Lab 4
Syntax-Directed Code Generation

Objective

During the previous lab, you have written your own interpreter of the MiniC language. In this lab the objective is to generate *valid* RISCV codes from MiniC programs:

- Generate 3-address code for the MiniC language.
- Generate executable "dummy" RISCV from programs in MiniC via two simple allocation algorithms.
- **Please follow instructions and COMMENT YOUR CODE!**

Student files are in the Git repository.

Make sure your Git repository is up-to-date, using `git pull`.

4.1 Preliminaries

This section must be read **carefully**.

Important remark From now on, we add the following restriction to the MiniC language: Values (variables, argument of `println_int`) are of type (signed) `int` or `bool` only (no float, no string, no char). Thus all values can be stored in regular registers or in one cell (64 bits) in memory. You can let your program crash (`raise MiniCUnsupportedError(...)`) if another type of variable is provided.

Note that real compilers would perform the code generation from a decorated AST (with type annotations attached to nodes). For simplicity, we will work on the non-decorated AST: our language is simple enough to generate code without decorations.

Structure of the compiler's code In the `MiniC/Lib` folder, we provide you with many utility functions. A detailed documentation of the library is given in the repository, and can be accessed at the root of the git repository by opening `docs/index.html` in a web browser such as Firefox.

As for other files in the MiniC directory:

- `CodeGen/MiniCCodeGen3AVisitor.py` is the code generation algorithm, implemented as a visitor.
- The file `TypingAndInterpret/MiniCTypingVisitor.py` is reused from lab3. If your typechecker is buggy, you can use the compiler's `--disable-typecheck` to run the code generation without typechecking, and give the value True to `DISABLE_TYPECHECK` in `test_codegen.py`.
- The main Python file, `MiniCC.py` as in lab3, now accepts new options related to code generation (check `python3 MiniCC.py --help` for a full list). Running

  `python3 MiniCC.py --mode codegen-linear <file>`

  launches the chain: production of 3-address code with temporaries, allocation, replacement, print.
- The script `test_codegen.py` will help you test your code. We will use it in Section 4.4 through Makefile targets.
- The `README-codegen.md` file is to be completed progressively during the lab.

Re-test the command-line version of the RISCV simulator, for example with code from TP03:

```
cd ../TP03/riscv/
riscv64-unknown-elf-gcc libprint.s test_print.s -o test_print.riscv
spike -m100 pk test_print.riscv
cd ../../MiniC/
```

4.1.1 Conventions used in the assembly code

- All data items are stored on 64 bits (double-words, 8 bytes).
- Registers `s1`, `s2`, and `s3` are reserved for temporary computations (e.g. to compute an address before or after an `sd` or a `ld`, or to store a value between a memory access and an arithmetic operation). Note that `s0` is an alias for `fp`, hence `s0` must not be used as a general purpose register either.
- Registers `s4`, ..., `s11`, `t0`, ..., `t6` are general purpose registers, that can be used freely by the code generator. In your Python code, you can access the list of general-purpose registers with `Operands.GP_REGS`. `si` and `ti` registers will behave differently in presence of function calls, but are considered equivalent for now.
- To store properly in memory, it is mandatory to compute offsets from the "reserved" register `fp`. To be compatible with the RISCV ecosystem, we will use a stack **growing with decreasing addresses**. Thus data in the stack is accessed by adding a **negative offset** (multiple of 8) to `fp`. In other words, we use the memory locations `-8(fp)`, `-16(fp)`, ... The `sp` register points to the first data contained in the stack. It is always 16-byte (2 double-words) aligned.
- Registers `a1` to `a7` are not used at all for the moment.

4.1.2 Conventions used in the test suite

A few reminders and new features of the test suite:

- Test files should contain directives giving the expected behavior:
  - `// EXPECTED` and the following lines to give the expected output;
  - `// EXITCODE` *n* gives the expected return code of the compiler, i.e. `// EXITCODE 1` when the code should be rejected by your typechecker (see previous lab for the specification of different exit codes);
  - `// SKIP TEST EXPECTED` to specify that this test should not be run through `test_expect` (see below);
- Several tests are run on each `.c` files when launching `make test-`*something* (`make test-naive`, `make test-lab4`, etc.):
  - `test_expect`, that compiles the file using `riscv64-unknown-elf-gcc`. It checks that `EXPECTED` directives are correct, but doesn't test your compiler.
  - `test_naive_alloc`, `test_alloc_mem`, `test_smart_alloc` that compiles the file using your compiler, using the corresponding register allocation algorithm. The test suite leaves generated `.s` files next to the `.c` source file.
- `make test FILTER=...` can be used to restrict the set of files on which to run the test. Specify either a single file or an extended wildcard like `CodeGen/tests/**/*while*.c`.

4.2 First step: get familiar with the code and test suite

In this section you have to implement the course rules in order to produce RISCV code with temporaries. These rules are given in Figure 4.2 on page 9 and Figure 4.3 on page 10.

Here is an example of the expected output of this part. From the following MiniC program:

```
#include "printlib.h"

int main() {
    int a,n;
    n = 1;
    a = 7;
```

```
    while (n < a) {
      n = n+1;
    }
    println_int(n);
    return 0;
}
```

the following code is supposed to be generated.

```
##Automatically generated RISCV code, MIF08 & CAP
##non executable 3-Address instructions version


##prelude
# [...] Some automatically generated code that will be explained in a future lab

##Generated Code
# [...] Some automatically generated code that will be explained in a future lab
        li temp_0, 0
        li temp_1, 0
        # (stat (assignment n = (expr (atom 1))) ;)
        li temp_2, 1
        mv temp_0, temp_2
        # (stat (assignment a = (expr (atom 7))) ;)
        li temp_3, 7
        mv temp_1, temp_3
        # (stat (while_stat while ( (expr (expr (atom n)) < (expr (atom a))) ) (
    stat_block { (block (stat (assignment n = (expr (expr (atom n)) + (expr (atom 1)))
    ) ;)) }))))
lbl_begin_while_1_main:
        li temp_4, 0
        bge temp_0, temp_1, lbl_end_relational_3_main
        li temp_4, 1
lbl_end_relational_3_main:
        beq temp_4, zero, lbl_end_while_2_main
        # (stat (assignment n = (expr (expr (atom n)) + (expr (atom 1)))) ;)
        li temp_5, 1
        add temp_6, temp_0, temp_5
        mv temp_0, temp_6
        j lbl_begin_while_1_main
lbl_end_while_2_main:
        # (stat (print_stat println_int ( (expr (atom n)) ) ;))
        mv a0, temp_0
        call println_int
# [...] Some automatically generated code that will be explained in a future lab

##postlude
# [...] Some automatically generated code that will be explained in a future lab
```

4.2.1 3-address code generation on supported C file

In the skeleton, we provide you an incomplete `MiniCCodeGen3AVisitor.py`. To run it, type

```
make  # to generate the lexer and parser
python3 MiniCC.py --mode codegen-linear CodeGen/tests/provided/step1/test00.c \
                  --reg-alloc=none
```

```
cat CodeGen/tests/provided/step1/test00.s
```

Observe the generated code, it is complete (the skeleton supports all features used in this input file). Since we used `--reg-alloc=none`, register allocation wasn't performed, so we still get temporaries in the output, which is not executable. We generated RISCV comments with MiniC statements to help debugging.

To get executable code, we provide you the naive (everything in registers) allocation:

```
python3 MiniCC.py --mode codegen-linear CodeGen/tests/provided/step1/test00.c \
                  --reg-alloc=naive --output CodeGen/tests/provided/step1/test00-naive.s
cat CodeGen/tests/provided/step1/test00-naive.s
riscv64-unknown-elf-gcc CodeGen/tests/provided/step1/test00-naive.s libprint.s \
                  -o CodeGen/tests/provided/step1/test00-naive.riscv
spike pk CodeGen/tests/provided/step1/test00-naive.riscv
```

This should ouptut 42 (plus the usual `bbl loader` line).

You can automate this, plus the run of `test_expect` (see above) with:

```
make FILTER=CodeGen/tests/provided/step1/test00.c test-naive
```

4.2.2 3-address code generation on C file not supported by the skeleton

Now, run the same command on an input file not yet supported by the skeleton:

```
python3 MiniCC.py --mode codegen-linear CodeGen/tests/provided/step1/test00b.c \
                  --reg-alloc=none
```

You should get a `NotImplementedError` exception, with a backtrace pointing to the location in the generator where the feature (here, additive expressions) need to be implemented. To get an idea of what you'll need to implement, you may run:

```
git grep NotImplementedError
```

Here also, you can automate the test with:

```
make FILTER=CodeGen/tests/provided/step1/test00b.c test-naive
```

Off course, you can also run the whole test suite without `FILTER`:

```
make test-naive
```

A coverage report of the test suite is generated in `htmlcov`, it can help you identify untested parts of your code.

It is strongly advised to adopt a "test driven" methodology, i.e. run the test suite, see what fails, fix it, and iterate. When implementing a feature for which the skeleton has no test, write a test first.

The rest of the lab boils down to "make sure all tests pass and the test suite covers 100% of the generator and allocators".

4.3 3 address code generation

EXERCISE #1 ▶ **Basic cases for 3 address code generation**
Implement the 3-address code generation corresponding to test cases in `CodeGen/tests/provided/step1/`, i.e. make the following command succeeds:

```
make FILTER="CodeGen/tests/provided/step1/*" test-naive
```

EXERCISE #2 ▶ **A few corner-cases**
Some points may require extra care, in the implementation or in the tests:

- Don't forget the explicit errors for division by zero. We provide you a piece of assembly code raising the error (see `print_code()` given in the library of the skeleton), you need to generate the instruction to jump to this label (we get it with `self.get_label_div_by_zero()`) when the right operand of a division or modulo is 0.
- `float` and `string` are unsupported. The compiler raises `MiniCUnsupportedError` when encountering any of them. Tests are provided for this.

Note that testing the division by 0 requires a bit of attention. We need to check that the executable exits with code 1 at runtime, that the output is correct, but we can't check that GCC gives the same behavior because GCC doesn't give a clean error message. A test case may therefore be:

```
#include "printlib.h"

int main(){
        println_int(1 / 0);
        return 0;
}
// SKIP TEST EXPECTED
// EXECCODE 1
// EXPECTED
// Division by 0
```

Test and implement all these features.

EXERCISE #3 ▶ **End of 3-address code generation for MiniC**
Implement the 3-address code generation rules:
- for boolean expressions and numerical comparison: compute 1 (true) or 0 (false) in the destination register; be careful the `not` boolean instruction is not as easy as you could wish;
- while loops;
- if then else.

At this point all the tests should be ok for all files in directory `CodeGen/tests/provided/step2/`. However these tests are not sufficient, you should add some other ones (in the directory `CodeGen/tests/students/`). Run the test suite with `make test-naive MODE=codegen-linear` to use all the test files.

About `if` and `while` For tests (and boolean expressions), make sure you generate "conditional jumps" with:

```
self.add_instruction(
          RiscV.conditional_jump(label, op1, cond, op2))
```

where `op1` (resp `op2`) is the left operand (resp right operand or the numerical constant 0, nothing else), i.e. a register or a value of the boolean condition `cond` (`Condition('beq')` for equality, for instance) [1], and `label` is a label to jump to if the condition evaluates to true.

4.4 More on the naive allocator

We provide you with an allocation method which allocates temporaries in registers as long as possible, and fails if there is no more available registers. The process takes as input the former 3-address code and transforms each instruction according to the allocation function. When there are not enough registers available, the allocator raises an `AllocationError` exception. The test suite is programmed to skip tests raising this exception (i.e. the test is neither considered as a success nor as a failure).

Open, read, understand the `NaiveAllocator` implementation in `Lib/Allocator.py` and how it is used to perform the actual RISCV code generation.

[1] We suggest to use `git grep` and find this class definition and this method somewhere in the library we provide.

4.5 RISCV code with "all-in-mem" allocation of temporaries

Tests Up to now, you used `make test-naive MODE=codegen-linear` to test your code, and at this point all tests should pass, or be skipped (do not forget to make a test where the naive allocation uses too many registers!). From now on, you should use the more complete `make test-mem MODE=codegen-linear` command, that tests everything with the provided naive allocator, and the all-in-memory allocator you have to write. If you use `MiniCC.py` directly, the corresponding option is `--reg-alloc=all-in-mem`.

Check that `make test-mem MODE=codegen-linear` does fail. You can also run `make test-lab4` to run the tests for all allocators in this lab.

Implementation As the number of registers for allocation is bounded by the number of available general purpose registers, i.e. `len(Operands.GP_REGS)`, the naive allocator cannot deal with more temporaries than general-purpose registers: we have to find a way to store the results elsewhere. In this particular lab, we will use the following solution:

- The generated code will use memory locations in the stack.
- All values that are propagated from one rule to another (sub-expressions, ...) must be stored in the stack, whose address will be stored in FP.
- $s1, s2, s3$ will be used to compute the value to store or as a destination register for the value(s) to read. Technically, only 2 of these registers are mandatory, but you should be cautious if you try a 2-registers-only solution.
- In order to know if a given (temporary) operand should be read and/or written, use the `is_read_only` method of the `Instruction` class.

Figure 4.1 depicts the stack implementation for the RISCV machine, that follows the RISC-V calling convention (stack growing downwards, stack-pointer always 16-bytes aligned).

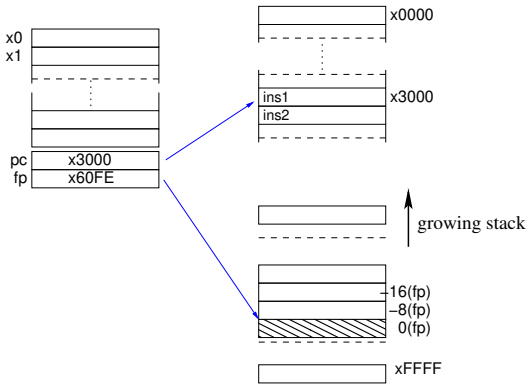

Figure 4.1: Memory model for RISCV

Following the convention that `fp` always stores the "begining of stack address", pushing the content of register $s3$ in the stack will be done following the steps:

- compute a new offset (call to the `fresh_offset` method).
- generate the following instruction:

```
sd s3, -offset*8(fp)
# sd = store double = 64-bits store
# -offset*8(fp) = memory location at address fp-offset*8
```

Getting back the value is similar.

To understand the principle, complete manually the expected output for the following two statements. The temporary `temp_3` is located at -32(fp) and `temp_4` is located at -40(fp):

```
int x, y;
x=4;
y=12+x
```

Listing 4.1: 'all in mem alloc for test_while2b.c'

```
##Generated code without prelude and postlude
        # (stat (assignment x = (expr (atom 4))) ;)
        # li temp_2, 4
        li s2, 4
        sd s2, -24(fp)
        # end li temp_2, 4
        # mv temp_1, temp_2
        ld s1, -24(fp)
        mv s2, s1
        sd s2, -16(fp)
        # end mv temp_1, temp_2
        # (stat (assignment y = (expr (expr (atom 12)) + (expr (atom x)))) ;)
        # li temp_3, 12
        # TODO 2 lines


        # end li temp_3, 12
        # add temp_4, temp_3, temp_1
        # TODO 4 lines




        # end add temp_4, temp_3, temp_1
        # mv temp_0, temp_4
        # NOT TODO
```

EXERCISE #4 ▶ **Implement**

Now you are on your own to implement this code generation. The relevant file is `CodeGen/AllInMemAllocator.py`. Here are the main steps (less than 50 locs of PYTHON):

1. We have implemented for you an `AllInMemAllocator.prepare()` method. It only maps each temporary to a new offset in memory (in a PYTHON `dict`), allowing the use of the method `get_alloced_loc()` on a temporary used in the code.

2. Complete the method `AllInMemAllocator.replace(old_instr)` taking as input a "3-address with temporaries" RISCV code and outputs a list of instructions as a replacement. For instance, each time we access a source operand, we have to load it from memory before, thus the `replace` should contain something like

   ```
   # regxxx is the register used to hold the value between the load and
   # the operation itself (one of s1, s2, s3).
   # loc is the place in memory where the temporary is allocated (of
   # the form Offset(..., fp), obtained with get_alloced_loc().
   before.append(RiscV.ld(regxxx, loc))
   ```

   Be careful to not add useless `ld` or `sd` instructions!

The files you generate have to be tested with the RISCV simulator with the same script as before. **Of course, with "all-in-mem" allocation, tests that were "skipped" due to the lack of registers with the naive allocation should not be skipped for `test_alloc_mem`.**

More tests Now that your compiler can deal with a large number of temporaries, make sure all features are heavily tested (the test suite we provide is in no way sufficient).

4.6 Extensions

You may need to write tests that are accepted by your compiler but not by GCC. If you do so, add a `// SKIP TEST EXPECTED` directive in your tests, to disable the `test_expect` that would otherwise check your file using GCC.

EXERCISE #5 ► **C- or Fortran-like for loops code generation**
If you implemented one of the extensions in Lab 3, you can add it to code generation.

Note that the semantics of fortran-like loops when the loop counter is assigned within the loop makes the code generation harder than C-like loops, where the loop counter is a variable like any other.

4.7 Delivery

This lab will be graded, but we will only ask you to upload it along with its second part (Lab4b), which takes place next week. We highly recommend you to finish this part at least up to the all-in-mem allocator before, in particular all tests from `make test-lab4 MODE=codegen-linear` (including your own) should pass.

c	`dest <- fresh_tmp()`
	`code.add("li dest, c")`
	`return dest`

x	`# get the temporary associated to x.`
	`tmp <- symbol_table[x]`
	`return tmp`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dest <- fresh_tmp()`
	`code.add("add dest, t1, t2")`
	`return dest`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dest <- fresh_tmp()`
	`code.add("sub dest, t1, t2")`
	`return dest`

true	`dest <-fresh_tmp()`
	`code.add("li dest, 1")`
	`return dest`

$e_1 < e_2$	`dest <- fresh_tmp()`
	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`endrel <- fresh_label()`
	`code.add("li dest, 0")`
	`# if t1>=t2 jump to endrel`
	`code.add("bge endrel, t1, t2")`
	`code.add("li dest, 1")`
	`code.addLabel(endrel)`
	`return dest`

Figure 4.2: 3@ Code generation for numerical or Boolean expressions

x = e	`tmp <- GenCodeExpr(e)`
	`loc <- symbol_table[x]`
	`code.add("mv loc, tmp")`

S1; S2	`# Just concatenate codes`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

`if` *b* `then` *S*1 `else` *S*2	`lelse <- fresh_label()`
	`lendif <- fresh_label()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add("beq lelse, t1, 0")`
	`GenCodeSmt(S1) # then`
	`code.add("j lendif")`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) # else`
	`code.addLabel(lendif)`

`while` *b* `do` *S* `done`	`ltest <- fresh_label()`
	`lendwhile <- fresh_label()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`code.add("beq lendwhile, t1, 0")`
	`GenCodeSmt(S) # execute S`
	`code.add("j ltest") # and jump to the test`
	`code.addLabel(lendwhile) # else it is done.`

Figure 4.3: 3@ Code generation for Statements