

CS 321 Homework 4 – due 1:30pm, Wednesday, December 1, 2004

This homework specification is copyright 2002-2004 by Andrew Tolmach. All rights reserved.

Typechecking

In this assignment, you will build a type-checker for the PCAT 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 a constant;
- incorrect number or types of arguments in procedure call;
- EXIT statements in inappropriate contexts;
- RETURN statements in inappropriate contexts, or lacking a return value when one is required or vice-versa;

Do *not* attempt to check that function procedures always execute a RETURN, or that array references are within bounds. Consult the PCAT 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 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.

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 Yylex(System.in));
            Ast.Program prog = (Ast.Program) parser_obj.parse().value;
            prog.check();
        } 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 `Ast.jar` file available on the web page. (To run this checker code, you should include `Ast.jar` in your classpath.) Your checker should issue 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.

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 variable declarations, type declarations, procedure declarations, and record or array constructors; l-values; procedure names; and FOR loop indices.
- Identifier *name* is already defined (at line *linenum*)
This applies to variables, parameters, procedure names, and type 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 variable declarations, type declarations, procedure declarations, and record or array constructors.
- Variable initialized to NIL must have explicit type constraint
- Type of initializing expression (*type₁*) does not match declared type (*type₂*)

- Duplicate field name *name* in record declaration
- Assignment LHS type (*type₁*) does not match RHS type (*type₂*)
- Identifier *name* is not a procedure
Applies to all procedure calls.
- Procedure *name* is a function procedure
Applies to procedure call statements.
- Procedure *procName* is not a function procedure
Applies to procedure call expressions.
- Wrong number of arguments provided
Applies to all procedure calls.
- Argument type (*type₁*) does not match declared type (*type₂*) for parameter *name*
Applies to all procedure calls.
- 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 *varname* of FOR statement is not a variable
- Index *varname* 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 from function procedure must have result expression
- RETURN from proper procedure 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
Applies 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.
- Identifier $name$ is not an array type name
Applies to array initializers.
- 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 initializers.
- 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 a variable
Applies to l-values in contexts where they will be updated.
- Identifier $name$ is not a variable or constant
Applies to all l-values.
- Array subscript expression has type $type$; must be 'INTEGER'
- Subscripted expression is not an array
- Field $name$ does not appear in this record
- Dereferenced expression is not a record

AST Changes

The `Ast.java` file provided for this assignment is different from the one provided in assignment 3 in several ways:

- Skeleton code for the typechecking functions has been added (see next section).
- Some purely internal reorganization of the AST class structure has been done. For example, `FormalParam` has been made a subclass of `Dec`, which now includes a name field. Also, there are new classes `BuiltinType` and `ConstDec` intended for use by the checker. These changes do not affect the already-existing constructors, parser, or pretty printer.
- One externally visible change has been made, which does require a change to the parser. You can download a revised working parser that matches this version of `Ast.java` from the web page. If you want to use your parser from assignment 3, you'll need to make the following change to the treatment of `VarDecs`. In the new AST, each `VarDec` declares only one variable, instead of a list of variables. To translate the concrete syntax for a multi-variable declaration, e.g.:

```
VAR a, b, c : INTEGER := 2 + 2;
```

you must now generate a list of `VarDec` nodes, each with the same type, and using the name of the *first* variable as the initializing expression for the others, e.g. to the equivalent of

```
VAR a : INTEGER := 2 + 2;  
    b : INTEGER := a;  
    c : INTEGER := a;
```

(Note that we can't just replicate the initializing expression, because the PCAT language spec dictates that it be evaluated just once – it might have side-effects!) The purpose of this change is to guarantee that each `Dec` object binds exactly one identifier.

Implementation and Program Submission

As noted above, the version of `Ast.java` on the course web page section for this assignment contains the skeleton of a typechecker that supports partial checking of PCAT. 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 procedure 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 fewer than 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 PCAT language, but not correctly! In most cases, it will err by not issuing an error message where one is needed. In a few 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 `?`).

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 `Dec` objects for these identifiers.
- 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; top-level identifiers are at level 1; identifiers defined within a top-level procedure are at level 2; identifiers defined within a level 2 procedure are at level 3; etc.
- Since PCAT uses pure name equivalence, we can compare types simply by doing string compares on their names. Moreover, since type names cannot be redeclared, we can determine the definition of a type just by looking up its declaration in the current environment.
- 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 function `promoteType` is used to handle the small number of situations where an exact type match is not the right thing to do. It should always be possible to use an `INTEGER` where a `REAL` is expected, or to use the value `NIL` where any sort of `RECORD` type is expected.

Your revised `Ast.java` file should be submitted as a plain text attachment to a mail message sent to `cs321-01@cs.pdx.edu`. The subject line of the mail should contain your name and the string “HW4”. Your code must work correctly with the provided (assignment 4) versions of `CheckDriver.java`, `ParseError.java`, `Symbol.java`, `Ylex.class`, `Parser.class`, `CUP$Parser$actions.class`, and `SymKinds.class`.