PCAT Semantic Errors

Harry Porter
Dept. of Computer Science
Portland State University

Alphabetical List

```
All expressions in FOR statement must have type INTEGER
An operand of this operator has the wrong type
Array constructor ID must name an ARRAY type
Array count must be an INTEGER expression
Array subscript must have type INTEGER
Array subscripting attempted on non-array value
Conditional expr after IF, ELSEIF, or WHILE is not BOOLEAN
Dead code - execution can never reach the bottom of this loop
Dead code - execution can never reach this statement
Every field in RECORD must be assigned to exactly once
EXIT statement is not within a FOR, WHILE, or LOOP
Expecting a local or formal name
Expecting a procedure name
Expecting a type name
Field accessing attempted on non-record value
Identifier is already defined
Identifier is not defined
In assignment, type of LHS is not compatible with type of RHS
Incorrect number of args in procedure invocation
Index of FOR must be previously declared as INTEGER
INTEGER, REAL, BOOLEAN, TRUE, FALSE, and NIL may not be redefined
Last executable stmt in this PROCEDURE is not a RETURN
Multiple assignment to field in RECORD constructor
Non-void procedure expected
READ stmt requires INTEGER or REAL args only
Record constructor ID must name a RECORD type
RETURN from a non-void procedure expects a result value
RETURN from a void procedure does not allow a result value
RETURN not allowed in the main program body
This field is already defined in this RECORD
This field is not in this RECORD
Type needed when initializer is NIL
Type of argument does not match parameter type
Type of expr is not compatible with ARRAY definition
Type of expr is not compatible with this field's type
Type of initializing expr does not match the given type
Type of RETURN value not compatible with PROCEDURE definition
Void procedure expected
WRITE requires INTEGER, REAL, BOOLEAN, or STRING args
```

Identifier is already defined

An identifier may name a

- variable
- parameter to a procedure
- procedure
- type
- field within a record

All variable, parameter, procedure, and type names defined in a given scope must be different. Each procedure constitutes a new scope and the main body constitutes the outermost scope. Each identifier may only be defined once in each scope, but may be defined in one scope and then again in an inner scope.

Field names are handled a bit differently from variable, parameter, procedure, and type names. Each record type introduces a new namespace for its fields. The same ID may be used to name fields in several different records (even nested records), but every field in any one record type must have a different name. Also, an ID used as a field name may also name a variable, parameter, procedure, or type; record namespaces are disjoint from scopes.

Identifier is not defined

Whenever an identifier is encountered in the following places, there must be a corresponding definition within this scope or within an enclosing scope.

- As a type name, for example in the position of **foo** in any of the following:
```
var x: foo := ...;
type t is foo;
type t is array of foo;
type t is record ... f: foo; ... end;
procedure ... (...) : foo is ...;
procedure ... (..., x:foo, ...) ...;
```
- As an l-value, for example in the position of **foo** in:
```
read (..., foo, ...);
foo := 23;
a := foo + 1;
```
- In a procedure invocation (either as a statement or within an expression):
```
foo (1,2,3);
a := foo (4.5);
```
- As a FOR index, for example:
```
for foo := 1 to 10 DO ... ;
```
- In a record constructor, for example in the position of **foo** in:
```
a := foo { f1:=100; f2:=200 };
```
- In an array constructor, for example in the position of **foo** in:
```
a := foo [< 1, 2, 100 of 3, 4, 5 >];
```
- As a field name. In the following two examples, there must be a field named **foo** in the record type of r:
```
a := r.foo;
a := r { ... foo:=100; ... };
```

Expecting a local or formal name
Expecting a type name
Expecting a procedure name

The name in question is defined but that name is not the right kind of thing. For example:

```
type foo IS ...;
x := 4 * foo;
```

Another example of the error is:

```
var bar: integer := 0;
x := bar (1,2,3);
```

This field is already defined in this RECORD

In a record type declaration, each field must have a different name. Fields in nested records may share names, as in the following legal example:

```
type t is record
            f1: integer;
            f2: record
                  f1: real;
                  f2: boolean;
                end;
          end;
```

Every field in RECORD must be assigned to exactly once

This error occurs when a RECORD constructor fails to assign to a field. In this example, an assignment to **f2** is missing:

```
type MyRec is record
                f1: integer;
                f2: real;
              end;
x := MyRec { f1:=100 };
```

Multiple assignment to field in RECORD constructor

The following is in error since **f1** is repeated.

```
x := MyRec { f1:=100; f2:=200; f1:=100 };
```

Record constructor ID must name a RECORD type

Here is an example of the error:

```
type foo is array of ...;
r := foo { ... };
```

Type of expr is not compatible with this field's type

In a record constructor, each field is given an initial value. The type of the initializing expression must be "assignment compatible" with the field's type. (See the error concerning type compatibility for assignment statements for the definition of "assignment compatible.") An INTEGER to REAL coercion will be inserted if necessary.

In the following example, the assignment to **f1** is okay (a coercion will be inserted), but the assignment to **f2** is in error.

```
type MyRec is record
                f1: real;
                f2: real;
              end;
x := MyRec { f1 := 13; f2 := true };
```

Array constructor ID must name an ARRAY type

Here is an example of the error:

```
type foo is record ... end;
r := foo [< 1, 2, 3 >];
```

Type of expr is not compatible with ARRAY definition

In an ARRAY constructor, the initial values are given by a series of expressions. The type of these expressions must be "assignment compatible" with the ARRAY type definition. Here is an example of the error:

```
type MyArr is array of boolean;
r := MyArr [< 1, 2, 3 >];
```

Array count must be an INTEGER expression

Here is an example of the error:

```
type MyArr is array of real;
r := MyArr [< 500.01 of 0.0 >];
```

Type needed when initializer is NIL

If the initializing expression for a VAR declaration is NIL, the type must be supplied. For all other expressions, the type can be deduced. However, "nil" is compatible with all RECORD and ARRAY types. The following is legal:

```
var x := 1.5;
var y: MyRec := nil;
```

In assignment, type of LHS is not compatible with type of RHS

The type of the expression on the right-hand side of the **:=** must be "assignment compatible" with the type of the l-value on the left-hand side. By this we mean: the type on the right-hand side must match the type on the left-hand side exactly, with two exceptions. First, "nil" matches all RECORD and ARRAY types. Second, the type on the right-hand side may be INTEGER when the type on the left-hand side is REAL. (In this second case, an INTEGER to REAL coercion will be inserted.)

Here is an example of the error:

```
var x: boolean := false;
x := 3.14159;
```

Type of initializing expr does not match the given type

In a VAR declaration, if a type is specified, the type of the initializing expression must be "assignment compatible" with the type supplied in the declaration.

Here is an example of the error:

```
var x: integer := 3.14159;
```

INTEGER, REAL, BOOLEAN, TRUE, FALSE, and NIL may not be redefined

These identifiers—they are not keywords—have predefined meanings. Any attempt to use them in variable declarations, type definitions, procedure definitions, as parameter names, or as field names is in error.

Array subscripting attempted on non-array value

Here is an example of the error:

```
var a: integer := 0;
a[4] := 1;
```

Array subscript must have type INTEGER

Here is an example of the error:

```
var i: real := 3.5;
a[i] := 1;
```

Field accessing attempted on non-record value

Here is an example of the error:

```
var r: integer := 0;
x := r.myField;
```

This field is not in this RECORD

Here is an example of the error:

```
var r: record
          f1: integer;
          f2: real;
       end;
x := r.f5;
```

Incorrect number of args in procedure invocation

If procedure **foo** is defined with 4 formal parameters, then any use of **foo** must provide exactly 4 arguments. Here is an example of the error:

```
procedure foo (i,j,k,l: integer) : integer is begin ... end;
x := foo (7,8,9);
```

Type of argument does not match parameter type

The type of the actual argument must be "assignment compatible" with the type of the formal parameter. A coercion from INTEGER to REAL will be inserted if the actual argument is of type INTEGER and the formal parameter is of type REAL.

Here is an example of the error:

```
procedure foo (a: integer) : integer is begin ... end;
x := foo (true);
```

Void procedure expected

Here is an example of the error. **Foo** should be used in expressions, not as a statement.

```
procedure foo (x: integer) : boolean is begin ... end;
...
foo (3);
```

Non-void procedure expected

Here is an example of the error. **Foo** should be called at the statement level, not used in expressions.

```
procedure foo (x: integer) is begin ... end;
...
a := foo (3);
```

Conditional expr after IF, ELSEIF, or WHILE is not BOOLEAN

The following code contains this error in both conditional expressions:

```
if 4+5 then
  ...
elsif 3.1415 then
  ...
else
  ...
end;
```

Here is another example of the error:

```
while (1) do ... end;
```

Index of FOR must be previously declared as INTEGER

Here is an example of the error:

```
var i: real := 0.0;
...
for i := 1 to 10 do ... end;
```

If the declaration of **i** had been completely forgotten, we would have gotten an "Identifier is not defined" error message.

All expressions in FOR statement must have type INTEGER

There will be two or three expressions in each FOR loop (the BY clause is optional). Here are examples of errors in all three places:

```
for i := 1.2 to (i<100) by MyRec{f1:=100;...} do ... end;
```

EXIT statement is not within a FOR, WHILE, or LOOP

Each EXIT is associated with the closest surrounding FOR, WHILE, or LOOP statement. Here is an example of the error:

```
procedure foo (...) is
  begin
    for ... do
      ...
    end;
    exit;
  end;
```

RETURN not allowed in the main program body

Execution automatically halts (i.e, returns to the operating system) after the last statement in a program body. Here is an example of the error:

```
program is
  begin
    ...
    write ("bye-bye");
    return;
  end;
```

RETURN from a non-void procedure expects a result value

Here is an example of the error:

```
procedure foo (...) : integer is
  begin
    ...
    return;
    ...
  end;
```

RETURN from a void procedure does not allow a result value

Here is an example of the error:

```
procedure foo (...) is
  begin
    ...
    return 56;
    ...
  end;
```

Type of RETURN value not compatible with PROCEDURE definition

The type of the expression appearing in a RETURN statement must be "assignment compatible" with the type specified in the header of the procedure. Here is an example of the error:

```
procedure foo (...) : boolean is
  begin
    ...
    return 56;
    ...
  end;
```

A coercion from INTEGER to REAL will be inserted in the RETURN statement in the following example:

```
procedure bar (...) : real is
  begin
    ...
    return 56;
    ...
  end;
```

Last executable stmt in this PROCEDURE is not a RETURN

Execution in a PROCEDURE must not "fall out the bottom." Regardless of whether a PROCEDURE returns a result or not, it should end with a RETURN. Here is an example of the error:

```
procedure foo (...) : integer is
  begin
    ...
    i := 10;
  end;
```

Note that the following is legal since execution can never fall out the bottom of a LOOP statement without any EXIT statements:

```
procedure bar (...) : integer is
  begin
    loop
      i := i*i;
      write (i);
    end;
  end;
```

Dead code - execution can never reach this statement

Here is an example of the error:

```
procedure foo (...) is
  begin
    ...
    return;
    write ("Hello");
    ...
  end;
```

Dead code - execution can never reach the bottom of this loop

Here is an example of the error. No matter which branch of the IF is taken, the body of the WHILE loop can be executed at most once. Execution can never take the branch back to the top of the loop; something is wrong with this program's logic.

```
procedure foo (...) is
  begin
    ...
    while ... do
      ...
      if ... then
        exit;
      else
        return;
      end;
    end;
    ...
  end;
```

An operand of this operator has the wrong type

The following operators require their operand(s) to be either INTEGER or REAL:

```
unary+    unary-    +    -    *    /    <    <=    >    >=
```

For the operators listed here, one operand may be INTEGER and the other may be REAL. When mixed, the compiler will quietly insert a conversion from INTEGER to REAL for the INTEGER operand. Also, the compiler will always insert a conversion for each INTEGER operand to the division operator (/), even when both operands are INTEGER.

The following operators require their operand(s) to be BOOLEAN:

```
NOT    AND    OR
```

The following operators require both their operands to be INTEGER:

```
DIV    MOD
```

The following operators can handle any types, but both operands must be the same type:

```
=    <>
```

If one operand is INTEGER and the other is REAL, a conversion will be inserted silently to coerce the INTEGER to a REAL. Also, “nil” can be compared to any RECORD or ARRAY type.

READ stmt requires INTEGER or REAL args only

A READ statement can be used to read in INTEGER and REAL values only. As such, the l-values listed must have types of INTEGER and REAL. In the following code, the variable **b** causes this error:

```
var i: integer := 0;
    f: real    := 0.0;
    b: boolean := false;
read (i, f, b);
```

`WRITE requires INTEGER, REAL, BOOLEAN, or STRING args`

This code is okay:

```
var i: integer := ...;
    f: real    := ...;
    b: boolean := ...;
    a: array of integer := ...;
write ("i=", i, "f=", f, "b=", b, "expr=", a[i+5]*7);
```

The WRITE statement cannot be used to print ARRAY or RECORD values (or “nil”). Here is an example of the error:

```
write ("a=", a);
```