Compiler project tasks — part 5, due Monday 12 November

The goal in this phase is to verify the type-correctness of the program.

Write a type-checker: compiler phase that will traverse the abstract syntax tree and annotate it with types, reporting any errors found along the way, and printing its annotations to a diagnostic file.

Given input program `primes.m`, the output file should be named `primes.typecheck`. Test on all programs we have used so far.

Implementation Hints

Types

Within the compiler, you will need to represent the types of the source language. The types are either base types (integers, reals, booleans) or constructed types (subranges, records, arrays, references). (However, the INTEGER type is best represented as a subrange MININT..MAXINT.)

This representation should include type equality testing and type assignability testing.

Values

You will need to represent the constant values of any type. This involves representing values of base types as values of appropriate types in your implementation language, and building data structures for values of constructed types.

This representation should include arithmetic on constant values, which will be needed later for constant folding.

Environments

You will need a compile-time representation of referencing environments. The kinds of objects that can be bound to names are: variables, constants, procedures, parameters, types, and modules.

Here is a snippet of ML code with a possible representation for an environment:

```
   datatype kind = VARKIND | CONSTKIND | PROCKIND | PARAMKIND | TYPEKIND | MOD-
ULEKIND

   datatype binding = VAR of {id: id, object: object}
                    | CONST of {id: id, object: object}
                    | PROC of {id: id, object: object, params: (id * ob-
ject * bool) list}
                    | PARAM of {id: id, object: object, lval: bool}
                    | TYPE of {id: id, ty: ty}
                    | MODULE of {id: id}
```

```
type env = binding list

val empty = nil

fun bindvar (env, id, object) = VAR {id=id, object=object} :: env

fun lookupvar (nil, id) = impossible ("VAR "^id)
  | lookupvar (VAR {id=id', object=object} :: t, id) =
      if id = id' then object
        else lookupvar (t, id)
  | lookupvar (_::t, id) = lookupvar (t, id)
```

Here I grouped all kinds of objects into a single construct, which records the type (if relevant), value (if relevant), and the static nesting level of the binding:

```
datatype object = ENTRY of
                      {ty: ty,
                       va: va,
                       display: int
                      }
```

Initial environment

An initial outermost environment should be built containing the bindings for the predefined names of base types and predefined names of constants: INTEGER, REAL, BOOLEAN, TRUE, FALSE, MININT, MAXINT.

Type checking

Type checking proceeds in the manner of attribute grammars. You do not need to formalize the process as an L-attributed attribute grammar, though you could. The two attributes needed are types and environments, and they are attached appropriately to most constructs in the abstract syntax.

For instance:

```
      datatype Program =
         PROGRAM of Module * env
...
      and      Declaration =
         DECLCONST    of id * object
...
      and      Stmt =
         STMTBLOCK    of Block  * env
       | STMTASSIGN   of Expr * Expr  * env
...
```

```
      and      Expr =
         EXP       of BaseExpr * ty * (*is lvalue?:*) bool  * env

      and      BaseExpr =
         EXPROR       of Expr * Expr
       | EXPRAND      of Expr * Expr
...
```

Type checking synthesizes types for complex expressions from types of constituent expressions, bottom-up. Meanwhile, environments carry type information for individual names up from declarations and down into statements.

Note that in my implementation one set of ML datatypes is used for the original abstract syntax tree, and another for the annotated abstract syntax tree. I traverse the first tree and build the second tree from scratch. Other implementation languages may favor in-place update of trees.