CS 321 Homework 4 – due 1:30pm, Thursday, March 15, 2012

Typechecking

In this assignment, you will build a type-checker for the **fab** AST structures you built in assignment 3. To do this, you'll process declarations to extract type information about identifiers and process statements and expressions to make sure that identifiers and literals are used correctly.

Your typechecker should detect all semantic errors, including:

- incompatible or illegal operand types in expressions, statements, or declarations;
- uses of undefined symbols;
- multiple declarations of a symbol within a single scope;
- attempting to assign to an identifier declared as being constant;
- incorrect number or types of arguments in a function call;
- `exit` statements in inappropriate contexts;
- `return` statements in inappropriate contexts, or lacking a return value when one is required or vice-versa;
- ill-formed record type definitions

Do *not* attempt to check that array references are within bounds; in general, this cannot be done until runtime.

As an (important!) side-effect, your typechecker must fill in the `type` field of all `Exp` and `LValue` nodes, and of any `VarDec` and `ConstDec` nodes where the concrete program text did not already specify a type. These type annotations will prove very useful in later stages of the compiler. They are now displayed (when present) by the AST `toString` functions; a dash ("`-`") is printed if the type field is null.

For a modest amount of *extra credit*, you may choose to include a check that functions whose return type is not `unit` actually execute a `return` on all possible paths through the function body. This check will necessarily need to be conservative, i.e., you will have to reject some programs that actually always do execute a `return`. The more precise your check is, the more credit you get. But from the **fab** user's perspective, it is very important that checks of this kind be *predictable*, so you must include an English-language *specification* of how your `return`-checker behaves.

Consult the **fab** Language Reference Manual for detailed rules about type compatibility. A full list of the error messages that you should be able to generate is given below.

Your typechecker must be implemented within file `Ast.java`, as a new method in class `Ast.Program`, with signature

```
void check() throws CheckError
```

where `CheckError` is a new class within `Ast`, defined as follows:

```
public static class CheckError extends Exception {
  CheckError(int line, String text) {
    super("Error at line " + line + ": " + text);
  }
}
```

If `check` encounters any type type error in the program, it should throw a `CheckError` exception with the line number at which the error occurs and a suitable explanatory message. Otherwise it should simply return silently.

Here's the driver that will be used to test your checker:

```
class CheckDriver {
  public static void main(String argv[]) throws Exception {
    try {
      Parser parser_obj = new Parser(new Scanner(System.in));
      Ast.Program prog = (Ast.Program) parser_obj.parse().value;
      prog.check();
      System.out.println(prog);
    } catch (ParseError exn) {
      System.err.println(exn.getMessage());
    } catch (Ast.CheckError exn) {
      System.err.println(exn.getMessage());
    }
  }
}
```

As usual, the "correct" behavior of the the typechecker is more precisely specified by the behavior of the reference checker embedded in the `WorkingChecker.jar` file available on the web page. (To run this checker code, you should include `WorkingChecker.jar` in your classpath.) The type-annotated AST produced by your checker, and printed by the driver above, should be the same as the reference checker. Your checker should issue errors whenever (and only when!) the reference checker does so. The text of your error messages and the associated line numbers do not have to match the reference checker exactly, but they should convey essentially the same information.

The reference parser includes a simple version of the extra-credit check for `return` statements, which considers all control paths through the program assuming that nothing is known about the value of any boolean expression.

Errors Generated

Here is a list of all the error messages the reference parser generates. Most are self-evident; explanatory comments are given for a few.

- `Identifier` *name* `is not defined`

  This applies to type names in declarations and record and array constructors; l-values; and `'for'`-loop indices.

- `Identifier` *name* `is already defined (at line` *linenum*`)`

  This applies to record type definitions, variables, parameters, and function names. No identifier can be defined more than once at a given scope level.

- `Identifier` *name* `is a type name and cannot be redefined`

- `Identifier` *name* `is not a type name`

  This applies to type names in declarations and array constructors.

- `Identifier` *name* `is a built-in name and cannot be redefined`

- `Variable initialized to 'nil' must have explicit type constraint`

- `Constant initialized to 'nil' must have explicit type constraint`

- `Type of initializing expression (`$type_1$`) does not match declared type (`$type_2$`)`

- `Duplicate field name` *name* `in record type declaration`

- `Cycle in record inheritance hierarchy`

- `Assignment LHS type (`$type_1$`) does not match RHS type (`$type_2$`)`

- `Function called in statement context must not return a value`

  Applies to function call statements.

- `Function called in expression context must return a value`

  Applies to function call expressions.

- `Function with return type` *type* `might not execute 'return' statement`

  Apply to function call expressions. (Generate these for extra credit.)

- `Call to expression of non-function type`

- `Wrong number of arguments provided`

- `Argument type (`*type*$_1$`) does not match declared type (`*type*$_2$`) for parameter `*name*
- `’read’ statement argument has type `*type*`; must be ’integer’ or ’real’`
- `’write’ statement argument has type `*type*`; must be ’integer’, ’real’, ’boolean’, or string`
- `Expression after ’if’ or ’elsif’ has type `*type*`; must be ’boolean’`
- `Expression after ’while’ has type `*type*`; must be ’boolean’`
- `Index of ’for’ statement has type `*type*`; must be ’integer’`
- `Expression in ’for’ statement has type `*type*`; must be ’integer’`
- `’exit’ statement is not inside a ’while’, ’loop’, or ’for’ statement`
- `’return’ statement not allowed in Main program body`
- `’return’ expression type (`*type*`) does not match declared procedure return type (`*type*`)`
- `’return’ missing result expression of type `*type*
- `’return’ from function with result type ’unit’ does not allow result expression`
- `Operand has type `*type*`; must be ’integer’ or ’real’`

  Applies to various arithmetic operators.
- `Operand types (`*type*$_1$`,`*type*$_2$`) do not match`
- `Functions cannot be compared for equality`

  Apply to = and <> operators.
- `Operand has type `*type*`; must be ’integer’`

  Applies to various arithmetic operators.
- `Operand has type `*type*`; must be ’boolean’`

  Applies to various boolean operators.
- `Array initializer count has type `*type*`; must be ’integer’`
- `Type of array initializer value (`*type*$_1$`) does not match declared array element type (`*type*$_2$`)`

- `Identifier` *name* `is not a record type name`

  Applies to record constructors and `extends` clauses in record type definitions.
- `Record initializer expression has missing field(s)`
- `Repeated field` *name* `in record initializer`
- `Type of expression (`$type_1$`) does not match type of field` *name* `(`$type_2$`)`
- `Undefined field` *name* `in record initializer`
- `Identifier` *name* `is not accessible in nested function`

  Applies to non-`const` parameters and locals declared in an outer enclosing function.
- `Identifier` *name* `cannot be assigned to`
- `Identifier` *name* `is not a variable`

  Apply to l-values being assigned into.
- `Identifier` *name* `is not a value`

  Applies to l-values being dereferenced for their contents.
- `Array subscript expression has type` *type*`; must be 'integer'`
- `Subscripted expression is not an array`
- `Field` *name* `does not appear in this record type`
- `Dereferenced expression does not have record type`

Implementation and Program Submission

The file `Ast0.java` on the course web page section for this assignment defines Ast classes very similar to the `Ast.java` file provided for homework 3, but also contains the skeleton of a typechecker that supports partial checking of **fab**. You are *strongly* urged (though not required) to use this code as the basis of your own checker.

The skeleton checker *omits* support for the following language features: all aspects of `boolean` types; all aspects of arrays; most aspects of function declarations; `read`, `if`, `while`, and `exit` statements; all operators except for `PLUS`, `MINUS`, `TIMES`, and `SLASH`; and call expressions. To implement the missing features requires about 100 lines of new code. Most (but not all) places where you need to add code are marked with a comment `NOT IMPLEMENTED`. You may want to change the signatures of some of the existing routines too.

In its present form, the skeleton checker will process the whole **fab** language without throwing any uncaught exceptions, but not correctly! In most cases, it will err by not issuing an error message where one is needed. In some cases it will issue inappropriate error messages (e.g., when the program mentions `true` or `false`, or when the skeleton checker incorrectly returns the type of an expression as `integer_t`) or assumes that every function type has an empty argument list.

Here are few notes on the design of this checker;

- The checker maintains an environment of all the identifiers currently in scope, implemented as a linked list of the `Binding` objects; `TypeBinding`s describe record types and the built-in types, and `ValBinding`s describe variables, parameters, functions, and built-in constants.
- Each identifier in the environment has an associated `level` number, which indicates how deeply its declaration is nested. Built-in identifiers are at level 0; record types are at level 1; top-level identifiers are at level 2; identifiers defined within a top-level function are at level 3; identifiers defined within a level 3 function are at level 4; etc.
- Since **fab** primarily uses structural type equivalence, we need a structural representation of types. The checker uses the existing `TypeExp` AST class for this purpose; objects of this class must be synthesized by the checker at the few places where they don't appear in the source program already. Named types can be compared by doing string compares on the names.
- In addition to the user-visible built-in types (e.g., `'integer'`), it is convenient to define a few "internal" built-in types (e.g., for strings). To avoid possible clashes with user-defined type names, these internal names begin with a character (`?`) that is illegal in concrete syntax identifiers.
- The `TypeExp` member functions `equiv_to` and `subtype_of` answer whether the receiving object is structurally equivalent to, or a subtype of, the argument object. In most cases, `subtype_of` is the correct test to apply.

Your revised `Ast.java` file should be submitted as a plain text attachment to a mail message sent to `cs321-03@cecs.pdx.edu` with subject line prfix "`[hw4]`". Be sure to include your name in a comment within your submitted code file. Your code must work correctly with the provided (assignment 4) versions of `CheckDriver.java`, `ParseError.java`, `Symbol.java`, `Scanner.java`, `Yylex.class`, `Parser.class`, `CUP$Parser$actions.class`, and `SymKinds.class`.