Compiler Construction
Assignment 1: Frontend

Deadline: 23:59 Monday 17th January 2022

The assignments for this course are based on a small C-like language called FenneC, for which you will be implementing a number of compiler features. We supply you with a framework of a compiler for this language. Before getting started with this assignment, you will need to download the framework code and install its dependencies. See the framework README for instructions on how to set this up (hint: do this first so that you can keep reading while LLVM compiles, this can take a long time). Also familiarize yourself with FenneC by reading the FenneC Language Reference (available on Canvas).

The framework contains an almost-complete compiler frontend in the `frontend/` directory, which is where you will be working for this assignment. `frontend/README.md` describes in detail how the different compiler phases work and where their code is located (read this too before continuing).

As said, the frontend is *almost* complete: it implements everything except floats and loops. Your task for this assignment is to implement these features in all the different steps of the frontend.

Also, the provided test suite is not complete and verifies for a *tentative* grade of roughly 5 - 5.5. You should add your own tests for missing features and corner-cases. Note: the tentative grade assumes that the implementation is actually correct (we always check your code when grading and verify that you're not "cheating" the tests) and is subject to manual plagiarism checks.

1 Floating point operations (3.0 pts)

This part of the assignment is meant to get you warmed up with the framework code by making some simple additions. The `float` type of FenneC is completely missing from the supplied code. You can draw inspiration from the existing `int` type to add support for floating point operations in all the relevant locations:

1. Type and node definitions in `ast.py`:
   Add a `FloatConst` node (look at the existing definitions for inspiration) and add `float` as a basic type.
2. Token definitions in `lexer.py`:
   Add a `FLOATCONST` token that supports C-like floating point constants (e.g., `0.5`, `.0`, or `1.`).
3. Production rules in `parser.py`:
   Add rules to generate the new `FloatConst` AST node from a `FLOATCONST` token.
4. Type checking errors in `typecheck.py`:
   Add support for operations like addition and multiplication on floats.
5. IR generation of floating point operations in `irgen.py`:
   Add `float` as a double-precision LLVM type. Also, generate floating point instructions described in the LLVM Language Reference[1] where applicable. Pay special attention to how comparisons with NaN values are handled in FenneC.

[1] `https://releases.llvm.org/10.0.0/docs/LangRef.html`

For most files, adding floating point support should not require more than adding a few lines of code. For inspiration, try to follow how an `INTCONST` token is processed throughout the different compilation phases.

For the lexer, you will need to write simple regular expressions. Python uses Perl-style regular expressions which you can brush up on at `https://www.regular-expressions.info/quickstart.html` (there is also no shortage of tutorials on the internet).

2 Basic loops (5.0 pts)

Now that you have had some practice with floating point operations, you should be able to do the same for while-loops, do-while-loops and for-loops, all of which are described in the FenneC Language Reference. You will need to:

1. Add the necessary reserved keywords as lexer tokens.
2. Extend the parser according to the FenneC grammar specification.
3. *Desugar* for-loops to while-loops:
   Desugaring is a phase before context analysis, and is explained in the frontend README. It expands syntactic shorthands to equivalent but longer counterparts so that the other phases (context analysis, type checking, IR generation) do not need to deal with syntactic sugar.

   For-loops are expanded to while-loops where the induction variable is defined as a regular variable, and explicitly incremented by one during each loop iteration. For-loops in FenneC are very simple and constrained, so the transformation is straight-forward. The transformation must guarantee that the start and stop expressions are both evaluated exactly once. It must also make sure that the induction variable name does not clash with other variables in the scope.
4. Enforce any type restrictions described in the language reference.
5. Generate appropriate basic blocks and branches during IR generation.

The framework has a built-in testsuite that tests if the basic grammar and functionality of loops works as expected. You can run these by executing `make check` (see the frontend README for more details). The supplied tests do not cover all possible edge cases (whereas the set of tests we use for grading *does*), so you are encouraged to add your own tests as well and see if your code is generic enough to support arbitrary FenneC code constructs.

3 `break`/`continue` (2.0 pts)

Loop control flow statements `break` and `continue` are not included in the basic loops, but are a separate part of the assignment. You will need to follow the same steps as above, and add type checks to see if the statements are indeed used within a loop. `continue` is slightly more difficult because it has different behaviour for while-loops than for for-loops. Think about why this is a problem and how to solve it. Note that these features are only covered very sparsely in the provided testsuite, so you will have to add your own tests to catch the difficult edge cases.