Lab 6
Part B – Code generation for functions

Objective

- Add functions to MiniC.
- Understand and implement memory layout (stack) for functions.

Getting started At this point, you should have a compiler with operational typechecking for functions.

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

6.1 Test your typechecker

Make sure that `make test-typecheck` works on various tests that include functions.

6.2 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_a`s 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 #1 ▶ **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 `Lib/FunctionData.py`; this is also where the size of the stack is already increased to take into account callee/caller-saved registers, with `fo += len(S[1:]) + len(T)`.

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 $\texttt{t}_i$ (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 $\texttt{a}_i$ registers;
- Reading arguments within a function by reading their value from $\texttt{a}_i$ registers to temporaries at the beginning of function bodies.

EXERCISE #2 ► **Possible extensions (no bonus point)**
This exercise is given just for completeness.

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).