Assignment 5
Code Generation
(Posted 12/15/2020, Due:12/31/2020)

In programming assignment #5, we will use the parser and the type checker implemented in the previous assignments as a base to conduct real code generation for C-- programs. I would encourage you to continue using your own type checker as the base. However, if your type checker is not yet fully functional, you are welcome to use the package we provided in Ceiba (which will be released by 12/17).

The target machine is the RISC-V architecture (RV64I). Since we have no RISC-V processors to use at this time, we will rely on QEMU (A machine emulator that can emulate RISC-V on PC/x86-based workstations) to verify the correctness of the generated code. The output file from your compiler will be called **output.s** which contains RISC-V assembly code (rather than RISC-V machine code). However, since the input executable for QEMU is an ELF(Executable and Linkable Format or Extensible Linking Format) file, so we need to use some tools to convert **output.s** to an executable file. In this assignment, we have attached an instruction document, called ***how_to*** (in the soon-be-released package), which gives instructions on how to use the provided tools to build the needed ELF files and how to debug them efficiently. In order to reduce your efforts in building your test environment, we also provide a **VirtualBox** image which already have all the tools installed. One sample assembly code (NOT optimized) output for the factorial function is included in the appendix.

Some useful references:
1. The RISC-V Instruction Set Manual
https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf
2. QEMU website and download site
http://wiki.qemu.org/Main_Page

Grading requirements:
We will use qemu-riscv64 to test run your compiler generated code.
In assignment#5, you need to generate and demonstrate correct code for the following C-- features:

1) Assignment statements
2) Arithmetic expressions
3) Control statements: while, if-then-else
4) Parameterless procedure calls
5) **Read** and **Write** I/O calls

More features (as listed below) will be implemented in assignment #6.

6) Short-circuit boolean expressions
7) Variable initializations
8) Procedure and function calls with parameters
9) **For** loops

10) Multiple dimensional arrays
11) Implicit type conversions

PS: For variable initialization, we support only simple constant initializations, such as
Int I=1;
Float a=2.0;

How to handle Read and Write?

Read and **Write** will be translated into external function calls.

For example:

`write("Enter a number\n");`

could be translated as follows:

First, the string "Enter a number\n" will be placed in the pseudo segment such as:

```
.LC0:
.string "Enter a number\n\000"
.align 4
```

Then the generated code will be as follows:

```
lui a5, %hi(.LC0)
addi a0, a5, %lo(.LC0)    # Load address of string to a0,
                             which is used to pass the
                             string label to _write_str
call _write_str             # jump to _write_str

#    a=read();
call _read_int
mv t0,a0     # the read integer will be put in a0.
str t0,-4(fp)

#    b=fread();
call _read_float
fmv.s ft0,fa0  # the read float number will be put in fa0.
str ft0,-8(fp)

#    write(a);  a is an integer variable
lw t1,-4(fp)
mv a0,t1       # a0 is used to pass the value you would like to write.
jal _write_int

#    write(b);  b is a floating point variable.
lw ft0,-8(fp)
fmv.s fa0,ft0      # fa0 is used to pass the value you would like to write out.
jal _write_float
```

Extra Credits:

If your compiler handles the required features correctly (passing all C-- tests posted on the web), your assignment will be awarded 100 points. To earn extra credits, you could implement at least one of the following features:

1) Minimizing register saving/restoring overhead of procedure calls. Although the sample code shown in the appendix already adopts register calling convention that the callee saves callee-save registers, and let the caller saves caller-save registers, there are still lots of room for improvement, for example, if a leaf procedure with no floating data encountered, then no needs to save/restore FP registers.
2) Using register tracking to keep more data items in registers to avoid memory reloads.
3) Implement value numbering which capture CSE's.

Each feature worth 5 points. This opportunity extends to Assignment#6, which means you may claim such credits either in the implementation of assign#5 or assignment#6.

Appendix I Sample output from a C--/RISC-V compiler

```
int n;
int fact()
{
     if (n == 1)
     {
          return n;
     }
     else

     {
          n =n-1;
          return (n*fact());
     }
}

int MAIN()

{

     int result;
     write("Enter a number:");

     n = read();
     n = n+1;
     if (n > 1)

     {
          result = fact();
     }
     else
```

Because of the usage of our specific tools, main() is replaced by MAIN().

```
     {
          result = 1;
     }
     write("The factorial is ");
     write(result);
     write("\n");
}
```

Sample un-optimized code from a C--/RISC-V compiler

```
.data
_g_n: .word 0
.text
.text
_start_fact:
sd ra,0(sp)
sd fp,-8(sp)
add fp,sp,-8
add sp,sp,-16
la ra,_frameSize_fact
lw ra,0(ra)
sub sp,sp,ra
sd t0,8(sp)
sd t1,16(sp)
sd t2,24(sp)
sd t3,32(sp)
sd t4,40(sp)
sd t5,48(sp)
sd t6,56(sp)
sd s2,64(sp)
sd s3,72(sp)
sd s4,80(sp)
sd s5,88(sp)
sd s6,96(sp)
sd s7,104(sp)
sd s8,112(sp)
sd s9,120(sp)
sd s10,128(sp)
sd s11,136(sp)
sd fp,144(sp)
fsw ft0,152(sp)
fsw ft1,156(sp)
fsw ft2,160(sp)
fsw ft3,164(sp)
fsw ft4,168(sp)
fsw ft5,172(sp)
fsw ft6,176(sp)
fsw ft7,180(sp)
la t5, _g_n
lw t0,0(t5)
.data
_CONSTANT_1: .word 1
.align 3
```

```
.text
lw t1, _CONSTANT_1
sub t0, t0, t1
seqz t0, t0
beqz t0, _elseLabel_0
la t5, _g_n
lw t0,0(t5)
mv a0,t0
j _end_fact
j _ifExitLabel_0
_elseLabel_0:
la t5, _g_n
lw t0,0(t5)
.data
_CONSTANT_2: .word 1
.align 3
.text
lw t1, _CONSTANT_2
subw t0, t0, t1
la t1, _g_n
sw t0,0(t1)
la t5, _g_n
lw t0,0(t5)
jal _start_fact
mv t1, a0
mulw t0, t0, t1
mv a0,t0
j _end_fact
_ifExitLabel_0:
_end_fact:
ld t0,8(sp)
ld t1,16(sp)
ld t2,24(sp)
ld t3,32(sp)
ld t4,40(sp)
ld t5,48(sp)
ld t6,56(sp)
ld s2,64(sp)
ld s3,72(sp)
ld s4,80(sp)
ld s5,88(sp)
ld s6,96(sp)
ld s7,104(sp)
ld s8,112(sp)
ld s9,120(sp)
ld s10,128(sp)
ld s11,136(sp)
ld fp,144(sp)
flw ft0,152(sp)
flw ft1,156(sp)
flw ft2,160(sp)
```

```
flw ft3,164(sp)
flw ft4,168(sp)
flw ft5,172(sp)
flw ft6,176(sp)
flw ft7,180(sp)
ld ra,8(fp)
mv sp,fp
add sp,sp,8
ld fp,0(fp)
jr ra
.data
_frameSize_fact: .word 184
.text
_start_MAIN:
sd ra,0(sp)
sd fp,-8(sp)
add fp,sp,-8
add sp,sp,-16
la ra,_frameSize_MAIN
lw ra,0(ra)
sub sp,sp,ra
sd t0,8(sp)
sd t1,16(sp)
sd t2,24(sp)
sd t3,32(sp)
sd t4,40(sp)
sd t5,48(sp)
sd t6,56(sp)
sd s2,64(sp)
sd s3,72(sp)
sd s4,80(sp)
sd s5,88(sp)
sd s6,96(sp)
sd s7,104(sp)
sd s8,112(sp)
sd s9,120(sp)
sd s10,128(sp)
sd s11,136(sp)
sd fp,144(sp)
fsw ft0,152(sp)
fsw ft1,156(sp)
fsw ft2,160(sp)
fsw ft3,164(sp)
fsw ft4,168(sp)
fsw ft5,172(sp)
fsw ft6,176(sp)
fsw ft7,180(sp)
.data
_CONSTANT_3: .ascii "Enter a number:\000"
.align 3
.text
```

```
la t0, _CONSTANT_3
mv a0,t0
jal _write_str
jal _read_int
mv t0, a0
la t1, _g_n
sw t0,0(t1)
la t5, _g_n
lw t0,0(t5)
.data
_CONSTANT_4: .word 1
.align 3
.text
lw t1, _CONSTANT_4
addw t0, t0, t1
la t1, _g_n
sw t0,0(t1)
la t5, _g_n
lw t0,0(t5)
.data
_CONSTANT_6: .word 1
.align 3
.text
lw t1, _CONSTANT_6
sgt t0, t0, t1
beqz t0, _elseLabel_5
jal _start_fact
mv t0, a0
sw t0,-4(fp)
j _ifExitLabel_5
_elseLabel_5:
.data
_CONSTANT_7: .word 1
.align 3
.text
lw t0, _CONSTANT_7
sw t0,-4(fp)
_ifExitLabel_5:
.data
_CONSTANT_8: .ascii "The factorial is \000"
.align 3
.text
la t0, _CONSTANT_8
mv a0,t0
jal _write_str
lw t0,-4(fp)
mv a0,t0
jal _write_int
.data
_CONSTANT_9: .ascii "\n\000"
.align 3
```

```
.text
la t0, _CONSTANT_9
mv a0,t0
jal _write_str
_end_MAIN:
ld t0,8(sp)
ld t1,16(sp)
ld t2,24(sp)
ld t3,32(sp)
ld t4,40(sp)
ld t5,48(sp)
ld t6,56(sp)
ld s2,64(sp)
ld s3,72(sp)
ld s4,80(sp)
ld s5,88(sp)
ld s6,96(sp)
ld s7,104(sp)
ld s8,112(sp)
ld s9,120(sp)
ld s10,128(sp)
ld s11,136(sp)
ld fp,144(sp)
flw ft0,152(sp)
flw ft1,156(sp)
flw ft2,160(sp)
flw ft3,164(sp)
flw ft4,168(sp)
flw ft5,172(sp)
flw ft6,176(sp)
flw ft7,180(sp)
ld ra,8(fp)
mv sp,fp
add sp,sp,8
ld fp,0(fp)
jr ra
.data
_frameSize_MAIN: .word 192
```

Additional Notes:

a) You may assume the identifier names will not exceed 256 characters. However, the number of distinct identifiers should not be limited.

b) In the hw5 directory you may find the following files:

1) src/lexer3.l	the lex program
2) src/header.h	contains AST data structures
3) src/Makefile	
4) src/parser.y	
5) src/functions.c	supporting functions
6) pattern/*.c	test data files

Submission requirements:

1) DO NOT change the executable name (parser).
2) Your compiler should produce the output RISC-V code in a file called “output.s”.
3) Compress all modules needed to generate your compiler. Then upload your packaged file to Ceiba.

3) We grade the assignments on the QEMU installed on Ubuntu 18.04. Before submitting your assignment, you should make sure your version can be compiled by using “make” and works correctly on such an environment.