

Project 1: Simple code generation — assigned Tuesday 23 September — final submissions due Tuesday 28 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 work in conjunction with a main program, which is responsible for all input/output. Both the assembly code and the main program will be generated by your compiler. Your compiler can run on any computer you like.

After that, both the assembly code and the main program will be processed by `gcc` on a RISC-V host. and then the resulting executable will be run on the RISC-V host. You will use the department's RISC-V machine `risc-machine-2.cs.unm.edu` for timing runs. (In your development feel free to use the department's machine or an emulator. An emulator is handy because it will run on your laptop. To access `risc-machine-2.cs.unm.edu` via `ssh` you must first `ssh` into any regular CS host. When you use `risc-machine-2.cs.unm.edu` make sure not to leave running or hanging processes behind.)

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 $b ::= \text{true} \mid \text{false} \mid \text{not } b_1 \mid b_1 \ o_b \ b_2 \mid a_1 \ o_r \ a_2$

Commands $c ::= x := a \mid \text{skip} \mid c_1; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi} \mid \text{while } b \text{ do } c_1 \text{ od}$

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 (both lowercase and uppercase), 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. They must be small enough to be representable as 64-bit signed integers.

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

Infix binary operators: $+$, $-$, $*$, `and`, `or`, $=$, $<$, \leq , $>$, \geq .

Prefix unary operator: `not`. It requires square brackets `[]` around its argument.

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

Any arithmetic expression can be put in round parentheses `()` and any boolean expression can be put in square brackets `[]`.

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). The command `skip` does nothing; the rest should be familiar. In this course we will not formalize the semantics (see CS550 for that).

1.4 Example programs

1.4.1 Factorial

```
{-
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
```

1.4.2 Collatz

```
{-
  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 above use variables such as “input” and “output”, there is nothing special about these names. Variables contain integral quantities, which we will take to be signed 64-bit integers.

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 (in C) 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/into 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), you must 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: Formally define the concrete syntax of the language.

Task 2: Write test programs to demonstrate complete coverage of language features. Write some test programs such that the running time of the program depends on the value of an input variable and can be varied over a large range. (Hint: at this point in the project, since you don’t yet have a compiler, it is convenient to develop test programs in C or Python syntax at first, then render them in While syntax.)

Task 3: Scan and parse a source language program to yield an abstract syntax tree (AST). To write the scanner and the parser you can use whatever tools are appropriate to your implementation language. Develop a separate set of test programs with syntax errors. Extensively test your compiler.

Task 4: 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 5: 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. Here, we do not use control flow graphs (that will be the topic of our next project); instead we produce assembly code directly from the AST, as described, for example, in Cooper & Torczon 5.3.3 or in Mogensen 6.3–6.6 (with RISC-V assembly in place of their intermediate languages). Extensively test your compiler.

Task 6: Carry out performance testing, by running the generated executable files on `risc-machine-2.cs.unm.edu`. You will use the test programs you wrote in Task 2 above, as well as test programs shared by the other teams and test programs supplied by the instructor (some of which were shared by past years' students).

Task 7: Write the final project report.

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] = atol (argv [1]);
    vars [0] = atol (argv [2]);
    vars [5] = atol (argv [3]);
    vars [4] = atol (argv [4]);
    vars [3] = atol (argv [5]);
    vars [2] = atol (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]);  
}
```

1.8 What to turn in

For each task submission, submit all the code and all the tests in source form and a suitably compiled form (depending on your implementation language).

For the final project submission, also include a file `project1.pdf` containing all of the above, as appropriate, 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.**

1.9 How to turn in

Package everything into a tar file and upload it on Canvas. Only one team member need submit.

1.10 Project presentations

Teams will present their projects in class. Presentations will take place at least twice; first around the time Task 3 is due and then right before the final project submission is due.