COMS W4115
Programming Languages and Translators
Programming Assignment 2: Static Semantics

Prof. Stephen A. Edwards
Columbia University

Assigned February 20th, 2002
Due 11:59 PM on March 15th, 2002

For this assignment, you will be adding static semantic checking to your Tiger compiler. The lexer/parser from the last assignment verified the program was syntactically correct, but accepted many nonsensical programs such as 1 + "foo". A program that makes it past this assignment should be correct, i.e., when implemented should not have any type errors.

For a Tiger program to be sematically correct,

- Any identifier that is used must have been defined within its scope.
- An identifier used as an lvalue must be a variable.
- The type of each identifier and sub-expression must be established.
- For each operator (e.g., `-`, `*`), the types of its operands must match and conform to the rules for the operator.
- For each assignment, the type of the expression being assigned and the lvalue must match.
- For each function call, including those to the standard library, the identifier must refer to a function and the number and type of arguments must match the declaration.
- For each record constructor, the given type must be a record and the number, names, and types of each fields must match.
- For each array constructor, the given type must be an array and the expression for its initial value must match.
- The predicate expression in both the `if` and `while` statements must return an integer.
- The *then* and *else* expressions in an if-then-else must be of the same type.
- The expression in an *if-then* expression must not return a value.
- The expression in a *while-do* expression must not return a value.
- The first and second expressions in a *for* expression must be integer, but the third must not return a result.
- No explicit assignment may be made to a *for* loop index variable.
- A *break* expression must appear within a *for* or *while*.
- An lvalue on the left side of a `.` field selector operator must be a record. The identifier on the right must refer to a field in that record.
- If a variable declaration specifies a type, it must match the type of the expression.
- If a function declaration specifies a type, its body must return that type.
- If a function declaration does not specify a type, its body must not return a type.
- It must not have circular type definitions, e.g.,

```
type a = b
type b = c
type c = a
```

There are probably other rules. Think about what it means for each AST node to be correct.

1 Getting Started

Files for this assignment are in `~cs4115/prog2` on the CUNIX cluster. Copy this subdirectory and all its files into your account to get started. Included are

- A skeleton semantic analysis phase, `TigerSemant.g`. This assignment amounts to adding actions to the rules in this file.
- A front-end that calls the lexer, parser, and static semantic checker. (`TC.java`)
- A package with classes for symbol tables, a checking environment, and Tiger types. (`Semant`).
- A working lexer and parser from the first assignment as `.class` files.

To compile the example checker under bash,

```
$ CLASSPATH=~cs4115/antlr:.
$ export CLASSPATH
$ java antlr.Tool TigerSemant.g
$ javac TC.java
$ java TC
1 + 2
^D
$ java TC
1 + break
^D
1:operands of + must be integer
$
```

For the purposes of this assignment, feel free to use, modify, or ignore any of the files I give you. However, we must be able to compile your project using the procedure described below, the "main" program must be the TC class, and it must return a non-zero value when an error has occurred.

2 Philosophy of Static Semantics

Your job is to complete the AST walker I started for you in `TigerSemant.g`. The basic idea of static semantics is, for each AST node, to verify its children are correct (e.g., establish their types), and then verify those types are consistent with the node.

Throughout the recursive walk, an environment of stacked symbol tables is maintained that keeps track of what identifiers are defined and their types. The `let` statement and its declarations add and remove things from this environment, and rules such as lvalue or a function call check this environment.

ANTLR is the most convenient way to write a node-type-specific recursive traversal of an AST. See the supplied `TigerSemant.g` for an example. This is `TigerASTGram.g` from the last assignment with added rules.

A question that often arises is "do I check this here?" In general, the answer is yes if it depends on your children but not on your parents (e.g., the children of a + BINOP must be integers), but no if it depends on your parents (e.g., an ID does not need to verify it is an integer if it happens to be an operand of a +).

3 Deliverables

As before, use `~cs4115/bin/submit_code` to submit

- Your customized `TigerSemant.g` file.
- A README file describing your semantic analyzer. I want to hear how you dealt with
  - Recursive function definitions.
  - Recursive type definitions.
  - Checking whether `break` was properly nested.
  - Testing your semantic analyzer
- A subdirectory called "tests" containing test problems.
- A file called `MEMBERS` that contains a space-separated list of the uni IDs of each of the members in your group.

Make sure we can build your semantic analyzer by running ANTLR on `TigerSemant.g` and then `javac` on `TC.java`. We do not want a Makefile.

```
lvalue returns [Type t]
    { Type a, b; t = env.getVoidType(); }
  : i:ID
    { /* Verify ID is a variable, return its type */
      Entry e = (Entry) env.vars.get(i.getText());
      if ( e == null )
        semantError(i, "Undefined identifier " + i.getText());
      if ( !(e instanceof VarEntry) )
        semantError(i, i.getText() + " is not a variable");
      VarEntry v = (VarEntry) e;
      t = v.ty;
    }
  | #( FIELD a=lvalue ID )
    { /* Verify lvalue is of record type with ID as a field */ }
  | #( SUBSCRIPT a=lvalue b=expr)
    { /* Verify lvalue is an array type and expr is an int */ }
  ;

expr returns [Type t]
    { Type a, b, c; t = env.getVoidType();}
  : "nil" { t = env.getNilType(); }
  | t=lvalue
  | STRING { t = env.getStringType(); }
  | NUMBER { t = env.getIntType(); }
  | #( NEG a=expr
       { /* Verify expr is an int */
           if ( !(a instanceof Semant.INT))
                semantError(#expr, "Negate operand not int");
           t = env.getIntType();
       }
     )
  | #( BINOP a=expr b=expr
       { /* Verify expr's types match, more picky for non-equality. */
         String op = #expr.getText();
         if ( op.equals("+") ||
              op.equals("-") ||
              op.equals("*") ||
              op.equals("/") ) {
             if (!(a instanceof Semant.INT) ||
                 !(b instanceof Semant.INT))
                 semantError(#expr, op+" operands not int");
             t = a;
         } else {
             semantError(#expr, "other operators unimplemented");
             t = env.getVoidType();
         }
       }
     )
  | /* ... */
  | #( "let"
       { env.enterScope(); }
       #(DECLS (#(DECLS (decl)+ ))* )
       a=expr
       {
         env.leaveScope();
         t = a;
       }
     )
  ;

decl
  : #( "var" i:ID (a=type | "nil" { a = null; } ) b=expr
       { env.vars.put(i.getText(), new VarEntry(b)); }
     )
  ;
```

Figure 1: A fragment of `TigerSemant.g` showing partial rules.