Programming Assignment 6

Due at 11pm on May 3rd, 2020

Overview

For this assignment you will write a code generator that generates MIPS assembly code (suitable as input to the Spim interpreter) for Wumbo programs represented as abstract-syntax trees.

Specifications

General information

Similar to the fourth and fifth assignments, the code generator will be implemented by writing `codeGen` member functions for the various kinds of AST nodes. *See the on-line Code Generation notes (as well as lecture notes) for lots of useful details*.

In addition to implementing the code generator, you will also update the main program so that, if there are no errors (including type errors), the code generator is called after the type checker. The code generator should write code to the file named by the second command-line argument.

Note that your main program should no longer call the unparser, nor should it report that the program was parsed successfully.

Also note that you are *not* required to implement code generation for:

- `structs` or anything `struct`-related (like dot-accesses)
- repeat statement

Getting started

Start by downloading p6.zip. After unzipping it, you will see all the files required for the project.

Some useful code-generation methods can be found in the file `Codegen.java`. Note that to use the methods and constants defined in that file you will need to prefix the names with `Codegen.`; for example, you would write: `Codegen.genPop(Codegen.T0)` rather than `genPop(T0)`. (Alternatively, you could put the declarations of the methods and constants in your `ASTnode` class; then you would not need the `Codegen` prefix.) Also note that a `PrintWriter p` is declared as a static public field in the `Codegen` class. The code-generation methods in `Codegen.java` all write to `PrintWriter p`, so you should use it when you open the output file in your main program (in `P6.java`); i.e., you should include:

```
Codegen.p = new PrintWriter(args[1]);
```

in your main program (or `ASTnode.p` if you put the declarations in the `ASTnode` class). You should also close that `PrintWriter` at the end of the program:

```
Codegen.p.close();
```

Spim

The best way to test your MIPS code is using the simulator SPIM (written by at-the-time UW-Madison Computer Science Professor Jim Larus). The class supports two versions of spim:

1. A command line program, called **spim**
   Accessing spim:
   - Installed on the lab computers at `~cs536-1/public/tools/bin/spim`
2. A GUI-driven program, called **QtSpim**
   Accessing QtSpim:
   - Installed on the lab computers at `~cs536-1/public/tools/bin/QtSpim`
   - Avaiable as a binary package here
   - An online version of SPIM (we haven't tested it yet) is available at this url

Both of these tools use the same backend, but we recommend using QtSpim since it is much more of a modern interface. Generally, it should be enough to run

```
~cs536-1/public/tools/bin/QtSpim -file <mips_code.s>
```

where *`mips_code.s`* is the name of your source file (i.e., the one containing your MIPS code).

And use the interactive help or menus from there. However, if you want more guidance on using spim, you can check out this (fairly old) Reference Manual (pdf). Also, check the tutorials page for a screencast on MIPS and SPIM.

To get the Spim simulator to correctly recognize your `main` function and to exit the program gracefully, there are two things you need to do:

1. When generating the function preamble for `main`, add the label "`__start:`" on the line after the label "`main:`" (note that `__start:` contains two underscore characters).

2. When generating the function exit for `main`, instead of returning using "`jr $ra`", issue a `syscall` to exit by doing:

   ```
   li $v0, 10
   syscall
   ```

   (Note that this means that a program that contains a function which calls `main` won't work correctly, which will be ok for the purposes of this project.)

Here is a link to an example Wumbo program and the corresponding MIPS code.

Changes to old code

Required changes:

1. Add to the name analyzer or type checker (your choice), a check whether the program contains a function named `main`. If there is no such function, print the error message: "No main function". Use 0,0 as the line and character numbers.

2. Add a new `"offset"` field to the `Sym` class (or to the appropriate subclass(es) of `Sym`). Change the name analyzer to compute offsets for each function's parameters and local variables (i.e., where in the function's Activation Record they will be stored at runtime) and to fill in the new offset field. Note that each scalar variable requires 4 bytes of storage. You may find it helpful to verify that you have made this change correctly by modifying your unparser to print each local variable's offset.

Suggested changes:

1. Modify the name analyzer to compute and save the total size of the local variables declared in each function (e.g., in a new field of the function name's symbol-table entry). This will be useful when you do code generation for function entry (to set the SP correctly).

2. Either write a method to compute the total size of the formal parameters declared in a function, or modify the name analyzer to compute and store that value (in the function name's symbol-table entry). This will also be useful for code generation for function entry.

3. Change the definition of class `WriteStmtNode` to include a (private) field to hold the type of the expression being written, and change your typecheck method for the `WriteStmtNode` to fill in that field. This will be useful for code generation for the *write* statement (since you will need to generate different code depending on the type of the expression being output).

Non-obvious semantic issues

1. All parameters should be passed by value.

2. The *and* and *or* operators (`&&` and `||`) are *short circuited*, just as they are in Java. That means that their right operands are only evaluated if necessary (for all of the other binary operators, both operands are always evaluated). If the left operand of "`&&`" evaluates to *false*, then the right operand is not evaluated (and the value of the whole expression is *false*); similarly, if the left operand of "`||`" evaluates to *true*, then the right operand is not evaluated (and the value of the whole expression is *true*).

3. In Wumbo (as in C++ and Java), two string literals are considered equal if they contain the same sequence of characters. So for example, the first two of the following expressions should evaluate to *false* and the last two should evaluate to *true*:

```
"a" == "abc"
"a" == "A"
"a" == "a"
"abc" == "abc"
```

4. Boolean values should be output as 1 for *true* and 0 for *false* (and that is probably how you should represent them internally as well).

5. Boolean values should also be input using 1 for *true* and 0 for *false*.

Suggestions for how to work on this assignment

1. Modify name analysis or type checking to ensure that a main function is declared.

2. Modify name analysis so that the code generator can answer the following questions:

    - Is an Id local or global?
    - If local, what is its offset in its function's AR?
    - For each function, how many bytes of storage are needed for its params, and how many are needed for its locals?

3. Implement code generation for each of the following features; be sure to test each feature as it is implemented!

    - global variable declarations, function entry, and function exit (write a test program that just declares some global variables and a main function that does nothing)
    - `int` and `bool` literals (just push the value onto the stack), string literals, and `WriteStmtNode`
    - `IdNode` (code that pushes the value of the id onto the stack, and code that pushes the address of the id onto the stack) and assignments of the form `id=literal` and `id=id` (test by assigning then writing)
    - expressions other than calls
    - statements other than calls and returns
    - call statements and expressions, return statements (to implement a function call, you will need a third code-generation method for the `IdNode` class: one that is called only for a function name and that generates a jump-and-link instruction)

Handing in

Please read the following handing in instructions carefully. You will be needed to submit the entire working folder as a compressed file as given below.

lastname.firstname.lastname.firstname.P6.zip
+---+ deps/
+---+ ast.java
+---+ Wumbo.cup
+---+ Wumbo.jlex
+---+ Codegen.java
+---+ DuplicateSymException.java
+---+ EmptySymTableException.java
+---+ ErrMsg.java
+---+ Makefile
+---+ P6.java
+---+ Sym.java
+---+ SymTable.java
+---+ Type.java
+---+ lastname.firstname.lastname.firstname.P6.pdf

Please ensure that you do not include any extra sub-directories. Do not turn in any `.class` files. If you accidentally turn in (or create) extra files or subdirectories, please remove them from your submission zip file.

Joining a Canvas group with your partner (also required for those who work alone)

To faciliate grading, before you submit the compressed file, please join a Canvas group with your partner (guide) so you can both receive grades for the same submission. Since you're working on P6, please join an empty group whose name is in the form of "P6-Pair #" (type "P6" in the search bar). Note that we have created the groups for you so you only need to join an empty one with your partner. If you work alone, please join an empty group as well.

If you are working in a pair, have only one member submit the program. Include both persons' name as given above. Also, mention the teammate's name as a comment while submitting the assignment on canvas.