CS5300
Compiler

In this project you will implement a compiler for the C- language. You will start with the code formatter you wrote in the last project. There are 14 C- files to support. Each file is worth 10% of the grade. Yep, that means that you can earn as much as 40% extra credit on this project. But be warned: this project is difficult and time-consuming. Start early.

General suggestions

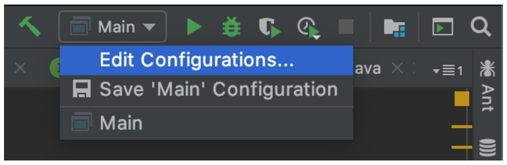
1. If you haven't yet learned out to set breakpoints and use the debugger in your IDE, this project would be a good time to learn.
2. As you develop code, consider commenting out code in the C- file you're working in to isolate only pertinent code. This helps keep your .asm files smaller and easier to debug. Don't forget to test on the entire file once you have all the features.
3. Don't worry too much about handling syntax errors. All of the C- code we'll test with will be syntactically correct. Our test files will be very similar to the 14 C- files included here.
4. To build your MIPS instructions there are a couple of different approaches you can take for efficient code:

   ```
   code.append("lw ").append(reg).append(" ").append(offset).append("($sp)\n");
   code.append(String.format("lw %s %d($sp)\n", reg, offset));
   ```

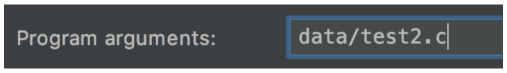
   The second approach is probably easier to read and debug.
5. This webpage can be a good resource for MIPS instructions: `www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf`.
6. Sometimes loading .asm files into Mars might take longer than you want to take. If you want to test your .asm files at the command-line then you can use the **data/test** script.

Starting out

1. Add the `MIPSResult` and `RegisterAllocator` classes (included in the project zip) to your `submit` package. Become familiar with them.
2. Add the `Main` class to the `main` package.
3. Replace your **data** directory with the one included in the project distribution.
4. Add an input file to your run configuration. In IntelliJ, you'll first open the Edit Configurations window:



Then you will add the filename of the C- file you want to compile:



Program arguments: data/test2.c

5. Add a method called `toMIPS` to your `Node` class. Most of your work will be in overriding this method in your various syntax tree node classes. The method should have the following signature:

```
MIPSResult toMIPS(StringBuilder code, StringBuilder data,
                  SymbolTable symbolTable, RegisterAllocator regAllocator)
```

6. You may want to consider creating an `AbstractNode` class and have your other node classes derive from it. If `AbstractNode` has a default implementation of `toMIPS()` you won't have to implement `toMIPS()` in every node class right off the bat.
7. As you implement `toMIPS()` in your AST nodes you may want to default to returning a void `MIPSResult`. You can then return address or register `MIPSResults` as needed.
8. You will be expanding your `SymbolTable` class. It will need methods to compute the size of an activation record, and you might consider using it to save and restore local storage registers (`$t0-9` and `$s0-9`). You can also use it for unique labels for data. For example, you could have a method called `getUniqueLabel()` that you'll use in the implementation of `toMIPS` in the `StringConstant` class. `getUniqueLabel()` should probably use a static integer to count to make new labels since the labels will be global across the machine code.
9. *Suggestion:* when you know roughly what MIPS code you need to generate, copy an example of the MIPS code from your homework or from what we wrote in class as a comment into your Java code.

Doing the work

Following are suggestions that will hopefully help you avoid getting sidetracked with problems as you write code to support the test files. They are not step-by-step instructions.

test1.c (Hello World)

1. For `println()`, add the functon to the symbol table in the symbol table's constructor and treat `println()` as a special case in your `Call` node.
2. The implementation of `toMIPS` in `StringConstant` will return an address `MIPSResult`.
3. See **data/examples/test1.asm** for an example of what your .asm file might look like.

test2.c (Arithmetic)

1. You're ready to use registers. The `RegisterAllocator` class can help. Use the `getT()` or `getAny()` method (they do the same thing) to get an available `$t0-9` register. For this project we'll use only `$t` registers for simplicity.
2. For multiplication you will only need to support the 32 least significant bits:

   ```
   mult $a0, $a1
   mfhi $a2 # 32 most significant bits of multiplication to $a2
   mflo $v0 # 32 least significant bits of multiplication to $v0
   ```

   Alternatively you could just use the `mul` instruction.
3. For integer division you will only need to support the quotient:

   ```
   div $a0, $a1
   mfhi $a2 # remainder to $a2
   mflo $v0 # quotient to $v0
   ```

test3.c (Variables)

1. You'll be using `SymbolTable` more extensively now. Start by adding `SymbolTable` as a data member to `CompoundStatement`. You'll add it in the constructor for `CompoundStatement`.
2. The `SymbolTable` class should store the size, in bytes, of the symbol table. It should also store the offset for each variable declared in the scope. You will use this when accessing variables.
3. In your `Assignment` node you'll store the value calculated by the expression on the right hand side onto the stack using the offset for the variable (mutable) you're assigning it to.
4. In nodes that have their own symbol table, when calling `toMIPS` of child nodes, pass the proper symbol table. Also, update the stack pointer (`$sp`).

test4.c (Lexical scoping)

1. *Note:* This feature can be skipped if you have trouble with it – you don't need it for subsequent features.
2. When computing the offset for a variable in your symbol table you may not find a symbol in the current symbol table. In that case you need to traverse up the hierarchy. If you need to do this when getting an offset for a variable you need to add sizes of symbol tables to get the correct offset from the current stack pointer.
3. For the runtime environment, you should update the stack pointer (`$sp`) in `CompoundStatement`. Use the parent's symbol table to know how much to decrement the stack pointer by.
4. We also see negation for the first time in this file.
5. The stack when executing line 16 should look like this:

   ```
    addresses --->            <--- stack|
   |                    |     |  main()|
   |                 b  |  a  |  b  a  |
                        ^
                      $sp
   ```

The address of `b` is `$sp-4` and the address of `a` is `$sp+4-4` where, to get the address of `a`, we took the stack pointer, added the size of the parent activation record, and then applied the offset of `a` (that we found in the parent symbol table).

6. You need to save your registers to the stack. Do this using `RegisterAllocator.saveT()`.

test5.c (Function call)

1. You had a special case in the `Call` node for `println`. You'll now implement the rest for custom functions.
2. Be careful when using `jal`. Nested function calls (such as is found in **test5.c**) will overwrite `$ra`. So before you call `jal` you should save `$ra` to a temporary register.
3. In this program, line 2 will be executed twice. The stacks will look like this (keep in mind there's nothing on the stack for `main()` or `foo()` since they don't have any variables):

```
addresses --->             <--- stack|
|                                foo |
|                                    |
                                     ^
                                    $sp
```

```
addresses --->             <--- stack|
|                       foo |    fum |
|                           |  b  a  |
                            ^
                           $sp
```

test6.c (Parameters)

1. The C- grammar says that any statement can be in a function declaration, but you can assume that the statement will be a `CompoundStatement`. This will simplify things when it comes to formal arguments, which should be placed in the scope of the compound statement.
2. When putting parameters on the stack, a good approach is to first update `$sp` to point to the new stack. You'll use the size of the current symbol table (in bytes) as well as how many registers you temporarily saved to the stack to determine what the new stack pointer should be. Then iterate through the formal parameters. After you evaluate each parameter, place it on the stack.
3. At line 2 the stack should look like this:

```
addresses --->             <--- stack|
|                       add |   main |
|                     y  x  |  b  a  |
                            ^
                           $sp
```

test7.c (Return values)

1. In `ASTVisitor.visitFunDeclaration`, add a special symbol to the child symbol table (the one that was created by the `CompoundStatement`) called "return." This will give you a place on the stack to store the return value.

2. At line 2 your stack should look like this. The `r` is for the return value.

   ```
    addresses --->            <--- stack|
   |                           identity |
   |                              r  x  |
                                        ^
                                       $sp
   ```

test8.c (Nested function calls)

1. If everything was done properly this file shouldn't require additional code.

test9.c (Boolean comparators and if statements)

1. Typically we treat zero as true and non-zero as false.
2. You'll probably want to use the *set on less than* instruction:

   ```
   slt rd, rs, rt # if rs < rt, rd = 1; else rt = 0
   ```

   You'll also want to use `beq` and/or `bne`.
3. You will probably want to use `getUniqueLabel()` from the symbol table.

test10.c (Saving registers)

1. You may run out of registers. To fix this, judiciously call `clear()` in `RegisterAllocator`. You'll do this after handling a `MIPSResult`.

test11.c (While loops)

1. Similar to the `if` statement.

test12.c (The reward)

1. This shouldn't require extra work if everything else was done properly. (Fingers crossed!)

test13.c (Arrays)

1. Note in the grammar that only constant values can be used in declaring array sizes. This is how Fortran works. The advantage is that you know at compile time how big the array is and can allocate exactly that much memory on the stack.
2. Consider modifying your `SymbolInfo` class and possibly your `SymbolTable` class to support arrays if you haven't done so already. You may need to modify `visitVarDeclaration` in `ASTVisitor`. These modifications will be made to help you get the correct amount of memory to allocate for arrays.

test14.c (Passing arrays as parameters)

1. One approach to this is, for the parameter, making a note in the `SymbolInfo` class that the parameter is an address instead of a value. Regardless of how you keep track, the array should be passed on the stack as an address and not a value.

Submission

1. Set `trace=false` in the `main` class and test your code to make sure you're getting the correct output without debug output.
2. Zip your **submit** directory and submit it. Zip only that directory! Make sure to zip the *directory* and not just the *files* in the directory. You will submit a single file called **submit.zip**.