THE UNIVERSITY OF ARIZONA®
DEPARTMENT OF COMPUTER SCIENCE

CSc 553 : Principles of Compilation

Programming Assignment 1 (Code Generation)

Milestone 3: All of C--

Start: Fri Sep 22, 2023
Due: 11:59 PM, Fri Sep 29

1. General

This assignment involves generating code for C--. You will write code to traverse the syntax tree for each function in a program and generate three-address statements, then translate these into MIPS assembly code and output the assembly code generated. Your code generator should reuse temporaries as far as possible.

For intermediate code generation, you can use the three-address instruction set given here; a translation from this instruction set to MIPS assembly code is given here. You are not *required* to adhere to this particular three-address instruction set, and are free to deviate from it if you want; however, if you do you will have to work out the mapping to MIPS assembly code yourself.

Note : Even though the intermediate code plays a relatively small role in this assignment, future assignments will involve dataflow analysis and machine-independent optimization using the intermediate code generated from this assignment.

2. Scope of this milestone

This milestone covers code generation for all of C--.

3. Behavior

3.1. General

The executable that implements your compiler should be called **compile**. It will take its input from **stdin** and generate MIPS assembly code to **stdout**. Error messages, if any, will be sent to **stderr**.[1]

3.2. I/O

Your programs will print out values using the following routine:

void println(int val)

[1] You shouldn't have to deal with errors because syntax errors and type errors are handled by the front end code you've been given.

Prints out the integer **val** to **stdout**, followed by a newline, in effect behaving as:

```
void println(int val) {
    printf("%d\n", val);
}
```

To make type checking work, these will be declared as **extern**s in the input programs. For example:

```
extern void println(int x);
void main(void) {
    int x;
    x = 123;
    println(x);
}
```

The examples below show the output generated when the code generated from the input program is executed on SPIM:

Example 1	**Example 2**

Input program:
 	
`extern void println(int x);`
`int main() {`
`    println(34567);`
`}`
	Input program:
	
	`extern void println(int x);`
	`int main() {`
	`  int x;`
	`  x = 12345;`
	`  println(x);`
	`}`

SPIM output: SPIM Version 8.0 of January 8, 2010 Copyright 1990-2010, James R. Larus. All Rights Reserved. See the file README for a full copyright notice. Loaded: /usr/lib/spim/exceptions.s **34567**	***SPIM output:*** SPIM Version 8.0 of January 8, 2010 Copyright 1990-2010, James R. Larus. All Rights Reserved. See the file README for a full copyright notice. Loaded: /usr/lib/spim/exceptions.s **12345**

Note that the input program *uses* `println()` but does not *define* it. This is similar to the way we use library functions like printf(). To make this work, your compiler should generate the following sequence of MIPS instructions for println(). (This is conceptually analogous to linking in the library code for printf() statically).

```
.align 2
.data
_nl: .asciiz "\n"

.align 2
.text
# println: print out an integer followed by a newline
```

```
_println:
   li $v0, 1
   lw $a0, 0($sp)
   syscall
   li $v0, 4
   la $a0, _nl
   syscall
   jr $ra
```

3.3. Makefile

You should submit a Makefile that provides (at least) the following targets:

make clean:
Deletes any object files (*.o) as well as the file 'compile'

make compile:
Compiles all the files from scratch and creates an executable file named 'compile'.

4. Gotchas to Watch Out For

1. Variables and function names in the input program that conflict with MIPS opcodes, e.g., '**b**'. The simplest way to guard against such conflicts is to add an underscore "_" at the front of each identifier in the generated code. (This is the reason the label for the function `println()` in the code shown in Section 3.2 is `_println`.)

   If you do this, however, you should keep in mind that execution still needs to begin at `main` (which then jumps to `_main`). The simplest way to handle this is to have the code generator create a label `main` whose code is a single unconditional jump to `_main`:

   ```
   main:
       j _main
   ```

2. *Large integer constants*: Immediate operands can be at most 16 bits wide. Loading a constant more than 16 bits wide into a register requires two instructions.

5. Running the Generated MIPS Code

I will run the MIPS assembly code generated by your compiler using **spim-stats**, a version of SPIM that provides statistics on different kinds of instructions executed by a program. (**Note:** For this assignment I will only test whether the code generated by your compiler behaves correctly. Instruction execution counts will be used to evaluate the effects of optimizations once we get to that point.)

The source code for **spim-stats** is available here (as a zipped file) and also on the CS Department server **lectura** in the directory **/home/cs553/fall22/spim-stats**; the executable for this version of SPIM is on **lectura** in the file

/home/cs553/fall23/bin/spim

To run SPIM on a file **foo.s** containing MIPS assembly code, use:

spim -file foo.s

To get execution count statistics, use:

spim -keepstats -file foo.s

For example: The following two files are the source and MIPS assembly code for a program to compute and print out the value factorial(7):

fact.c
fact.s

When we run it on SPIM using the command "`spim -keepstats -file fact.s`" the generated output is

```
5040
Stats -- #instructions : 307
         #reads : 84  #writes 69  #branches 31  #other 123
```

The first line is the output from the program; the last two lines are statistics about the executed instructions.

6. Submitting your work

Submit your work in GradeScope in the submission area created for this assignment. You should submit the following files:

- All files needed to build an executable of your compiler; and
- a Makefile that supports the functionality described above.