CS4121 SomeLife Compiler Project 1: An Expression Evaluator
Due Date: Friday, Feb. 15, 2013 at *5:00pm*

Purpose

The purpose of this project is to gain experience in giving meaning to a programming language by generating Mips assembly for a subset of SomeLife. Specifically, you will be generating assembly for integer I/O operations, integer arithmetic and logical expressions and assignment statements. Completely read this document and the SomeLife language specification.

Project Summary

In this project, you will add actions to the provided parser that will do the following

1. Assign space for global integer variables declared in a SomeLife program.
2. Generate assembly pseudo-ops for string constants used in a SomeLife program.
3. Generate assembly to print string constants.
4. Generate assembly to print integers.
5. Generate assembly to read integers.
6. Generate assembly to compute integer expressions.
7. Generate assembly to compute logical expressions.
8. Generate assembly to assign values to integer variables.

The Scanner and Parser

Your code will be based on a scanner using the *flex* scanner generator system and a parser using the *lemon* parser generator system. This project must be done in C or C++. Lemon and flex generate C code.

1. Read all of the related documentation on *flex* and *lemon* following the "Web Links" on the CS4121 Canvas page.
2. Unzip and untar the attached SomeLifeProject1.tgz to your own working directory.
3. Use "make" to compile the project (the makefiles are provided).
4. You must change the actions in SomeLifeParser.y to generate Mips assembly. See the remaining sections for templates of assembly code to be generated.

Prologue and Epilogue Code

Since you are converting a SomeLife program to Mips assembly, you must begin each assembly file with any data declarations and a declaration of where the main program begins. This is done with the following code:

```
        .data
.newl: .asciiz "\n"
        .text
        .globl main
main:   nop
```

This code declares a data section with a string `.newl` that is just the newline character, followed by a text section (instructions) containing a declaration of the main routine. Each Mips assembly file should begin with this sequence. If you assign space in the static data area for variables, then the space may be allocated with directives after the ".data" directive and before the ".text" directive, or by assigning an offset off of `$gp`.

To end a Mips assembly routine, add the code

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

These instructions exit a program.

Assigning Variable Space

Memory for global variables declared in a SomeLife program may be allocated on the stack of the `main` function or in the static area. You can choose one of these two approaches.

1. Stack Allocation
   Each integer requires four bytes of space. This space is allocated by adjusting the stack pointer the requisite number of bytes. Since stacks grow in the negative direction in memory, space is allocated by subtracting from the stack pointer. The SomeLife declarations

   ```
   VAR i,j,k :INTEGER;
   ```

   require 12 bytes of space. That space is allocated on the stack with the instructions

   ```
   move $fp, $sp
   sub $sp, $sp, 12
   ```

   which should be placed *immediately* following the prologue code. Variables may be addressed as a negative offset off of the frame pointer. The first variable is located at 4 bytes off the frame pointer, `$fp` and each successive variable is located 4 bytes from that point. For example, one may load the value of `j` above with the following instructions:

   ```
   add $s0, $fp, -8
   lw $s1, 0($s0)
   ```

2. Static Area Allocation (Recommended Implementation)
   An alternative implementation (used in my notes) is to use the static data segment pointed by `$gp`. For the SomeLife declarations

   ```
   VAR i,j,k :INTEGER;
   ```

   variables i, j, and k may be addressed as a positive/negative offset off of the global pointer, `$gp`. For this project, I suggest you use non-negative offset.The first variable is located at the global pointer with offset 0, and each successive variable is located 4 bytes from that point. For example, one may load the value of `j` above with the following instruction:

   ```
   add $s0,$gp, 4
   lw $s1, 0($s0)
   ```

String Constants

A SomeLife program may use string constants in write statements. These constants are declared in the data section using the `.string` pseudo-op. For the SomeLife statement,

```
WRITE('Hello');
```

The following declaration must be added to the data section of the assembly file:

```
.string0: .asciiz "Hello"
```

The label `.string0` is implementation dependent. You may name your string constants however you wish.

Printing Strings

Printing strings requires using a system call. For a string, the system call service for printing strings is 4. Since a character string is stored in memory, you must pass the address of the string to the system call in register `$a0`. As an example, the code to implement the `write` statement in the previous section would be:

```
la $a0, .string0
li $v0, 4
syscall
```

Note that you will need to additionally print the newline character when printing any data.

Printing Integers

Printing integers is similar to printing strings except that the actual integer is passed to the system call rather than an address and the system call service is 1. As an example, to implement the statement:

```
write(7);
```

the following Mips assembly would need to be generated:

```
li $a0, 7
li $v0, 1
syscall
```

Reading Integers

To read an integer, the system call service is 5. The read value is returned in register `$v0`. Thus, to read an integer, the following instructions are needed:

```
li $v0, 5
syscall
```

Integer Arithmetic Expressions

In Mips assembly, all operations are done on registers. The best way to generate code is to store all intermediate values in Mips registers. Using the registers `$s0, ..., $s7` should be sufficient. You should not need any other temporary registers. For an operation, the operands should all be put into registers, a result register should be allocated, the operations should be performed and then the input registers should be released to be reused later. As an example, the statement

```
write(a+b);
```

might result in the code (if `a` is the first declared variable and `b` is the second)

```
add $s0, $gp, 0
lw $s1, 0($s0)
add $s0, $gp, 4
lw $s2, 0($s0)
add $s0, $s1, $s2
move $a0, $s0
li $v0, 1
syscall
```

Logic Expressions

Logic expressions are similar to arithmetic expressions. For the Mips, the value for `false` is 0 and the value for `true` is 1.

Storing Integer Variables

To store a value in a variable, first compute the address and then store the value into that location. For example, the statement

```
b = 5;
```

could be implemented with

```
li $s0, 5
add $s1, $gp, 4
sw $s0, 0($s1)
```

Requirements

Write all of your code in C or C++. It will be tested on CS lab machines (Rekhi 112, 112a, or 117) and MUST work there. You will receive no special consideration for programs which "work" elsewhere.

Input. Sample input is provided in the directory `SomeLifeProject1/input1`. To run your compiler, use the command

```
<myexecutable> <file>.sl
```

which will output to `<file>.s`
To run the assembly, you can download the Mars simulator from http://courses.missouristate.edu/KenVollmar/MARS/.

Submission. Your code should be well-documented. You will submit all of your files, by tarring up your working directory using the command

```
tar -czf SomeLifeProject1.tgz SomeLifeProject1
```

Submit the file `SomeLifeProject1.tgz` via Canvas. Make sure you do a 'make clean' of your directory before executing the tar command. This will remove all of the '.o' files and make your tar file much smaller.

An Example

Given the following SomeLife program (3.add.sl):

```
PROGRAM exprAdd;

VAR i,j,k,l : INTEGER;

BEGIN
        WRITE(10+20);
        i := 1; k := 3; l := 4;
        j := i + l + k;
        WRITE(j)
END.
```

it may be implemented with the following Mips assembly.

```
.data
.newline: .asciiz "\n"
.text
.globl main
main: nop
move $fp,$sp
li $s0, 10
li $s1, 20
add $s2, $s0, $s1
move $a0, $s2
li $v0, 1
syscall
li $v0, 4
la, $a0, .newline
syscall
add $s0, $gp, 0
li $s1, 1
sw $s1, 0($s0)
add $s0, $gp, 8
li $s1, 3
sw $s1, 0($s0)
add $s0, $gp, 12
li $s1, 4
sw $s1, 0($s0)
add $s0, $gp, 4
add $s1, $gp, 0
lw $s2, 0($s1)
add $s1, $gp, 12
lw $s3, 0($s1)
add $s1, $s2, $s3
add $s2, $gp, 8
lw $s3, 0($s2)
add $s2, $s1, $s3
sw $s2, 0($s0)
add $s0, $gp, 4
lw $s1, 0($s0)
```

```
move $a0, $s1
li $v0, 1
syscall
li $v0, 4
la, $a0, .newline
syscall
li $v0, 10
syscall
```