Extending Programming Language

with Switch Case, Timer, and Working Directory Functionalities

Structure for Programming Languages

Abhishek Pandit

May 6, 2025

Executive Summary

Overview

This project aims to extend the functionalities of an existing programming language with three new features:

- **switch case**
- **timer(<name>)**
- **presentworkingdirectory()**

These additions enhance the language functionalities for the control flow, performance tracking, and system interaction.

Goals and Objectives

- Implementing “switch case” for improved conditional branching
- Adding functionality “timer(<name>)” to measure the time taken by specific code sections.
- Adding functionality “presentworkingdirectory()” to retrieve and display current working directory

Summary of accomplishments and challenges

Successfully implemented all of the features, tested the program to get the evidence that the feature works perfectly, solved all the challenges and integrated it into the existing programming language.

Problem Statement

What problem or needs does the project address?

The project addresses the need for more flexible and easy-to-understand and implement control flow. It further extends better functionalities for example to check the execution time of a code or modules or anything, and can better evaluate which things could be changed to optimize it.

Who are the stakeholders or intended users?

Teachers, students and users of the extended programming language and those who need advanced features for their programs.

Why is the problem important or interesting?

These features improve the languages usability, provide better debugging capabilities and offer system information which are crucial for certain types of applications.

System Overview

The extended functionalities aligns with the teachings of Abstract Syntax Tree (AST), the addition of a declaration in the tokenizer, proper definition in the parser, and evaluating accordingly after splitting into the tree in the evaluator.

Implementation Details

Switch-case: First, it gets tokenized in the tokenizer. It gets converted to the list of tags [{“tag”: “switch”, “value”: “switch”, “position”: 5},...]. Then, it goes into the parser where it is parsed/split into tag, expression, and cases, and many more. It then goes into the evaluator to get evaluated.

Time function: Keywords are added in the tokenizer and it directly gets evaluated in the evaluator. The time with the parameter as a string gets added to the dictionary as a key and value as the current time. When the time with the same parameter is invoked again, it uses the new current time and the time stored in the dict to calculate the total time taken for that invocation and its previous invocation.

Presentworkingdirectory function: Keywords are added in the tokenizer and it directly gets evaluated in the evaluator. In the evaluator, it directly gets the current working directory from **os.getcwd()** and prints it out on the screen.

Testing and Validation

Each of the functionalities of switch-case, time, and presentworking directory is tested and evaluated to the highest standard.

Examples of test cases:

Test code for switch-case:

```
def test_evaluate_switch_statement():

    print("test evaluate_switch_statement")

    # evaluating teh switch statements using different cases ...

    env = {}

    equals('x=2; switch(x) { case 1: { y = 10 } case 2: { y = 20 } }; y', env, 20, {'x': 2, 'y': 20})

    equals('x=3; switch(x) { case 1: { y = 10 } default: { y = 30 } }; y', env, 30, {'x': 3, 'y': 30})

    equals('x=4; switch(x) { case 1: { y = 10 } case 2: { y = 20 } }; y=5;', env, 5, {'x': 4, 'y': 5})

    equals('x=1; switch(x) { case 1: { y = 10; break; } case 2: { y = 20 } }; y', env, 10, {'x': 1, 'y': 10})

    equals('y=20;x=3; switch(x) { case 1: { y = 10 } case 2: { y = 20 } case 3: { y = y + 5 } }; y', env, 25, {'x': 3, 'y': 25})

    equals('y=5;x=3; switch(x) { case 1: { y = 10 } case 2: { y = 20 } default: { y = y + 1 } }; y', env, 6, {'x': 3, 'y': 6})

    equals('x=5; switch(x) { case 1: { y = 10 } default: { y = 30; break; } }; y', env, 30, {'x': 5, 'y': 30})

    equals('x="b"; switch("a"+x) { case "ab": { y = 1 } case "ac": { y = 2 } }; y', env, 1, {'x': 'b', 'y': 1})

    # print("the value of the env is :", env)

    equals('switch(1) {}; x=1;', {}, 1, {'x': 1})
```

Test code for time(<name>):

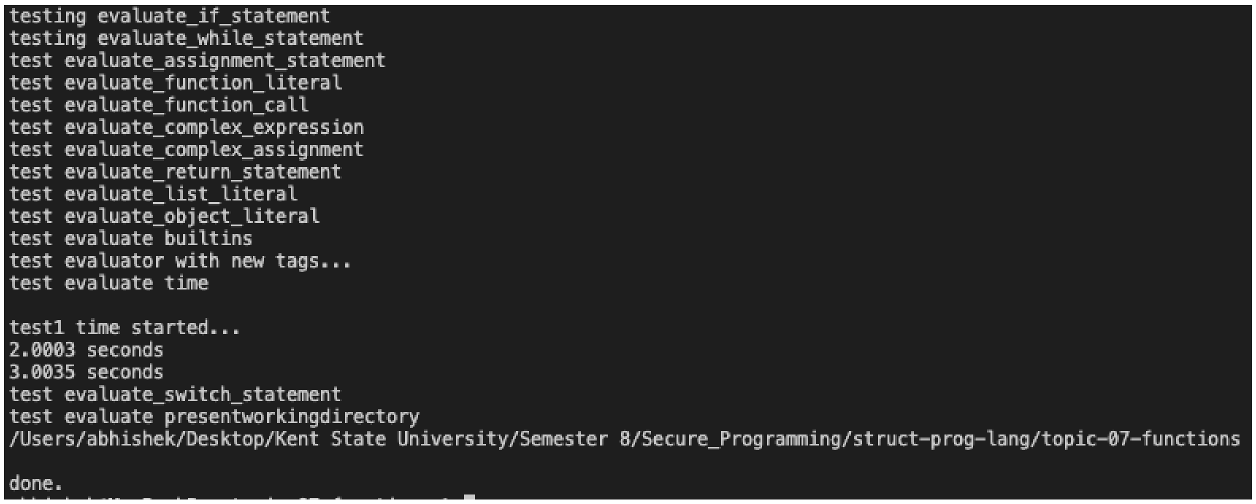
```
def test_evaluate_time():

   """Tests the custom time() built-in function."""

   print("test evaluate time\n")

   global _timer_starts

   _timer_starts = {} # ensuring timers are cleared before starting the test

   environment = {}

   tolerance = 0.1 # we need to allow for some inaccuracy in time.sleep() and
execution (in seconds)

   # test case 1: initializating for the first time

   # firts call should return None (just tells us that the time has started) and will
start the timer

   code1 = 'time("test1")'

   ast1 = parse(tokenize(code1))

   result1, _ = evaluate(ast1, environment)

   assert result1 is None, f"Expected None on first call of time('test1'), got
{result1}"

   assert "test1" in _timer_starts, "Timer 'test1' was not added to _timer_starts"

   # we need to capture the start time

   start_time_1 = _timer_starts["test1"]

   # test case 2: calling second time after the first call with a custom delay

   delay1 = 2

   time.sleep(delay1)

   code2 = 'time("test1")'

   ast2 = parse(tokenize(code2))

   result2, _ = evaluate(ast2, environment)
   assert isinstance(result2, (int, float)), f"Expected a number on second call of
time('test1'), got {type(result2)}"

   assert abs(result2 - delay1) < tolerance, f"Expected elapsed time ~{delay1}s for
time('test1'), got {result2}s"

   assert _timer_starts["test1"] > start_time_1, "Timer 'test1' start time was not
updated after second call"

   start_time_2 = _timer_starts["test1"]

   # test case 2: calling the time after some more delay

   delay2 = 3

   time.sleep(delay2)

   code3 = 'time("test1")'

   ast3 = parse(tokenize(code3))

   result3, _ = evaluate(ast3, environment)

   assert isinstance(result3, (int, float)), f"Expected a number on third call of
time('test1'), got {type(result3)}"

   assert abs(result3 - delay2) < tolerance, f"Expected elapsed time ~{delay2}s for
third call of time('test1'), got {result3}s"

   assert _timer_starts["test1"] > start_time_2, "Timer 'test1' start time was not
updated after third call"

   #test case 4: checking some of the edge case scenario to check whether it fails or
not

   _timer_starts = {}

   environment = {}

   # testing no arguments passing

   try:

       evaluate(parse(tokenize('time()')), environment)
```

```
        assert False, "time() with no arguments should have raised an Exception."

    except Exception as e:

        assert "requires exactly one string argument" in str(e), f"Wrong error for
time(): {e}"

    # testing wrong argument passing

    try:

        evaluate(parse(tokenize('time(123)')), environment)

        assert False, "time(123) should have raised an Exception."

    except Exception as e:

        assert "requires exactly one string argument" in str(e), f"Wrong error for
time(123): {e}"

    # testing multiple parameter passing in the timer

    try:

        evaluate(parse(tokenize('time("a", "b")')), environment)

        assert False, 'time("a", "b") should have raised an Exception.'

    except Exception as e:

         assert "requires exactly one string argument" in str(e) or \

                "takes 1 positional argument but 2 were given" in str(e) or \

                "Expected ')'" in str(e), \

                f'Wrong error for time("a", "b"): {e}'
```

Test code for presentworkingdirectory():

```
def test_evaluate_presentworkingdirectory():

   print("test evaluate presentworkingdirectory")

   captured_output = io.StringIO()

   sys.stdout = captured_output

   code = 'presentworkingdirectory()'

   ast = parse(tokenize(code))

   evaluate(ast, {})

   sys.stdout = sys.__stdout__

   expected_output = os.getcwd() + "\n"

   assert captured_output.getvalue() == expected_output, f"Expected
'{expected_output}', but got '{captured_output.getvalue()}'"
```

Output for all at once:



```
testing evaluate_if_statement
testing evaluate_while_statement
test evaluate_assignment_statement
test evaluate_function_literal
test evaluate_function_call
test evaluate_complex_expression
test evaluate_complex_assignment
test evaluate_return_statement
test evaluate_list_literal
test evaluate_object_literal
test evaluate builtins
test evaluator with new tags...
test evaluate time

test1 time started...
2.0003 seconds
3.0035 seconds
test evaluate_switch_statement
test evaluate presentworkingdirectory
/Users/abhishek/Desktop/Kent State University/Semester 8/Secure_Programming/struct-prog-lang/topic-07-functions

done.
```

Results and Evaluations

We can evaluate from the testing part that each of added functionalities works perfectly and are very much optimized. Thus, it can be inferred that the project was successful.

Future Work and Recommendation

There is a catch in the timer function. Once it starts it does not stop. Stopping functionality can be extended further. Also, more features and functionality can be build upon them.

User Manual

To install and run the application, you can just download the python file and run the runner.py using `python3.12 runner.py` or `python3.12 runner.py <filename>`

Appendices

Use this link to go to GitHub repo:
https://github.com/AbhishekPanditPro/structure_final