Typechecking BPL

Your type checker needs to make two passes through your parse tree:

A. A top-down pass that links every symbol (variable or function) usage to its declaration.
B. A bottom-up pass that assigns a type to every expression (remember that expressions represent values, so they have types) and checks that the program is type-correct. One important part of this is ensuring that function calls have the correct number and types of arguments.

A program that successfully passes through your scanner, parser and type checker should be a correct, valid program that you can generate code for.   If you aren't signaling errors by throwing exceptions you should give some kind of signal that there is an error in the program so you don't try to generate code for it.

Debugging Info

We verified the correctness of your parser by printing out the parse tree returned by your parser. There is no single object returned by the type checker -- it just modifies the parse tree. You should include a variable DEBUG in your type checker that, when set to True, causes two kinds of information to be printed:

- When a symbol usage is linked to a declaration in the first phase you should print something like
  "Symbol X on line 23 is linked to declaration on line 5."
  The line numbers for both the symbols nodes and the declaration nodes should both be in your parse tree (if they aren't, add them).
- Each time you assign a type to an expression print something about it.

The second of these print statements can be handled in many different ways, some more verbose than others. For example, if line 12 of your program is the line y = x+1 you might print

```
Variable y on line 12 assigned type integer
Variable x on line 12 assigned type integer
Integer constant on line 12 assigned type integer
+ expression on line 12 assigned type integer
Assignment on line 12 assigned type integer
```

Some of these items might be omitted; the variables and constants might not need to have their types printed. But some information of some sort coming out of the type checking stages, combined with correct error reporting on bad programs, will help you have confidence that your type checker is finding all of the information it needs. It is easy to read around excess print statements.

Be sure that you have a way to turn off the printing of this debugging information. You don't want all of these print statements coming out when you are in the code generation phase.

Types in BPL

BPL by design has a very simple type system. The only possible types are int, string, void (as a return-type for functions), pointer to int, pointer to string, array of int, array of string.

For more general type systems you would need to do a lot more, but I wanted to focus on other things in the compiler project (like getting it done). With these simple options you can represent types with strings. I have a "type" field in every exp node, that my type checker fills in. This field is just a string: "int", "string", "void", "int array", "string array", "pointer to int" and "pointer to string". In general, something is type correct if its type field is what I expect it to be.

Symbol Tables

As I said earlier, I do two passes: a top-down one for resolving symbols and a bottom-up one for checking types. You could combine these into one -- resolve references in a function declaration, then on the way back up check types. I find it easier to keep the two passes separate.

To resolve references you need to maintain a table of all current symbols. There are three places in the code where you could change the active environment:

- top-level declarations (these are usually functions but could be arrays or variables)
- parameters in function declarations. These are visible in the body of their function, nowhere else.
- local declarations at the start of a compound statement. These are visible only in the body of their compound statement.

For the symbol table you need to be able to look up a the name of an object and find its declaration, so implicitly the table is indexed by strings and the table entries are declaration nodes. There are lots of ways to arrange this. I do different things in the two basic levels: top-level declarations I put in a hash map of type (I do this in Java) HashMap<String, DecNode>. Declarations that are local to a function or compound statement I do with a linked list of DecNodes. It is easy to add onto the front of a linked list in such a way that I preserve a pointer to where I started. When I reach a compound statement and variable localDecs is a pointer to my current list of local declarations, I add the new declarations onto localDecs, find references in the body of the compound statement with the extended list of declarations, then find references in the next statement, if there is one, with the original localDecs. When I am trying to resolve a symbol I first look in the local declarations, then if those don't resolve the symbol look in the hashmap of global declarations.

There are lots of alternatives to what I do. One would be to use a hashmap for everything. This requires you to have a naming convention for multiple uses of the same name (you might have a global function f and a local variable f). Alternatively, you could use a linked list for everything. Commercial compiler-writers worry a lot about lookup times in a symbol table of a large program, but you don't need to consider this; anyone who writes large programs in BPL deserves what they get. Do be sure that you handle the scoping correctly. The compound statement on the left below is incorrect and should give an error; the one on the right is correct and should print 2.

```
{
   { int x;
      x = 1;
   }
   write (x);
}
```

```
{ int x;
   x = 1;
   { string x;
       x = "bob";
    }
     write(x+1);
}
```

The top-down pass: FindReferences(tree)

The creates the top-level symbol table. Since a program in BPL consists of a sequence of declarations, this walks through those declarations, extending the symbol table for each. What it does for a declaration depends on the type of declaration:

a) For variable declarations and array declarations, it just adds the declared variable to the symbol table.
b) For function declarations there is much more to do. I create a list of local declarations, which starts with the parameters of the function. I find it helps to make a new list structure for the local declations; each item has just a declaration and a next field. If you are careful you could probably avoid this. I then call a function that finds all of the references for a statement, giving it the global symbol table, the list of local declarations, and the function body, which is a statement.

FindReferencesStatement(s, symbolTable, localDecs) looks at the kind of statement node S is. If it is a while-statement we call FindReferencesExpression on the condition, and FindReferencesStatement on the body. If it is an if-statement we call FindReferencesExpression on the condition and then findReferencesStatement on each of the branches. And so forth. None of the statement types are difficult.

FindReferencesExpression has a little more to do. For one thing, if we get to a variable or array reference, we need to actually resolve it. We first look up the name in the local declarations, and if we don't find it there in the global symbol table. If it isn't found we give an undeclared variable error message (I throw an exception and halt; you can do what you want). If we do find it we set up a link from the symbol to its declaration. All of my variable and array nodes have a field called declaration, which is just a declaration node. We set this to the value bound to the name in the symbol table.

The other expression node that makes us work is a function call. The function itself should be found in the symbol table. Once we set up the link from the function to its declaration, we also need to call FindReferencesExpression on each of the arguments to the call. For example, we might have call

```
z = f(x+1, x+y).
```

Both of the arguments have symbol uses that need to be resolved.

The bottom-up pass: TypeCheck

For this pass we do a simple leaf-first traversal of the tree (TypeCheck the children, use their types to TypeCheck the root). At the top level we walk along the list of declarations that make up the program. For variable and array declarations there is nothing to type check. For function declarations we call TypeCheckStatement on the body of the function.

For most of the statement types, TypeCheckStatement is easy. For example, to typecheck a While-statement we call TypeCheckExpression on the condition. This should result in a type being assigned to the condion node. Back in TypeCheckStatement, we check that this type is "int" (remember that BPL doesn't have a boolean type). If the type is not "int" we throw an error; it is is we call TypeCheckStatement on the two branches.

Type checking a return statement is more difficult. Our functions could return an int or a string. If the return type of a function is "int" we should check that it does have a return statement and that this statement returns a value of type int.

This is the one place that your type checker needs non-local information -- whether the return statement is type correct depends on the function this statement belongs to. I give TypeCheckStatement a parameter that has the return type of the current function whose body is being type checked. That means it is easy to check that the type of the value being returned is correct.

However, I cheat a bit by not going the extra step of ensuring that the function contains a return statement. We really sure check that in all branches of the function there is a return statement, and this requires more run-time analysis than we have time to do. So I would find the following program type-correct:

```
        int f(int x) {
                write(x);
        }
        void main(void) {
                f(5);
        }
```

That doesn't seem like a big problem. However, if we changed the body of main to

```
                write( f(5));
```

I would call it type-correct and it runs into trouble because f never returns anything.   So let the programmer beware.

TypeCheckExpression should assign a type to the expression node. For most of the expressions this is easy. Int nodes have type "int". Read nodes also have type "int". String values have type "string". A binary operator has two children that are expressions, so we recursively check their types. All of the operators require these two children to have the same type; arithmetic operators require their types to both be "int". Throw an error if any of these conditions aren't met. If they are met assign the obvious type to the parent node.

Unary operators & and * have an extra step but aren't difficult to handle. Array element references such as A[i+1] are also fairly simple. A must resolve to an array of some base type ("int" or "string"), the index expression must be an "int". If these are satisfied then A[i+1] has the base type of the array.

Function calls are a bit more work. If we have a call such as f(x+1, y) we need to type check each argument, then walk along the parameter list of the declaration of f. The argument list must have the same length as the parameter list and each argument must have the same type as the corresponding parameter. If all of this is satisfied, the type of the call node is the return type of the function.

The other expression that requires care is the assignment expression. It isn't enough for both sides of the assignment to have the same type -- you don't want to accept an expressin 1=0. For an assignment statement to be type-correct, the left side must be an "L-value", i.e., something we can assign to. For BPL the only legitimate L-values are

- variables: x = 5 is type correct if x is a variable of type "int".
- array subscripts: A[i] = 0 is correct if A has base-type "int".
- pointer dereferences: *x = 23 is correct if x has type "pointer to int".

I have a little function IsLValue(node) that says if a given tree node satisfies one of these conditions. I call this on the left child of any assignment node and throw an error if it returns False.