Lab 7: Compiling Procedures

Exercise 1: Simple Procedure Declarations

We're going to begin by compiling programs that use only very simple procedures, and gradually work our way up to fancier ones. And as before, we're aiming only for correctness and not efficiency. We'll start with the simplest procedures—ones that:

- have no parameters
- have no local variables
- do not call other procedures
- do not return a value

Let's start by compiling programs like the following one, which declares a simple procedure declaration but never calls it. On the right is the desired MIPS code output by the compiler. The portion of the MIPS code that corresponds to the declaration of procedure `foo` begins with the label `procfoo`. At the end of the subroutine, the "jr $ra" instruction tells the MIPS emulator to jump back to the return address (which will be set when we start calling subroutines).

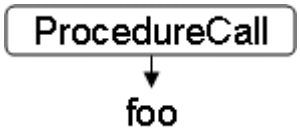
```
PROCEDURE foo();
WRITELN(1);

BEGIN
END;
.
```

```
.text
   .globl __start
__start:
   li $v0 10
   syscall
procfoo:
   li $v0 1
   move $a0 $v0
   li $v0 1
   syscall
   la $a0 newline
   li $v0 4
   syscall
   jr $ra
   .data
newline:
   .asciiz "\n"
```

Go ahead and write the `compile` method for your `ProcedureDeclaration` class, so that it can compile such simple declarations. Check that the output MIPS code looks correct.

Exercise 2: Simple Procedure Calls

In our first attempt to compile procedure calls, we'll output just a single `jal` instruction, which will initiate a jump to the procedure and save the next instruction address in the register `$ra.`

`foo()`		`jal procfoo`
Pascal	*AST*	*MIPS*

Go ahead and write the compile method for your `ProcedureCall` class, so that it outputs a single `jal` instruction. Test that you can compile and run the following simple program. (What value will be stored in `ignore` when your compiled program terminates?)

```
VAR ignore;

PROCEDURE foo();
WRITELN(2);

ignore := foo();
.
```

Exercise 3: Procedures Calling Procedures

Without modifying your compiler, compile and run the following Pascal program.

```
VAR ignore;

PROCEDURE foo();
BEGIN
   ignore := bar();
   WRITELN(-3);
END;

PROCEDURE bar();
WRITELN(3);

ignore := foo();
.
```

Can you explain the behavior you observe?

In procedure `foo`, the `jal` instruction that calls procedure bar clobbers the original return address in register `$ra`. After printing `-3`, procedure `foo` tries to return to the main program, but instead returns back to the point where `bar` returned to—causing it print `-3` again, and again, etc.

The solution, of course is to save `$ra` on the stack before each procedure call, and to restore it back to `$ra` afterward. Modify your compiler so that it does exactly this. Then, compile and run the Pascal program above, and test that it works correctly now.

Exercise 4: Passing Arguments

When you wrote MIPS programs by hand, you passed argument values in registers `$a0` through `$a3`. We might consider using this same convention in our compiler, and it will turn out to be the most efficient approach. But in terms of correctness, passing arguments in registers has a number of drawbacks relative to pushing them onto the stack:

- If there a procedure requires more than 4 arguments, the additional arguments must be placed on the stack. Why not just store all arguments on the stack?

- If a procedure calls another procedure, the first procedure must first save the values of argument registers on the stack. Why not just keep those argument values on the stack in the first place?

- If we want to allow users to declare additional local variables inside a procedure, we'll need to store their values on the stack. This will be easy if we're already storing argument values on the stack.

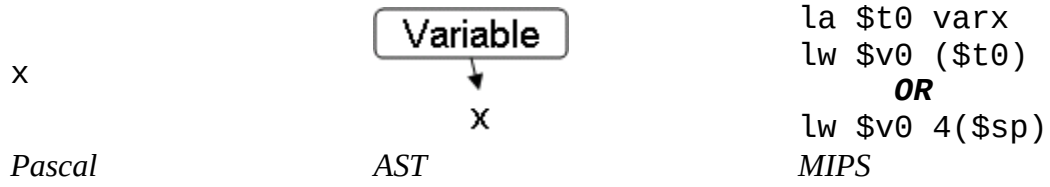
Modify your compiler so that the output MIPS code evaluates each of the procedure call's arguments and pushes those values onto the stack. When the procedure call below is evaluated, the values `4`, `5`, and `6` should be pushed onto the stack *in that order*, so that `6` is at the top of the stack. After the procedure call, the output MIPS code should pop the argument values off the stack.

```
(VAR ignore;)

PROCEDURE foo(x, y, z);
BEGIN
END;

ignore := foo(4, 2 + 3, 2 * 3);
.
```

Compile and use the MIPS emulator to step through the above program. Make sure the correct values appear in the correct positions on the stack.

Exercise 5: Remembering Procedure Context

We are now faced with a problem. When we compile the variable `x` (for example), we currently assume it's a global variable, and we emit code to load the value from address `varx`. But, if `x` is a procedure parameter (or any local variable), then we need to find its value in the stack.



How can the compiler determine if the variable is local or global? The simplest solution is to have the `Emitter` class keep track of which procedure declaration is currently being compiled. So, when we compile a variable, we'll first check the current procedure context to determine if that variable is one of the procedure's parameters (or other local variable). If so, we'll emit code to find the value in the stack, and otherwise we'll emit code to fetch the value of a global variable.

Modify your `Emitter` class so that it remembers the current `ProcedureDeclaration` (which will be `null` if we're not currently compiling a procedure). Then implement the following simple `Emitter` methods. We'll make use of these methods in the next exercise.

```
//remember proc as current procedure context
public void setProcedureContext(ProcedureDeclaration proc)

//clear current procedure context (remember null)
public void clearProcedureContext()
```

Exercise 6: Setting Procedure Context

Modify your `ProcedureDeclaration` class's `compile` method. At the beginning of the `compile` method, the `ProcedureDeclaration` should set itself as the `Emitter`'s procedure context. The end of the `compile` method should clear the `Emitter`'s procedure context.

Exercise 7: Identifying Local Variables

Now that the `Emitter` knows which `ProcedureDeclaration` (if any) is currently being compiled, we can write Emitter's isLocalVariable method, which returns true if the given variable corresponds to a local variable name (which for now means that it's the name of one of the current procedure declaration's parameter names) or a global variable name (we'll assume all unknown names are global).

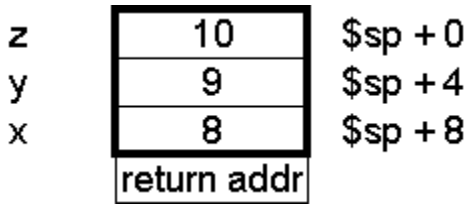
```
public boolean isLocalVariable(String varName)
```

Exercise 8: Offsets of Local Variables

Given the name of a local variable, we need to know where in the stack to find that variable. Consider the following program.

```
VAR ignore;

PROCEDURE foo(x, y, z);
BEGIN
END;

ignore := foo(8, 9, 10);
.
```

The procedure call `foo(8, 9, 10)` pushes the values `8`, `9`, and `10` onto the stack as follows:



These top three values on the stack correspond to local variables `x`, `y`, and `z` (with the procedure's first parameter `x` appearing lowest on the stack). Together, they constitute the procedure call's *stack frame*. The address of the last parameter is found in register `$sp`, the next to last in `$sp` + 4, and so on, as shown above. We'll write the `Emitter` method `getOffset` to determine the offset from `$sp`. For the example above,

`getOffset("x")` returns `8`,
`getOffset("y")` returns `4`, and
`getOffset("z")` returns `0`.

In writing the `getOffset` method in the `Emitter` class, you'll first need to determine the index of the parameter whose name matches the given `localVarName`. From there, you'll need to find a function to map from that index to the corresponding offset. Make sure your function will work correctly for the example procedure call shown earlier.

```
//precondition:  localVarName is the name of a local
//               variable for the procedure currently
//               being compiled
public int getOffset(String localVarName)
```

Exercise 9: Compiling Local Variables

You can now go ahead and modify the `Variable` class's `compile` method so that it handles both local and global variables. Go ahead and test that you can compile and run the following program. What should it print?

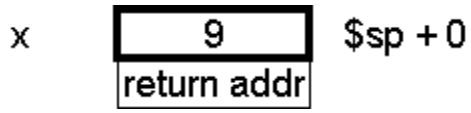
```
(VAR a, b;)

PROCEDURE foo(b, c);
BEGIN
   WRITELN(a);
   WRITELN(b);
   WRITELN(c);
END;

BEGIN
   a := 9;
   b := 10;
   a := foo(11, a + 3);
END;
.
```

Exercise 10: Stack Attack!

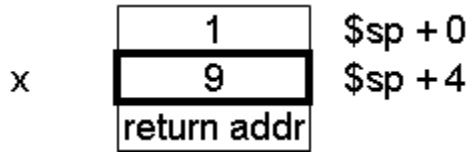
What should the following program print?

```
VAR ignore;

PROCEDURE foo(x);
WRITELN(1 + x);

ignore := foo(9);
.
```

Go compile and run this program. Why didn't it print the correct value? Here's what the stack looked like when we first called `foo(9)`:



But the compiled code for the expression `1 + x` pushes the value `1` onto the stack before evaluating `x`, leaving the stack as follows.



This means that the variable `x` is no longer associated with `$sp` + 0. Now that the stack has grown, `x`'s value is found in `$sp` + 4. How can we fix this problem?

One way is to have the `Emitter` class remember how many additional values have been pushed onto the stack since the beginning of the last procedure declaration. (We might name this new instance variable something like `excessStackHeight`.) This value should be reset to `0` whenever `setProcedureContext` is called. Next, the `Emitter` class's `emitPush` and `emitPop` methods will need to adjust this value. And finally, we'll need to modify `getOffset` to take this excess stack height into account.

Go ahead and make all these changes, and test that the program above now compiles and runs correctly.

Exercise 11: Assigning to Parameters

The `foo` procedure below changes the value of its parameter `x`. Go ahead and modify your compiler to support this functionality. Test that you can compile and run this program.

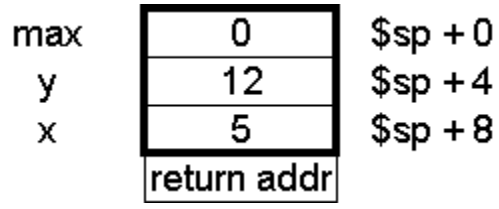
```
VAR ignore;

PROCEDURE foo(x);
BEGIN
   x := x + 1;
   WRITELN(x);
END;

ignore := foo(10);
.
```

Exercise 12: Returning Values

Consider the following program, whose `max` procedure must return a value.

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

BEGIN
   WRITELN(max(5, 12));
   WRITELN(max(13, 7));
END;
.
```

We can handle the return value by pushing an extra 0 onto the stack at the beginning of the procedure declaration, *before setting the procedure context*, so that the stack appears as shown below.



We'll now need to modify the `Emitter`'s `isLocalVariable` method to return `true` for the name of the procedure. We'll also need to modify `getOffset` to take this extra variable into account. Now we can assign values to the procedure name in the code emitted for the body of the procedure declaration.

Finally, at the end of the emitted code for a procedure declaration, we'll need to pop off that return value into `$v0` before returning.

Go ahead and make this changes, and test that you can compile and run the program above.

Exercise 13: Declaring Local Variables

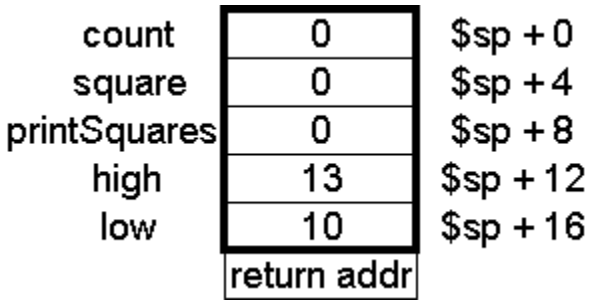
Finally, we would like to be able to declare additional local variables in a procedure, as shown below. Here, the variables `count` and `square` are declared as local variables inside `printSquares`.

NOTE – We can discover the local variables. There should never be an excess stack height when we are making an assignment since the rhs must be fully resolved before hand. We will stick to using the VAR statement to define local variables and it must be at the top of the procedure declaration. We will need to allow excessStackHeight to increment as we reserve space for local variables so that we can still find the parameters correctly in the stack.

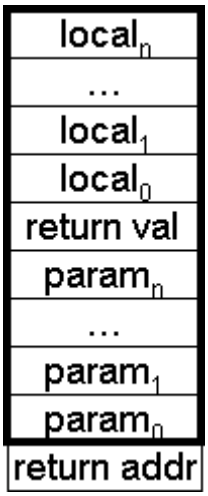
```
VAR count, ignore, times;

PROCEDURE printSquares(low, high);
VAR count, square;
BEGIN
   count := low;
   WHILE count <= high DO
   BEGIN
      square := count * count;
      WRITELN(square);
      count := count + 1;
      times := times + 1;
   END;
END;

BEGIN
```

```
   count := 196;
   times := 0;
   ignore := printSquares(10, 13);
   WRITELN(count);
   WRITELN(times);
END;
```

As before, we can store the values of these local variables on the top of the stack as follows, initializing each local variable to `0`.



In general, the stack will be laid out as follows.



But of course you'll need to implement many changes before local variables work, including changes to:

- `ProcedureDeclaration`'s instance variables
- `Parser`'s `parseProcedureDeclaration` method
- `Emitter`'s `isLocalVariable` method
- `Emitter`'s `getOffset` method
- `ProcedureDeclaration`'s `compile` method