Lab 6
Code generation and typechecking for functions

Objective

- Add functions to MiniC.
- Understand and implement typing rules for functions.
- Understand and implement memory layout (stack) for functions.
- Your work is due **on `https://etudes.ens-lyon.fr` (NO EMAIL PLEASE)**, before **2021-11-29 23:59**. More instructions in section 6.5.

This lab lasts 2 sessions.

Getting started At this point, you should have a compiler with operational typechecking and code generation (code from labs 4 and 5), except for functions.

If you didn't re-plug the typechecker of your lab3 in your compiler, you should do it now. We won't recheck details of the function body's typechecking, but your typechecker should be working at least on basic examples.

The lab contains two parts: typechecking, and code generation. They are independent, so you can work in any order.

Run the command `git pull` (you may need to run `git commit` first) to get new test files for functions (in `TP06/`).

6.1 Specifications and limitations

We implement a subset of C-like functions:

- Functions always have a return type, and, unlike C, function calls cannot appear as a statement (in other words, `x = f();` is accepted as a statement, but not `f();` alone);
- Function calls can appear anywhere in an expression, i.e. `x = f(g(x) + h())` is valid;
- Functions can have 0 to 8 arguments, but not more;
- MiniC allows "forward declarations", like C, such as

  ```
  int f(int x, bool y);
  ```

  These declarations produce no code, but allow a function to be called before it is actually defined (i.e. given a body), and external function calls. A function call is valid only if the function is either defined or declared above. There can be any number of forward declaration for the same function, but at most one definition. When several declarations or definitions are made for the same functions, they must have the same arguments and return types.
- Arguments and return type may be `int` or `bool`.

6.2 Testing

A few reminders and new features of the testsuite:

- When we evaluate your testsuite, we consider that any `*.c` file whose name starts with a letter is a testcase. You may need to write other C files: in this case use a filename like `_foo.c`

- 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;
  - `// EXECCODE` *n* gives the expected return code of the generated executable, i.e. the return value from the `main` function;
  - `// SKIP TEST EXPECTED` to specify that this test should not be ran through `test_expect`;
  - `// LINKARGS` *string* to provide arguments that should be used by the linker (`riscv64-unknown-elf-gcc`) when assembling and linking the generated assembly code. *string* may contain the special string `$dir`, which will be replaced with the directory containing the testcase. A typical example is to link with an external C library, using `// LINKARGS $dir/lib/_hello.c` (note the use of `_` in the filename to mark that `_hello.c` is not a test case).
- Several tests can be ran on each `.c` files:
  - `test_expect`, that compiles the file using `riscv64-unknown-elf-gcc`.
  - `test_naive_alloc`, `test_alloc_mem`, `test_smart_alloc` that compiles the file using your compiler, using the corresponding register allocation algorithm.
  - to lauch all these tests (and pyright) you can use the command

    ```
    make tests-codegen SSA=1 TEST_FILES=path/to/testfile.c
    ```

    if you do not give the option `TEST_FILES`, the default value from `test_codegen.py` will be used.
- We provide a few tests in `TP06/tests/provided/basic-functions/`. As the name suggests, these are only basic tests, in particular no test is provided for many error cases. You need to add them to the variable `ALL_FILES` in `test_codegen.py`.
- Your own tests must be added in `TP06/tests/students/` as usual. Any test outside this directory will be ignored while grading your work.
- Remember that you can check how much of your code is covered by your tests by opening `htmlcov/index.html` in your web browser.

6.3 Up to you: implement the front-end: Lex, Parse, Type

Functions do not require any new tokens in MiniC, so there is no modification required to the lexing part of the grammar.

The parser needs to be modified in several ways to properly deal with functions: the function declaration rule which is provided is incomplete, and function calls should be implemented.

To test your typer on a program, run `python3 MiniCC.py --typecheck-only <program.c>`. If it does not print anything, it means it has not detected any error.

EXERCISE #1 ► **Parse function definitions**
Modify the `function` rule of `MiniC.g4` to allow functions to take an arbitrary number of parameters, and allow functions to return an arbitrary expression.

To simplify the lab, we still hardcode the `return` *`expression`*`;` statement at the end of functions. It is not possible to use `return` anywhere else.

Make sure your compiler accepts `TP06/tests/provided/basic-functions/test_fun_ret*.c` and `TP06/tests/provided/basic-functions/test_fun_def*.c` programs.

EXERCISE #2 ► **Parse (empty/forward) function declarations**
Add an alternative in the `function` rule of `MiniC.g4` to accept forward declarations declarations. Make sure your compiler accepts `TP06/tests/provided/basic-functions/test_fun_decl*.c` programs.

EXERCISE #3 ► **Parse function calls**
Function calls are expressions in MiniC, which means that expressions such that `f(x)+g(y+1)` should be accepted.

Add an alternative to the `expr` rule to accept function calls. Make sure your compiler accepts `TP06/tests/provided/basic-functions/test_fun_call*.c` programs.

EXERCISE #4 ► **Type**

Implement the type checker for functions. Your implementation should respect the specifications described in Section 6.1. The type checker should check the body of each function in an empty variable environment (there is no global variable), but you should maintain an environment for function signatures.

- Function declaration (with empty body).
- Function definition: your type checker should check that two arguments cannot have the same name, and in MiniC we forbid that a (local) variable has the same name as an argument.
- Function call and return: a function cannot be called before its declaration, and in all calls the type of the arguments, the number of arguments, and the return type must match (not necessarily the name).

Consult the tests for the expected wording for the errors. Be careful: sometimes a wrong return type should raise a type mismatch error in the assignment $x = f(3)$;!

6.4 Code Generation

Some advice:

- The course slides also contains useful information !
- Getting the code to "mostly work" can seem easy, but debugging issues in the generated code is often tricky. Start small, and test properly each feature before you move to the next one. As much as possible, write tests before you write the Python code.
- Your generated code should be compatible with GCC's generated code. Your functions should be able to call GCC's functions, and vice-versa. To test this, you need to use `// LINKARGS $dir/lib/`*file*`.c`. An example is given in `TP06/tests/provided/basic-functions/test_extern.c` (it tests function call from your code to GCC's code, but the other way around should be tested too).
- Register-saving and restoring code is hard to test: you should write functions that use all registers to make sure any improperly saved register is detected. This can be done by calling external functions written manually in assembly (using `LINKARGS`, see example in `TP06/tests/provided/basic-functions/test_extern_asm.c`), and/or by calling functions whose code puts a lot of pressure on registers.
- Code coverage is not a good way to evaluate the quality of your testsuite in this lab: the Python code is easy to cover, but corner-cases (e.g. a register improperly saved and restored) may happen regardless of which Python code is covered.

EXERCISE #5 ► **Code generation for functions**
Implement code generation for functions definition and call. The skeleton provided already generates part of the code needed for function declaration, to set up and restore `fp` and `sp` (see `print_code` in `CFG.py` ; this may also be a good place to increase the size of the stack to take into account callee/caller-saved registers).

A check-list of things to be implemented (it is advised but not mandatory to do it in this order):

- Proper `return` *`expr`* statement (can be tested without function calls using the return value of the `main` function, using `// EXECCODE` in your test file);
- Registers s_i (callee-saved) saving and restoring at the beginning and end of function bodies;
- Implementation of function calls, using `call` *`function`*;
- Getting the result from a function after the `call` *`function`* instruction: read `a0` to a temporary;

- Registers $\mathtt{t}_i$ (caller-saved) saving and restoring before and after function calls;
- Passing arguments: generate code that evaluates actual parameters values to temporaries, and then code that writes their value to $\mathtt{a}_i$ registers;
- Reading arguments within a function by reading their value from $\mathtt{a}_i$ registers to temporaries at the beginning of function bodies.

EXERCISE #6 ► **Possible extensions (no bonus point)**
This exercise is given just for completeness, but no bonus points will be given if you implement it.

You may implement the following:

- Functions with more than 8 arguments. The 9^{th} argument and following are passed on the top of the stack.
- Save and restore only registers that are actually used. If a function does not use an `si` register, it doesn't need to be saved and restored. If a `ti` register isn't live when a function is called, this function call doesn't need to save and restore it.
- Use `ai` registers as general purpose registers within functions, but keep them special at function call time. Slides given in the course explain how to do this (use a temporary per `ai` register, make sure all these temporaries are in conflict, perform graph coloring, and then hardcode the mapping of these temporaries to the correct register).

6.5 Final delivery

We recall that your work is **personal** and code copy is **strictly forbidden**.

EXERCISE #7 ► **Archive**
You MiniC is due on the course's webpage

`https://etudes.ens-lyon.fr/course/view.php?id=4814`

Python code and C testcases will be graded. **Late deliveries will get a heavy penalty, and deliveries more than one hour late will not be accepted.**

Type `make tar` to obtain the archive to send (change your name in the Makefile before!). Your archive must also contain tests (TESTS!) and a (minimal) `README-functions.md` with your name, the functionality of the code, your design choices if any, known bugs, and a checklist.