5 "Type Rules" of the TACK specification.

The following archive file contains input TACK programs and corresponding outputs:

http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/pr3/test.tar

Please turn in the entire, self-contained code for your semantic analyzer, including a `README` with instructions for how to compile and run.

Hints and tips

Here is a recommended implementation strategy:

1. Get started early.
2. Read the scope and type rules in the language specification:
   `http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/tack-spec.pdf`
3. Reuse code from the previous milestone. You can either reuse your own code, or use the example solutions, as you prefer.
   `http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/pr2-example-solutions.tar`
4. First, write the scope analyzer. Start by getting a simple test working, then incrementally implement more cases until you cover all tests.
5. When you are done with the scope analyzer, write the type analyzer. Follow the same incremental testing strategy as with the scope analyzer.

And here is an outline for the solution:

- *Parser and AST generator.* This code is the same as in milestones 1 and 2.
- *Symbols, scopes, and symbol table.* You will need symbol classes for fields, functions, and variables. Each symbol has a definition, which is the AST node where the symbol is defined. You will need a scope class, which maps from strings to symbols. Each scope has an owner, which is the AST node that contains the scope. And finally, you will need a symbol-table class, which contains the scopes.
- *Scope analyzer.* The scope analyzer is a visitor that walks the AST and populates the symbol table. You need to define the intrinsic functions in the top-level scope. The analyzer should perform actions at each AST node that owns a scope or defines a symbol or both. It should report an error if multiple definitions in the same scope have the same name. Note that blocks only have a scope in certain contexts, not when they are an immediate child of a function or a for-statement. Further, note that formal parameters show up as field types, but also need to be declared as variables in the function scope.
- *Printing the symbol table.* You can add these print methods to the scope classes as needed.
- *Semantic analyzer.* The semantic analyzer is another visitor that walks the AST after the scope analyzer is done. It needs to re-push and pop the scopes discovered by the scope analyzer. The semantic analyzer finds types of literals and expressions bottom-up. You can represent these types using the AST classes for types; if you do that, just add a class for the null-type. Besides finding types, you need to check the type rules for literals, expressions, and statements. Print an error message if the type rules are violated. For these checks, you need methods that test the type relations (same type, sub-type, or castable). Finally, the semantic analyzer should rewrite the AST to insert casts for each implicit type conversion.

Example code

```
class TypeAnalyzer extends Visitor {
  SymbolTable _symTab;
  static final PrimitiveType STRING_TYPE = new PrimitiveType(null, PrimitiveType.STRING);

  static boolean sameType(Type t1, Type t2) { ... }
  static boolean subType(Type t1, Type t2) { ... }
  static boolean castable(Type t1, Type t2) { ... }

  ...

  Object visit(BlockStmt ast) {
    if (null != ast._heldScope)
      _symTab.push(ast._heldScope);
    for (Stmt s : ast._stmts)
      s.accept(this);
    if (null != ast._heldScope)
      _symTab.pop(ast._heldScope);
    return null;
  }

  Object visit(InfixExpr ast) {
    Type lhsType = (Type)ast._lhs.accept(this);
    Type rhsType = (Type)ast._rhs.accept(this);
    if ("||".equals(ast._op) || "&&".equals(ast._op)) {
      ...
    } else if ("+".equals(ast._op)) {
      if (sameType(lhsType, STRING_TYPE) || sameType(rhsType, STRING_TYPE)) {
        ast._type = STRING_TYPE;
        if (!sameType(lhsType, STRING_TYPE)) {
          if (castable(lhsType, STRING_TYPE))
            ast._lhs = new CastExpr(ast._lhs, STRING_TYPE);
          else
            ErrorPrinter.print(ast._lhs._loc, "Cannot convert from type '" + lhsType + "' to type 'string'");
        }
        ...
      } else {
        ast._type = INT_TYPE;
        ...
      }
    }
    return ast._type;
  }

  Object visit(VarId ast) {
    Symbol s = _symTab.lookup(ast._id);
    if (null == s) {
      ErrorPrinter.print(ast._loc, "Unknown variable '" + ast._id + "'");
    } else if (!(s instanceof VarSym)) {
      ErrorPrinter.print(ast._loc, "Variable name expected");
    } else {
      ast._sym = (VarSym)s;
      ast._type = ast._sym.type();
    }
    return ast._type;
  }
}
```