Lab 6
Part A – Parsing and typechecking functions

Objective

- Add functions to MiniC.
- Understand and implement typing rules for functions.

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 lab 3 in your compiler, you should do it now. We won't re-check details of the function body's typechecking, but your typechecker should be working at least on basic examples.

The lab 6 contains two parts: typechecking (part A), and code generation (part B).

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 are made for some function, they must have the same arguments types and return type (but not necessarily the same names for arguments).
- Arguments types 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 test-case. 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 when assembling and linking the generated assembly code (command `riscv64-unknown-elf-gcc`). *string* may contain the special string `$dir`, which will be replaced with the directory containing the testcase. It can pass `.s` or `.c` files as arguments. In this last case the file will be compiled using GCC before it is linked with the testcase.

    An 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 launch all these tests (and pyright) you can use the command

    ```
    # if your SSA is working:
    make test-codegen MODE=codegen-ssa FILTER=path/to/testfile.c
    # otherwise:
    make test-lab4 MODE=codegen-cfg FILTER=path/to/testfile.c
    ```

    if you do not give the option `FILTER`, the default value from `test_codegen.py` will be used.
- Your own tests must be added in `TP06/tests/students/` as usual.
- Remember that you can check your code coverage by opening `htmlcov/index.html` in your web browser.

6.3 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 grammar on a program, run `python3 MiniCC.py --mode=parse <program.c>`. If it does not print anything, it means it has not detected any error.

To run your parser on the testsuite, run `make test-parse`.

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 forward function declarations**
Add an alternative in the `function` rule of `MiniC.g4` to accept forward function 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 definition and all its declarations, and in all calls the type of the arguments, the number of arguments, and the return type must match (not necessarily the name).

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

To run your typechecker on the testsuite, run `make test-typecheck`.

Consult the provided tests for a description of the errors expected wordings.