Sprint 4

Project Overview

- Input: Python
- Output: C
- Supports most primitive types & operators, function definitions

Notable features:

- The compiled C code also compiles to executable and the behavior should resemble how python run the input code
- Supports function overloading
- Built-in input and print functions
- Translates list and tuple to C

Optimization:

- constant folding
- constant propagation
- dead code elimination

File Structure

There are many files in the directory. Hopefully these tables can give a better understanding of the files.

Compiler-Related Files

`lex.py`	Our lexer using PLY
`yacc.py`	Our parser using PLY
`AST.py`	Definition of all classes for the AST
`symbol_table.py`	Symbol table implementation, which is being used in both type-checking and IR-to-target steps
`type_checker.py`	Performs type-checking on the AST
`ir_gen.py`	Generates IR from AST
`C_AST.py`	A simpler AST definition that's closer to C. Also contains the code

	to translate it to actual C code
`C_AST_gen.py`	Translate IR to C_AST
`starter.c`	C helper functions. This is being `#include`d from all target code
`ply/`	The PLY library

Testing-Related Files

See Testing section for more details on testing.

`test.py` & `tests/`	Prepared test files that demonstrate all features of our compiler
`test_c_gen.py` & `tests_c_gen/`	Prior test cases that ensure input code compiles through our compiler and also compiles through gcc
`test_ir_gen.py` & `tests_ir_gen/`	Prior test cases that ensure input code is compiled to expected IR
`test_yacc.py` & `tests_yacc/`	Prior test cases that ensure input code is parsed to expected AST
`test_C_AST.py`	Prior test cases that ensure C_AST is translated to expected C code

Caveats & Notes

1. **Indent.** The indent has to be a tab. Indentation using spaces is not supported. In fact, if you accidentally use spaces, the compiler will reject it in the parser step.
2. **Comments.** Comments are not designed to be part of the language. We retro-fitted it afterwards due to limitations on the time available. Any line that has `#` as the first character is removed before handing it over to the actual compiler.
3. **Type annotation for list and tuple.** These are different from Python's official type annotation format. We use `[inner_type]` and `(inner_type)` for list and tuple respectively.
4. **New line is required at the end of a file.** The compiler will fail to compile if there is no new line at the end of a file and a parsing error will be thrown.
5. **String and String concatenation only works when the optimization flag is on.**

Type Extension

- List, tuple
- String

All of the extended types require memory allocation in C. We defined some helper functions in the `starter.c` file to handle memory allocation and deallocation.

List and Tuple

Our compiler uses a `list` struct data structure to store both list and tuple.The type of the list is defined using the `union` type in C, which includes all supported types of our compiler. Values in a collection must be in the same type and match the types in list and tuple declaration, e.g. `x: [int] = [1]`. Type mismatching errors are handled in type checker. List supports element value reassignment and value appending, whereas tuple does not allow these. List and tuple are treated the same at the backend after type checking.
Our compiler also supports `indexing` and `slicing` of list and tuple. We created new AST nodes to specify these two extensions. The type checker enforces `integer` type only for the indices in `indexing` and `slicing`. In `starter.c`, we implemented `list_get(type, list, index)` and `list_slice(list, start, end)` helper. Parameter `type` is needed for getting the value from the `union` type structure. The value at corresponding index is returned. In `list_slice`, a new list is created and values are initialized given the `start` and `end` index, where `end` index is exclusive.
Indexes out of bound and invalid `start` and `end` pairs can be detected by our compiler and the corresponding `Runtime Error` is thrown.

String

We created `String` instances in IR and C_AST to distinguish string type values and values that have been stored in string format. The quotation marks do not appear correctly for string when writing it to a file. We used the `json.dumps` to reproduce the quotation marks. Also, in order to avoid reproducing a string multiple times in optimization, we use `converted_str_lst` to store all the string variable names that have been reproduced. String assignments are treated as string literal in C. Our compiler also supports string concatenation. The result of this operation is represented using memory. The input of `string concatenation` uses the `+` operator, and the output is `str_concat(str1, str2)` helper function.

Optimization

- Constant propagation
- Dead code block removal
- Constant folding

Constant Propagation and Constant folding

For constant propagation, a dictionary is used to store variable values when an assignment statement is being evaluated. The values are only stored if they are of primitive type or value `NONE_TYPE`. The dictionary also stores a boolean state on whether the variable should be propagated or not. This is for when the variable is reassigned in an if statement, where it is difficult to determine the actual value of the variable after the if statement body during compile time. This also happens in while loop and for loop bodies. Constant folding is able to compute binary operations during compilation when the values are known. This also works in conjunction with constant propagation, so a binary operation of two variables can be computed if the two variable values are stored in the propagation dictionary.

Temporary Register Removal

In sprint 3, our translation from IR to target code involved making all temporary registers used to store values as variables in the target code. In sprint 4 we removed all instances of these temporary registers, and replaced them with the values they were associated with in the target code. This leads to the target code being more memory efficient since less variables are being declared. This also makes the code look cleaner and more of a direct translation from the input code.

Dead Code Block Removal

We focused on the variable optimization scope and identifying dead code blocks. The former is to keep the correctness of the code in the optimization step and the latter is to execute the optimization.

For the variable optimization scope, we use the pre-processing flag `pre_run` and the `variants` list. When the `pre_run` flag is set to `True`, most of the optimizations are disabled and code generation is omitted for lightweight code pre-processing. During this pre-processing, all variables on the LHS of assignment statements are stored in `variants`. We then reset the value of `variants` to its name in `var_dict` to stop their constant propagation.

In the for loops, we evaluate the loop bounds and the condition of the while loop. If the loop invariant is always out of bound or the condition is always `False` after evaluation, we

remove the whole code block. The pre-processing and `variants` comes in for nested loops. Those are used to detect variants in the loop to ensure the correctness of the code in the evaluation step.
For the if statement, we have 2 conditional variables `has_if_head` and `ignore_if`. The former is set to `True` when the previous `if` statement is removed, then we need to use the next `elif` to replace `if`. The `ignore_if` is set when the previous condition is always True, then we can ignore all subsequent `elif` and `else`.

Testing

Although we have our prior test cases that test individual components of the compiler, we decided to create a new set of test suites to better demonstrate the compiler's features and capabilities. Some were cherry picked from existing tests, some were created fresh.

Ensure you have testing dependencies installed:

```
pip install pytest
pip install pytest-xdist    ### NEW ###
```

All new tests are in the older `tests/`, we have 4 folders to demonstrate 4 different sets of scenarios.

- Tests in `compile/` aim to demonstrate all features of the compiler. These tests should go through and we will use gcc to make sure the output c code compiles.
- Tests in `error/` aim to demonstrate how the compiler handles all kinds of error, including in the preprocessing stage, parser stage, type checking stage and runtime.
- Tests in `opt/` aim to demonstrate how the optimization flag changes the output c code while ensuring the behavior of the final program is unchanged.

To run all the tests prepared for sprint 4 (31 test cases): `pytest --forked test.py`
To run all prior tests (125 test cases): `pytest --forked`
The argument forked ensures each test is run in a separate python environment. There are problems if we don't do that.

compile/

To run these only: `pytest --forked 'test.py::test_compile'`

The setup for these tests are pretty simple – each file shows a closely related set of language features and the file names should give a pretty good idea on what each is about. These tests are compiled with optimization off.

error/

To run these only: `pytest --forked 'test.py::test_error'`
Each test contains an error that we expect. The first two lines in each file should be comments. The first comment is a brief description on what the error is. The second line is the error message we expect from the compiler.
The test runner runs each of them, expecting an error and will check if each word in the second comment is contained in the error message.
If the special keyword `[RUNTIME]` appears in the first line, then the test runner will run the executable and catch the error from there instead.
All tests are compiled with no optimization.

opt/

To run these only: `pytest --forked 'test.py::test_opt'`
These tests show how the optimization flag changes the result code. Additionally we also run the executable to show that the program behavior is the same.
After running the test cases once, you should be able to find `<test_name>_opt_on.c` and `<test_name>_opt_off.c`.
The test runner compiles the source twice, once with optimization flag set and once unset. It then compiles and executes these files and assert their output is exactly the same (in the code we just printed the final values of all variables).

Playground

To make ad-hoc testing easier, we created an easy to use command line interface to compile and run programs. To use it:

1. Create your input file in `playground/`, name it `<name>.py`
2. Run `python compiler.py <name>`

Two flags are available:

```
-o, --opt   Enable optimization
-r, --run   Run the code if successfully compiled
```