Hello_Compiler Report
Michael Santacroce

Building a compiler has been an arduous journey that both took away some of the magic of programming while also introducing new magic that, for me, is best left untouched. I am not one for names, so my compiler is simply named Hello_Compiler and is available at https://github.com/santacml/Hello_Compiler. Compiler backend filenames all follow "hc_NAME.py" and consist of scanner, parser, typeCheck, and irGenerator, and the main frontend driver is hello_compiler. All of these files are written in Python. There is also a runtime folder consisting of a header and C file for runtime code functions. Python was used as it is the best programming language :).

The first two stages, scanning and parsing, are unique in that the scanner uses finite-state automata and the parser is an LR parser. Both decisions were made because they seemed more adaptable to other projects if desired than the alternatives, the alternatives being a chain of if-statements for scanning and an LL parser. For instance, the scanner has already been adapted for use in other projects twice.

The scanner consists of a root class, called a StateMachine, from which different token classes inherit. They each define a dictionary of next states to be in depending on an input token. This code became difficult to incorporate comments into as nested comments require some counter to keep track of the current nesting level. However, the classes were designed to only be instantiated once and therefore not keep track of any other state. Solving this required redesigning how characters were accepted and custom code for comments.

The LR parser consists quite simply of a table of all patterns and what token they match to. Here a Pattern class is defined that is extensively used throughout the project. A shift table is also defined to resolve shift/reduce conflicts by simply checking forcing the parser to shift when a set of tokens is found in the shift table.

Type checking solely consists of a giant if-statement that checks whether the resultant types of patterns are compatible. Here the symbol table is also introduced. The main purpose of the symbol table is to keep track of the "irHandle" of any variable so that the irGenerator is able to refer to them. It also keeps track of scope by having a list of dictionaries for each time a new scope is introduced.

The irGenerator is where the magic happens (IR being intermediate representation), and was largely and unexpectedly one of the harder parts of the compiler. The class functions through a library called LLVMLite, which is a very lightweight and sometimes poorly-documented set of bindings for LLVM (it seems to be used by numba and nobody else really, I am considering contributing a few things to it). There are seemingly infinite possible scenarios to cover, so the code got pretty gross and long. It takes the same structure as the type checker, a giant if-statement that switches based on an input token, while also having functions for conditionals and loops. Both conditionals and loops were accomplished by instantiating basic blocks in LLVM. Incorporating arrays into the generator took nearly as long as everything else combined, as they require a lot of additional thought and code to accommodate (especially for the complex operations such as adding one int to an entire array).

Finally, runtime was achieved by a simple C file of a few functions that do not do much (bools sanitize from 1 to true and 0 to false) except handle i/o. The library was easy to bind through LLVMLite.

The entire program is run by hello_compiler, which instantiates all classes and runs the main loop. The scanner is written to yield a stream of tokens, and the parser takes that in so that it may yield a string of Patterns. These Patterns are then type-checked and fed to the irGenerator. At the end, the LLVM is automatically called to be binded and run.

To run the compiler, go through the following steps:

1. Compile the runtime library for your system (just in case):
    a. Navigate to the “runtime” folder
    b. gcc -c -Wall -Werror -fpic runtime.c
    c. gcc -shared -o runtimelib.so runtime.o
2. Install llvmlite
    a. With pip: pip install llvmlite
    b. With conda: conda install llvmlite
3. Call the compiler
    a. With a file: python hello_compiler.py FILE_NAME
    b. If no FILE_NAME is provided, the compiler defaults to “test.src” (because I am lazy)