Lab 6: Code Generation

In this lab, we'll implement a simple compiler, which will read in a Pascal-like program, and output an equivalent MIPS program. *Do not attempt to optimize your output code.* Although you may be tempted to worry about efficiency at this stage, you should focus only on finding the simplest way to generate *correct* code. Real compilers first generate correct but inefficient code, and only optimize it later. Likewise, we'll save optimization for later in the course.

BEFORE YOU START THIS LAB, BACK UP YOUR WORK TO A SERVER!

Background: `Emitter`

Download Emitter.java. Initially, this simple class will enable you to write to a file as follows.

```
Emitter e = new Emitter("testfile.txt");
e.emit("first line of file");
e.emit("second line of file");
e.close();
```

Our compiler will use an `Emitter` to output MIPS instructions to a file. As we compile more complex Pascal programs, we'll expand the role of the `Emitter` class.

Exercise 1: Emitting a Simple MIPS Program

Add a `compile` method to your `Program` class that takes in an output file name and uses an `Emitter` to write the following code to that file.

```
    .text
    .globl main

main:  #QTSPIM will automatically look for main

    # future code will go here

    li $v0 10
    syscall   # halt
```

Test that you can use SPIM to run your output code (which, of course, should do nothing).

Exercise 2: The Plan

Just as we wrote an `execute` or `evaluate` method for each of the components of our abstract syntax tree, we will now write a `compile` method for each component. Each compile method will take in an `Emitter` and use it to emit the sequence of MIPS instructions corresponding to that AST component. Rather than write all of these methods at once, we're going to write them one at a time, testing each one before moving on to the next. An easy way to go about this is to add the following method to your abstract `Expression` and `Statement` classes.

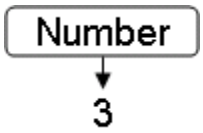
```
public void compile(Emitter e)
{
   throw new RuntimeException("Implement me!!!!!");
}
```

Now, we'll override this method in each AST component, one by one. And if we forget to override one, a friendly error message will let us know. Go ahead and add this method to your abstract `Expression` and `Statement` classes.

Exercise 3: Numbers

What should happen when we compile an expression? Well, a sequence of instructions should be emitted that compute the value of that instruction. But, assembly instructions never "return" a value. Instead, the best we can do is agree on where to store that computed value. By convention, we'll always leave the computed value in register `$v0`.

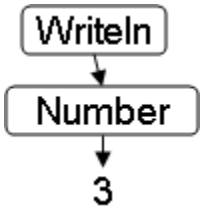
Let's start by compiling the simplest possible expression: a number. Suppose we wish to compile the number `3`. Then we should simply emit a MIPS instruction that loads the value 3 into register `$v0`, as shown below.

`3`		**`li $v0 3`**
Pascal	*AST*	*MIPS*

Go ahead and write the `compile` method for your `Number` class. We'll hold off on testing for a couple exercises.

Exercise 4: `WRITELN` Statements

Compiling statements is pretty-straightforward. Let's start with `WRITELN`. It's `compile` method should emit a sequence of instructions to print the value of some expression, as shown below.

`WRITELN(3);`		`li $v0 3` **`move $a0 $v0`** **`li $v0 1`** **`syscall`**
Pascal	*AST*	*MIPS*

Notice that the first MIPS instruction is the result of asking the nested expression `3` to compile itself. (There is a subtle difference between the behaviors of the original Pascal code and the generated MIPS code above. Can you spot it?)

Go ahead and write the `Writeln` class's `compile` method.

Exercise 5: My First Compiled Program

Recall that each Pascal program consists of a single main statement. Modify your `Program` class so that its `compile` method also compiles this statement. Now compile the following program, and test it runs correctly in SPIM.

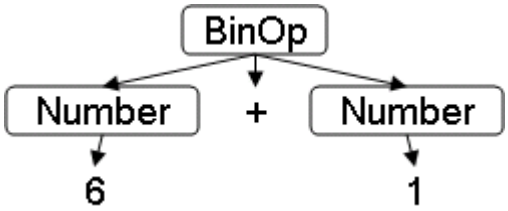
```
WRITELN(5);
.
```

Exercise 6: Newlines

Fix your `Writeln` class's `compile` method so that, after printing the value of the nested expression, the resulting MIPS code also prints a newline character. (Hint: How can you print the string `"\n"` in MIPS?) When you're done, go ahead and test that the cursor moves to the next line when you run your generated code.

Exercise 7: `BinOp` Expressions

Here is an attempt to compile the `BinOp` expression `6 + 1`.




```
6 + 1
```

```
li $v0 6
…
li $v0 1
addu $v0 ??? $v0
```

Pascal *AST* *MIPS*

Here we have a problem. Both subexpressions generate code to leave their value in `$v0`, which means that evaluating the second expression will overwrite the value of the first one.

The following shows a simple but naive solution.

```
li $v0 6
move $t0 $v0
li $v0 1
addu $v0 $t0 $v0
```

Here, we move the value of the first expression into temporary register `$t0` before evaluating the second expression. But what would happen if we used this strategy to compile the expression `6 + 1 * 2`? First we would store 6 in `$t0`. Then, in evaluating `1 * 2`, we would store 1 in `$t0`, overwriting the value 6.

In other words, our compiled code needs to work correctly no matter how complex the second subexpression is. The only truly safe place where we can store the value of the first expression is: the stack! So, compiling `6 + 1` will generate the following code.

```
li $v0 6

subu $sp $sp 4
sw $v0 ($sp)     # push $v0

li $v0 1

lw $t0 ($sp)     # pop $v0
addu $sp $sp 4

addu $v0 $t0 $v0
```

Incidentally, it can be very helpful to include comments and newlines in your generated code. Also, this is not the last time you'll need to generate code for pushing and popping, so it would be a good idea to add methods for these operations to your `Emitter` class.

```
public void emitPush(String reg)
public void emitPop(String reg)
```

Go ahead and implement the `BinOp` class's `compile` method. Then compile a program like the following, and test that it runs correctly in SPIM.

```
WRITELN(1 + 2 * 3);
.
```

Exercise 8: `Block` Statements

Go ahead and write the `Block` class's `compile` method, and test that you can correctly compile a program like the following one.

```
BEGIN
   WRITELN(-3 + 11);
   WRITELN(16 / 2);
END;
.
```

Exercise 9: Declaring Variables

We can greatly simplify the problem of compiling programs with variables if we modify our Pascal syntax to require variable declarations using the following syntax.

```
VAR x, y;

BEGIN
   x := 3;
   y := 2 * x;
   WRITELN(x + y);
END;
.
```

Thus, a program will consist of (1) a list of global variable names, (2) a list of procedure declarations, and (3) a main statement. You should therefore go ahead and modify your `Program` class and your `parseProgram` method.

If we do not declare variables in this manner, we will need to be able to discover the variables and somehow build the data section. This can be done using a symbol table that is similar to what we did with the environment. The trouble here is with forward references. When we interpret the program, we store variables within the interpreter, but when we compile, we must know where in memory the variables are actually stored. Adding the `Var` statement to our grammar simplifies this whole process and that is the approach we shall take.

Exercise 10: Compiling Variable Declarations

There are several places we could store variable values in our target assembly code:

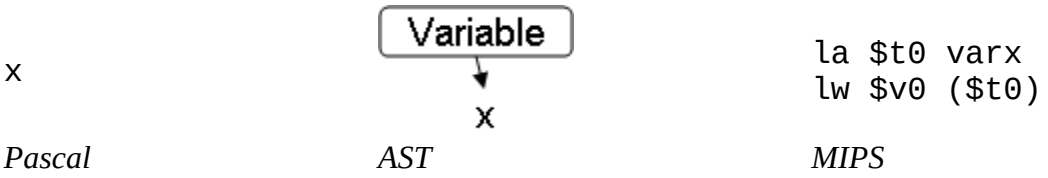
- registers
- the stack
- the beginning of the `.data` section of memory

Although registers may be the most obvious and efficient choice, in practice they are by far the most difficult to use. And if at any point your program simultaneously uses more variables than you have registers, you'll need to store some of the values to RAM anyway. The stack is definitely the best place to store local variables used in procedure calls, but this is tricky business, too.

The easiest solution for now will be to store our global variables at the beginning of the `.data` section of memory. Thus, when you compile your program, you'll also have to list all global variables in the data section. It's a good idea to use a naming convention like `varcount` for `count`, and so on, because otherwise it would be disastrous if a user's variable name matched one of the jump target labels in your compiled code. Zero would be a good default value.

Exercise 11: `Variable` Expressions

In order to find the value of a variable, your target MIPS code will need to load the value associated with that variable's address into register `$v0`, as shown below.



Go ahead and implement your `Variable` class's `compile` method.

Exercise 12: `Assignment` Statements

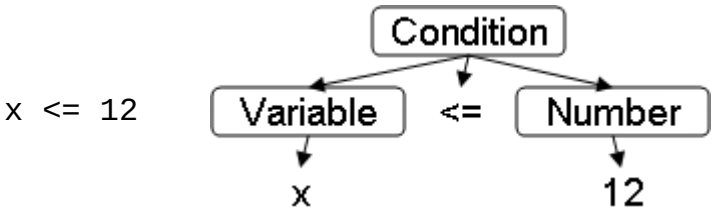
Implement the `Assignment` statement's `compile` method, and test that you can run the code produced when you compile programs like the following.

```
VAR x, y;

BEGIN
   x := 3;
   y := 2 * x;
   WRITELN(x + y);
END;
.
```

Exercise 13: Conditions

Since both `If` and `While` statements use conditionals, it will be helpful to write a `compile` method for the `Condition` class that we can reuse. A good plan is to have `Condition`'s `compile` method take in both an `Emitter` and a target label (a `String`). So, if asked to compile the condition `x <= 12` with target label `endif1`, the generated code would appear as follows.



Notice that, when testing for `x <= 12`, the generated code branches if `x` is *greater* than 12. This will simplify the work we need to do in compiling `If` and `While` statements. Go ahead and implement the compile method for your `Condition` class.

Exercise 14: Labels

To compile an `If` statement, you'll need a label name to brach to. Since the full compiled program may require many labels, we'll need to ensure that the label name we choose is unique. A good plan is to use the label name `endif1` for the first `If` you compile, `endif2` for the second one you compile, and so on. We'll keep track of label numbers in the `Emitter` class. Add the following method to your `Emitter` class, which should return 1 the first time it's called, 2 the next time, then 3, etc.

```
public int nextLabelID()
```

Exercise 15: `If` Statements

Now write the `If` class's `compile` method, and test that, when you compile a program like the following one, it runs correctly in SPIM.

```
BEGIN
   IF 14 = 14 THEN
   BEGIN
       IF 14 <> 14 THEN WRITELN(3);
       IF 14 <= 14 THEN WRITELN(4);
   END;
   IF 15 > 14 THEN WRITELN(5);
END;
.
```

Exercise 15: `While` Statements

Now write the `While` class's `compile` method, and test that, when you compile a program like the following one, it runs correctly in SPIM.

```
VAR count;

BEGIN
   count := 1;
   WHILE count <= 15 DO
   BEGIN
      WRITELN(count);
      count := count + 1;
   END;
END;
.
```