Project 1: Simple code generation — assigned Wednesday 29 September — due Tuesday 26 October

The goal is to translate programs in a very simple canonical imperative language, known in the literature as WHILE, or sometimes IMP, into RISC-V assembly code. The resulting assembly code will be linked with a main program, written in C and compiled with gcc, and then run on either the SiFive board (risc-machine.cs.unm.edu) or a RISC-V emulator.

1.1 Abstract syntax

The abstract syntax of the language WHILE is:

Arithmetic expressions $a ::= x \mid n \mid a_1\ o_a\ a_2$
Boolean expressions $b ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{not}\ b_1 \mid b_1\ o_b\ b_2 \mid a_1\ o_r\ a_2$
Commands $c ::= x := a \mid \mathsf{skip} \mid c_1; c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} \mid \mathsf{while}\ b\ \mathsf{do}\ c_1\ \mathsf{od}$

where x ranges over (scalar) variable names; n ranges over integer literals; o_a ranges over binary arithmetic operators, namely $+$, $-$, $\times$; `true` and `false` are boolean literals; `not` is a unary boolean operator; o_b ranges over binary boolean operators, namely $\wedge$, $\vee$; o_r ranges over binary relational operators, namely $=$, $<$, $\leq$, $>$, $\geq$.

1.2 Concrete syntax

Variable names are standard programming-language identifiers—letters, digits, underscore, single-quote—and start with a letter.

Integer literals are standard (zero or positive) whole numbers written in decimal, i.e., strings of digits.

Keywords are exceptions from identifiers: `true`, `false`, `not`, `and`, `or`, `skip`, `if`, `then`, `else`, `fi`, `while`, `do`, `od`.

Infix binary operators: `+`, `-`, `*`, `and`, `or`, `=`, `<`, `<=`, `>`, `>=`.

Prefix unary operator: `not`. For simplicity, it requires parentheses round its argument.

Operators have the usual mathematical precedence: `*` has higher precedence than the additive operations `+` and `-`. Operators are left-associative.

Any expression can be put in parentheses.

Assignment is written `:=`. Command sequencing is written `;`.

Comments are as in Haskell, `--` for end-of-line comments and `{-...-}` for longer comments.

1.3 Semantics

The language WHILE is meant to represent the essence of most extant imperative programming languages (without procedures). (A formal, executable, structural operational semantics will be given in class.)

1.4 Example programs

```
{-
Computes the factorial of the number stored in x and leaves the result in output:
-}
y := x;
z := 1;
while y > 1 do
  z := z * y;
  y := y - 1
od;
y := 0;
output := z
```

```
{-
  compute Collatz
  input number is "input"
  output is the number of steps, in "output"
-}
n := input;
steps := 0;
while n > 1 do
  rem := n; --Here we divide n by 2:
  quot := 0;
  while rem > 1 do
    rem := rem - 2;
    quot := quot + 1
  od;
  if rem = 0 then
    n := quot
  else
    n := 3*n+1
  fi;
  steps := steps + 1
od;
output := steps
```

1.5 Code generation

There can be any number of variables in the program. Even though the example programs use variables such as "input" and "output", there is nothing special about these names. Variables contain integer quantities, which we will take to be 64 bits wide.

There is no I/O in our programming language. The program is self-contained and its effect is to produce some final assignment of values to variables, given an initial assignment of values to variables. In practice, this means that our program is like a single procedure of a full-fledged programming language, and we will need a main program to perform input, call our procedure, and then perform output.

The main program will allocate an array with an 8-byte entry for each variable. It will initialize all these entries using values supplied by the user on the command line. It will also print the initial values of all the variables, in alphabetical order. Then, it will pass the array by reference as the only argument to the compiled code.

The naive code generator's code will load / store a variable from that array each time the variable is used / assigned to. To generate the code for evaluating each arithmetic expression (as well as each boolean expression), one possibility is to follow the approach taken in the series of compilers for the language of arithmetic expressions that were shown in class, namely, compiling via an abstract stack machine.

Once the compiled code has returned to the main program, the main program will print the final values of all the variables, in alphabetical order.

1.6 Tasks

Task 1: Scan and parse a source language program to yield an abstract syntax tree (AST). You can use whatever tools are appropriate to your implementation language.

Task 2: Display the AST. For the purposes of the project report, generate a PDF of the AST. (Hint: you can use an external graph-drawing tool, such as graphviz, to render an AST.)

Task 3: Convert the AST into RISC-V assembly code, and then produce a standalone Fedora Linux RISC-V executable by linking with a custom main program in C, which you will also generate.

Task 4: Extensively test your compiler. Write test programs to demonstrate complete coverage of language features.

1.7 What will be provided

An example main program follows. This corresponds to the Collatz program above.

```
#include <stdlib.h>
#include <stdio.h>
extern void example6 (long int * vars);
long int vars [6];
```

```
int main (int argc, char ** argv)
{
  int i;
  if (argc != 7)
    {
      printf ("Usage: example6-main input n output quot rem steps\n");
      exit(1);
    }
  vars [1] = atoi (argv [1]);
  vars [0] = atoi (argv [2]);
  vars [5] = atoi (argv [3]);
  vars [4] = atoi (argv [4]);
  vars [3] = atoi (argv [5]);
  vars [2] = atoi (argv [6]);
  printf ("Initial state:\n");
  printf ("input=%ld\n", vars [1]);
  printf ("n=%ld\n", vars [0]);
  printf ("output=%ld\n", vars [5]);
  printf ("quot=%ld\n", vars [4]);
  printf ("rem=%ld\n", vars [3]);
  printf ("steps=%ld\n", vars [2]);
  example6 (vars);
  printf ("Final state:\n");
  printf ("input=%ld\n", vars [1]);
  printf ("n=%ld\n", vars [0]);
  printf ("output=%ld\n", vars [5]);
  printf ("quot=%ld\n", vars [4]);
  printf ("rem=%ld\n", vars [3]);
  printf ("steps=%ld\n", vars [2]);
}
```

What to turn in

Submit all the code and all the tests in source form and a suitably compiled form (depending on your implementation language). Include a file `project1.pdf` containing all of the above nicely formatted and preceded by a project report. In the project report, **please remember to describe your team composition and how the team collaborated on the task, and comment on the level of effort needed for the project**.

How to turn in

Instructions will be provided later.

Project presentations

Teams will present their projects in class. Presentations will take place before the project is due.