Sprint 1 Documentation

Abstract Syntax Tree

Our components in the AST are represented as python classes, which use `@dataclass` decorator to reduce the boilerplate code required, making our tree more readable and easier to verify its correctness.

In our CFG, `expression` is the type which all input will end up with. Its corresponding node, `Expression,` is used as an intermediate node, which can be reduced from all nodes. All nodes are capable of being the root in our AST.

The `PrimitiveLiteral` node represents all the primitive values in our compiler. Its attribute `name` stores the type, and `value` to store the actual value. The `NonPrimitiveLiteral` node serves the same purpose for non-primitives. Instead of storing a single value, we use the `children` attribute to store a list of sub-nodes.

All binary operations are represented as `BinaryOperation` nodes. This node uses `left`, `operator` and `right` to indicate its left, operator and right child respectively. Its left and right child have an `Expression` type and the operator is stored as a string. Node `UnaryOperation` has a similar structure, but it only has an `operator` and `right` child.

In our compiler design, all variables can be static typed. We have a `Type` node to store the information of input type. Its attribute `value` is either `PrimitiveType` or `NonPrimitiveType`. Both of these nodes have `value` attributes to store the primitive type of input. A `name` attribute is also present in `NonPrimitiveType` to store non-primitive types of input. An input, like `[3],` will have `NonPrimitiveType` with `name='list'` and `value=PrimitiveType(value='int').`

Variable assignment node `Assignment` has three attributes, `left, right` and `type`. The `left` child must be an `Id`, which has `name` to store as an attribute. The `right` child can be any type of node to assign to the `left` child, and `type` must be a `Type` node. In the backend, we will check if the `type` matches the `right child’s` type.

The parser splits each line of input code as a `Statement`. A statement could be an assignment, if-elif-else statement, a for or while loop, and a function declaration or call. Statements that have bodies such as if statements can have a list of child statements. Statements that have no “parent” statement are outputted as individual AST nodes in the output JSON file.

Language Features Yet to be Completed

1. Enforcing if and “Else” statement is present, it must come directly after an if statement
2. Enforcing return statement must be inside a function definition

Testing

We use pytest to run our test cases. We have test cases for the lexer and parser.
To run all tests:

1. Ensure you have pytest installed: `pip install pytest`
2. `cd sprint1/`
3. `pytest`

We have about 40 test cases for the lexer, including tests for edge cases; they are in `lex_test.py`. To run this test suite only, run `pytest lex_test.py`.

We have about 22 test cases for the parser. Since the input is usually multiline, we create one input file for each test case. Alongside with input, we also have an expected output file that is being checked against during testing. We created a script that can automatically discover these tests. To run them, run `pytest parser_test.py`.

Specifically, the test cases for the parser is organised as follows:

- `tests/<test_id>_input.txt`: parser input.
- `tests/<test_id>_output.json`: expected parser output, converted to JSON
- `tests/<test_id>_received.json`: actual parser output, converted to JSON. This file is not saved in version control and is automatically cleaned up between each run.
- `tests/<test_id>_received.diff`: a diff between the expected output and actual output. This file only exists if they are different.