Procedures

Having added support for conditionals and loops to our interpreter, it is time to add procedures to our language. Again, we'll be borrowing from Pascal syntax, but with various changes to make our work a bit simpler. As always, back up all your work to somewhere safe before going ahead with these changes.

Exercise 1: Simple Procedures

Consider the following code, which prints the number 5 using the `add3` procedure. We'll save arguments and return values for later.

```
PROCEDURE add3();
count := count + 3;

BEGIN
   count := 2;
   ignore := add3();
   WRITELN(count);
END;
.
```

The grammar below has been extended to support procedures.

program → **PROCEDURE id ( ) ;** *stmt program* | *stmt* **.**
stmt → **WRITELN (** *expr* **) ;** | **BEGIN** *stmts* **END ;** | **id :=** *expr* **;**
| **IF** *cond* **THEN** *stmt* | **WHILE** *cond* **DO** *stmt*
stmts → *stmts stmt* | ε
expr → *expr* **+** *term* | *expr* **-** *term* | *term*
term → *term* ***** *factor* | *term* **/** *factor* | *factor*
factor → **(** *expr* **)** | **-** *factor* | **num** | **id ( )** | **id**
cond → *expr relop expr*
relop → **=** | **<>** | **<** | **>** | **<=** | **>=**

Notice that a *factor* consisting of an **id** followed by parentheses now indicates a procedure call. More importantly, until now a program has simply been a *stmt* (usually a `BEGIN`/`END` block) followed by a period. But now each *program* may begin with an arbitrary number of procedure declarations, eventually followed by a *stmt* and a period. Finally, the body of each procedure declaration is a single *stmt* (usually a `BEGIN`/`END` block).

Here are some of the changes you'll need to make to your interpreter in order for it to handle simple procedures correctly:

- Add a `ProcedureDeclaration` object to your `ast` package. What instance variables will a `ProcedureDeclaration` need?

- Add a `ProcedureCall` object to your `ast` package. What instance variables will a `ProcedureCall` need?

- Modify `Environment` so that, in addition to mapping names to variable values, it also maps procedure names to declarations. (Yes, the same name may serve as both a variable name and a procedure name.) Also implement the methods `getProcedure` and `setProcedure`.

- Implement a `Program` class, which will serve as the root of your AST. What instance variables will a `Program` need?

- Modify your `Parser` class, including adding a method `parseProgram`, which should return a `Program` object. This method should keep parsing procedure declarations as long as the current token is `PROCEDURE`. Then it should parse a single statement.

- Modify your interpreter so that it correctly handles the execution of a program, including both procedure declarations and procedure calls. Evaluating a procedure call should (1) look up the corresponding declaration from the environment, (2) execute the body of that procedure, and (3) return the value 0 (for now).

Test that you can execute simple programs with zero, one, or more procedures like the one shown earlier.

Exercise 2: Passing Arguments to Procedures

The procedure `countUp` below prints out all numbers from `count` up to `max`.

```
PROCEDURE countUp(count, max);
IF count <= max THEN
BEGIN
   WRITELN(count);
   ignore := countUp(count + 1, max);
END;
```

The expanded grammar below allows for a procedure declaration to take in zero or more parameters, and for a procedure call to pass the values of zero or more expressions as arguments.

program → **PROCEDURE id (** *maybeparms* **) ;** *stmt program* | *stmt* **.**
maybeparms → *parms* | ε
parms → *parms* **, id** | **id**
stmt → **WRITELN (** *expr* **) ;** | **BEGIN** *stmts* **END ;** | **id :=** *expr* **;**
| **IF** *cond* **THEN** *stmt* | **WHILE** *cond* **DO** *stmt*
stmts → *stmts stmt* | ε
expr → *expr* **+** *term* | *expr* **-** *term* | *term*
term → *term* ***** *factor* | *term* **/** *factor* | *factor*
factor → **(** *expr* **)** | **-** *factor* | **num** | **id (** *maybeargs* **)** | **id**
maybeargs → *args* | ε
args → *args* **,** *expr* | *expr*
cond → *expr relop expr*
relop → **=** | **<>** | **<** | **>** | **<=** | **>=**

Here are some of the changes you'll need to make to your interpreter in order for it to handle passing arguments to procedures:

- Modify `ProcedureDeclaration`. What additional information must each `ProcedureDeclaration` store?

- Modify `ProcedureCall`. What additional information must each `ProcedureCall` store?

- Modify `Parser` so that it correctly parses procedure calls with zero or more arguments, and declarations with zero or more parameters. In both cases, a comma token will tell you to keep parsing, while a closed parenthesis will tell you you've reached the end.

- Modify your interpreter to evaluate procedure calls with zero or more arguments. Evaluating a procedure call should (1) look up the corresponding declaration from the environment, (2) evaluate each argument, (3) in the environment, assign each of the argument values to the corresponding parameter name, (4) execute the body of the procedure, and (5) return the value 0 (for now). Consider the following simple program.

```
PROCEDURE printSquare(n);
WRITELN(n * n);

BEGIN
   x := 1;
   ignore := printSquare(x + 2);
END;
.
```

  Before the call to `printSquare`, the environment contains the single variable `x` and its value 1. To evaluate the call to `printSquare`, first look up the declaration associated with the name `printSquare`. Evaluate the argument `x + 2`. In the environment, assign the argument value 3 to the parameter name `n`. The environment now has values for both `x` and `n`. (We will improve on this plan in the next exercise.) Next, execute `WRITELN(n * n)`. Finally, return the value 0 as the value of `printSquare(x + 2)`.

Test that you can now pass zero, one, or more arguments to a procedure.

Exercise 3: Variable Scope

Consider the following program. A seasoned programmer would expect this code to print a 5 followed by a 3. However, our interpreter prints the value 5 twice because the call to `print` overwrites the value of `n`. The problem is that our simplistic environment model features only a single list of variables. The solution is to create multiple connected environments.

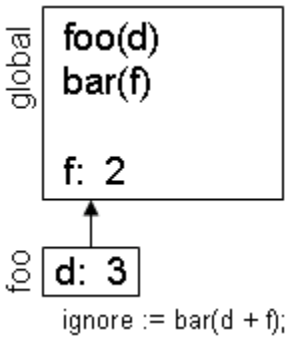
```
PROCEDURE print(n);
WRITELN(n);

BEGIN
   n := 3;
   ignore := print(5);
   WRITELN(n);
END;
.
```

We'll use the following more complex program to illustrate the behavior of the enhanced environment model. Think about what values should be printed by this code.

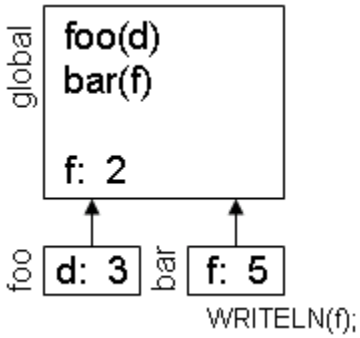
```
PROCEDURE foo(d);
ignore := bar(d + f);

PROCEDURE bar(f);
WRITELN(f);

BEGIN
   f := 2;
   ignore := foo(3);
   WRITELN(f);
END;
.
```

When this program runs, a global environment is created, as shown below. It contains all procedure declarations. This is also where the value of the global variable `f` is stored prior to the call to procedure `foo`.

The call to `foo` creates a new environment, hanging off the global one, as shown below. In other words, its parent is the global environment. This is where `foo`'s parameter `d` is associated with the value 3. The body of procedure `foo` is executed in this new environment. From here, `d` has the value 3 and `f` has the value 2, so `d + f` is 5.



Next, `foo` calls procedure `bar`, which also results in a new environment hanging off the global one. In the new environment, the parameter `f` is bound to the argument value 5. Notice that the global variable `f` is unaffected by this call. The body of `bar` is executed in this new environment. From here, `d` is undefined and `f` has the value 5. Thus, the statement `WRITELN(f)` prints out 5. When these procedures terminate and control returns to the main `BEGIN`/`END` block, the variable `d` is undefined again, and `f` is still 2.



There are many ways we could implement this model. One way is to modify the `Environment` class so that it stores another `Environment` as its parent. The parent of the global environment would be `null`. A call to *any* `Environment`'s `getProcedure` method should retrieve a procedure from the global environment (root). Likewise a call to `setProcedure` should store a procedure in the global environment. A call to `getVariable` should look first in the current environment, and should then search the parent if the variable is not found.

A call to the new method `declareVariable` should declare a variable to have a given value *in the current environment*. This is exactly what `setVariable` used to do. `setVariable` should continue to work as before, but with one significant change: If `setVariable` is called on a child environment in which that variable has not already been declared, it will set the value of the variable *in the global environment*.

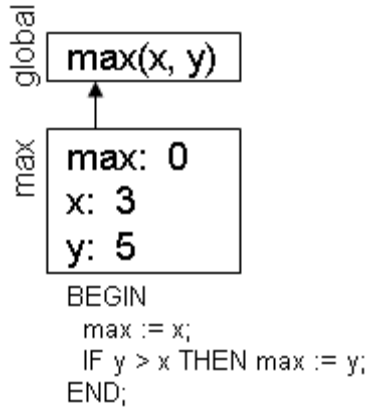
Once you have modified the `Environment` class (or classes), modify your interpreter to handle procedure calls in this new environment model. Specifically, a procedure call should (1) create a new environment hanging from the global environment, (2) declare the parameters in this new environment, (3) execute the body of the procedure in the new environment, and (4) return the value 0.

Exercise 4: Return Values

Pascal distinguishes between functions (which return values) and procedures (which don't), but our type-free language will ignore this distinction and refer to everything as a procedure. Pascal uses the following goofy syntax for returning values, which we will adopt in our interpreter.

```
PROCEDURE max(x, y);
BEGIN
   max := x;
   IF y > x THEN max := y;
END;

WRITELN(max(3, 5));
.
```

This program prints 5, the maximum of the values 3 and 5. As the following environment diagram illustrates, when `max` is called, the new environment contains the parameters `x` and `y`. However, the new environment also contains a new variable named `max`—*the same name as the procedure being called*. By default, we'll set this value to be 0. When the procedure terminates, this value will serve as the procedure's return value. If we never set the value of this variable, the original value 0 is returned. In our `max` procedure, the first line sets the value of `x` as the value to return, *but does not return it*. Instead, execution continues to the end of the procedure. Only then is the value returned.



Modify your interpreter so that it declares a variable with the same name as the procedure in the new environment, and then returns this value as the value of the procedure-call expression.

If You Finish Early

a procedure call as a statement
the EXIT statement
local variables