CS75 Project 3b: Complete Code Generation for C- -
Spring 2007, Meeden
Due by midnight, Monday, April 23

Conventions

- Registers `$a0` and `$v0` are reserved for system calls.
- Registers `$a1-$a3` and `$v1` are reserved for function call arguments and return values. We will assume that no function takes more than three parameters.
- On every function call we will save every register than must be preserved across calls (except the `$gp` which we will never modify).
- Registers `$t0-$t9` will be used for evaluating expressions. These registers should be reclaimed after the completion of an expression evaluation.
- Local variables will be stored in registers `$s0-$s7`.

Completing the remaining functionality

You should continue to work incrementally on the code generator. I suggest you follow the progression given below for adding the remaining functionality.

1. identifiers

   The symbol table will now play a much more significant role. It will need to be updated to include information about the various kinds of identifiers (keywords, parameters, locals, globals, functions), their types (int, char, array), and their locations. Identifiers can be located in a register or offset from some pointer to memory (`$gp` for globals and `$fp` for locals). It would be useful to have methods to test whether a particular identifier is `inRegister` or `inMemory`.

   As your code generator encounters a new function or a new block containing local variable declarations, it should initialize a new scope in the symbol table. As you leave the function or block, it should finalize the current scope in the symbol table. Each parameter or local variable should be entered into the symbol table as you encounter it.

   Your code should be able to detect duplicate definitions of the same name within the same scope, and report an error. It should also report an error if a C - - program tries to re-declare a function name or parameter name.

   (a) functions

   The symbol table should maintain information about the return type, parameter types, and parameter names. This information should be entered into the symbol table by the **parser**. The function information should be inserted at the global scope and remain in tact throughout the code generation process.

   (b) globals

   Global variables will be stored relative to the `$gp`. Remember to use **positive offsets** from this pointer.

   (c) locals

   For now, local variables will be stored in the save registers.

   (d) parameters

   Parameters will be stored in the argument registers, but may need to be spilled to the stack. When in the stack they will be stored relative to the `$fp`. Remember to use **negative offsets** from this pointer.

2. assignment

   For now we are assuming that we have simple assignment statements of the form: $lvalue = rvalue$ where the $rvalue$ is the value of an expression and the $lvalue$ is the address of an identifier or an array reference. Initially, only handle simple identifiers, we'll add array references later.

```
evaluate the right-hand side to get an rvalue
lookup the id from the left-hand side in the symbol table
if no entry exists
   report an error
if entry is in register
   move rvalue into this register
if entry is in memory
   store the rvalue into this location
```

3. read

   The read statement only accepts simple identifiers, and not array references. Reading in an integer is done with a system call. It will put the value into register `$v0`. Then you must transfer this value into the proper storage location using a very similar method to assignment.

4. break

   The break statement is only appropriate when used within a loop. To implement this feature you will need to add a label argument to your methods that handle statements and blocks. In most cases, this label will not be used, so should have a default value of `None`. In your code that processes loops, you will need to pass the loop ending label into the code that handles the loop statement.

5. function definitions

   Each function definition will have a boiler plate opening, which saves the 10 registers that must be preserved across calls, and then sets the frame pointer to the beginning of the activation record. Each function will also have a boiler plate closing, which restores the stack pointer to the head of the saved registers, restores the registers, and then returns. The frame pointer should stay fixed throughout the execution of a function, while the stack pointer may change as additional space is needed on the stack.

   If the body of a function definition contains a function call, then we will immediately spill the function's arguments to the stack, prior to evaluating the function body. You can use the helper function below to determine if a nested list `ls` contains a particular item `x`. In our case the nested list will be the AST of the function body and the particular item we will be searching for is `'funcall'`.

   If the body of a function definition does not contain a function call, then the function's arguments can remain in the argument registers. In either case the symbol table should be updated appropriately to reflect the locations of the parameters in memory.

```
def deepMember(x, ls):
    if len(ls) == 0:
        return False
    if type(ls[0]) == type(ls):
        return deepMember(x, ls[0]) or deepMember(x, ls[1:])
    if ls[0] == x:
        return True
    return deepMember(x, ls[1:])
```

6. function calls

   Here are the steps needed to handle function calls with pass by value arguments. This will need to be modified once you add the ability to pass arrays.

```
lookup the name of the callee in the symbol table
if the name is not declared or is not a function
   report an error
if number and type of arguments do not match parameters
   report an error
```

```
if currently using any temporary registers
   spill them to the stack
for each argument
   evaluate the argument expression
   move the result into an argument register
jump to the callee
if temporaries were spilled
   restore them
result of function call is in $v1
```

7. arrays

   In the symbol table you should keep track of the size and type of the array.

   Whether you are assigning a value to an array reference or simply accessing an array reference you will need to go through the following steps.

```
load the address of the base of the array into a register
evaluate the index expression
multiply the index value by 4 (using a shift left by 2 operation)
if the array is global
   add the offset to the base address
else
   subtract the offset from the base address
return the register containing the address
```

   Then you can offset this address by 0 to store or load values.