Extending The *Trivial* Language: Adding “*for*” and a “*loop*” Idiom

---

CS33101 Structures of Programming Languages
Maxim Mamotlivi
Professor Greg Delozier
Computer Science Dept., Kent State University
1300 University Esplanade, Kent, OH 44240, United States
Date of Submission: May 07, 2025

Executive Summary

A Brief Overview

This project is the culmination of a semester's worth of effort to understand the structure of programming languages by participating in the development of a language and its necessary components. Throughout the course of lecture during the Spring 2025 semester for CS33101 Structures of Programming languages, Professor Delozier has continually developed *Trivial*, a Python-based language seeking to implement JavaScript-like features.

Trivial is designed as a lightweight, interpreted language with a syntax and semantics inspired by both Python and JavaScript; utilizing EBNF grammar and recursive descent parsing of abstract syntax trees. The language is somewhat of an experiment created to explore operations in how programming languages are structured, parsed, and executed as well as an educational tool for understanding language design and implementation.

Goals and Objectives

At the close of the final lecture class, Professor Delozier directed students to extend the language largely at their discretion, recommending the implementation of statement features such as the "switch" statement or others of similar complexity.

Thus, the chosen goal of this project was to further extend the loop-feature suite of the language, the objectives therein including:

- Completion of the "for" loop statement.
- The experimental addition of a "loop" idiom primarily for the reduction of boilerplate and exploration of potential implementation techniques for such a statement.

Major Deliverables and Features

**Please see the Marp slides "parse-slides.md" file included in Professor Delozier's repository for an overview of the base features of the Trivial language.*

A modified version of the "topic-11-completed-language" *Trivial* language module from Professor Delozier GitHub repository with added "for" loop functionality and a novel "loop" idiom. The "for" loop functions as any traditional C-style for loop would be expected to. The idiomatic loop, denoted by the "loop" keyword, automatically decrements its global iterator from a supplied integer or integer variable and resets its iterator upon completion. The idiomatic loop may simplify the coding process by reducing, in aggregate, a considerable amount of boilerplate code, typically associated with basic "for" or "while" loop structures.

Summary of Accomplishments and Challenges

Successfully implemented and tested both loop features, extending *Trivial's* feature set. Challenges included ensuring the "loop" idiom's iterator reset mechanism was functional and that the global iterator was stably accessible outside the loop using the "iter" keyword. Debugging certain edge cases in recursive descent parsing was time-intensive.

Problem Statement

This project addresses the need to expand *Trivial's* feature set to support more complex control flow, making it a more practical tool. Stakeholders include Professor Delozier and CS33101 students learning language design. Enhancing loop constructs can be important to demonstrate real-world language functionality as well as potential optimizations for boilerplate minimization. Because loop statements are ubiquitous among programming languages, minor improvements in the coding experience might prove cumulatively beneficial.

System Overview / Solution Description

Again, *Trivial* uses a recursive descent parser to process EBNF grammar, generating ASTs for execution. New loop constructs integrate into the parser and interpreter pursuant to the existing paradigm of development, testing, etc.
**Please see the Marp slides "parse-slides.md" file included in Professor Delozier's repository for a more detailed overview of the base features of the Trivial language.*

Technologies Used:
- Language: Python 3
- Tools: Git, Python standard libraries, Python 're' Regular Expression library

Overview of Subsystems
- Tokenizer: added "loop" and "iter" keywords
- Parser: Extended "for" and "loop" grammar rules, subsequent AST parsing and tests.
- Evaluator: Updated to execute "for", "loop" constructs using "iter" global, along with the requisite tests.

Notable Design Decisions
- "For" loop mimics C-style syntax for familiarity.
- "Loop" will automatically increment/decrement an integer value toward zero, supports infinite looping, and using iter to access the global iterator for whatever reason the user might desire.

Development Process

**Please see Professor Delozier's GitHub repository activity tab for CS33101 Structure of Programming Languages as the majority of the semester was a code-along during lecture*

The final week of the semester was spent solo-developing the aforementioned objectives and writing corresponding documentation as directed during the preceding last week of classes. It can be said that the whole development process was generally Agile.

- Tools: VSCode, GitHub
- Source Control: Git, forked from Professor Delozier's repository.

Implementation Details

Key Algorithms

- **For Loop**: Parses init, condition, update, and body, generating AST nodes for execution.
- **Loop Statement**: Parses optional iter_expr (iteration count or None for infinite), tracks iterations via iter keyword.

Codebase Structure

- Modified parser.py to add grammar rules for "for" and "loop".
- Updated evaluator.py for execution logic.

Diagrams

Rough Outline of AST for "for" Loop:

```
ForNode
├── Init: AssignNode
├── Condition: BinaryOpNode
├── Update: AssignNode
└── Body: StatementListNode
```

Rough Outline of AST for "loop" Statement:

```
LoopNode
├── IterExpr: NumberNode | None
└── Body: StatementListNode
```

Testing and Validation

The testing approach in this project builds on the functional testing style used in the original code, where each part of the parser is checked with specific examples to make sure it works as expected. By systematically running small pieces of code through the parser and comparing the output to the expected results, the tests provide solid coverage and help catch mistakes early. This method has been extended in the feature-test.t file, which now includes additional tests for new features and edge cases introduced during development.

Some Example Tests (Excerpts from code)

“for” and “loop” tests from parser.py

```
# also added the test for the above
def test_parse_loop_statement():
    """
    loop_statement = "loop" [ "(" expression ")" ] statement_list
    """
    print("testing parse_loop_statement...")

    # loop with a number
    ast, tokens = parse_loop_statement(tokenize("loop(3){print iter}"))
    assert ast == {
        "tag": "loop",
        "iter_expr": {"tag": "number", "value": 3},
        "body": {
            "tag": "statement_list",
            "statements": [
                {"tag": "print", "value": {"tag": "iter"}}
            ]
        }
    }


def test_parse_for_statement():
    """
    for_statement = "for" "(" [ assignment_expression ] ";" [ expression ] ";" [
assignment_expression ] ")" statement_list
    """
    print("testing parse_for_statement...")

    # Basic for loop: for (i=0; i<5; i=i+1) {print i}
    ast, tokens = parse_for_statement(tokenize("for(i=0;i<5;i=i+1){print i}"))
    assert ast == {
        "tag": "for",
        "init": {
```

```python
            "tag": "assign",
            "target": {"tag": "identifier", "value": "i"},
            "value": {"tag": "number", "value": 0}
        },
        "condition": {
            "tag": "<",
            "left": {"tag": "identifier", "value": "i"},
            "right": {"tag": "number", "value": 5}
        },
        "update": {
            "tag": "assign",
            "target": {"tag": "identifier", "value": "i"},
            "value": {
                "tag": "+",
                "left": {"tag": "identifier", "value": "i"},
                "right": {"tag": "number", "value": 1}
            }
        },
        "body": {
            "tag": "statement_list",
            "statements": [
                {"tag": "print", "value": {"tag": "identifier", "value": "i"}}
            ]
        }
    }

    # for loop with empty init: for(;i<5;i=i+1){print i}
    ast, tokens = parse_for_statement(tokenize("for(;i<5;i=i+1){print i}"))
    assert ast == {
        "tag": "for",
        "init": None,
        "condition": {
            "tag": "<",
            "left": {"tag": "identifier", "value": "i"},
            "right": {"tag": "number", "value": 5}
        },
        "update": {
            "tag": "assign",
            "target": {"tag": "identifier", "value": "i"},
            "value": {
                "tag": "+",
                "left": {"tag": "identifier", "value": "i"},
                "right": {"tag": "number", "value": 1}
            }
        },
        "body": {
            "tag": "statement_list",
```

```
            "statements": [
                {"tag": "print", "value": {"tag": "identifier", "value": "i"}}
            ]
        }
    }

    etc.
```

“for” and “loop” tests from evaluator.py

```
def test_evaluate_loop_and_iter():
    print("test evaluate loop and iter")
    # Counted loop, positive
    equals("sum=0; loop(3){sum=sum+iter}; sum", {}, 6, {"sum": 6})
    # Counted loop, negative
    equals("sum=0; loop(-2){sum=sum+iter}; sum", {}, -3, {"sum": -3})
    # Infinite loop, break after 3 iterations
    equals("sum=0; loop() {sum=sum+1; if(sum==3){break}}; sum", {}, 3, {"sum": 3})
    # iter is accessible before, inside, and after the loop
    equals("iter=42; a=iter; loop(2){b=iter}; c=iter", {}, 0, {"a": 42, "b": 1, "c": 0})
    # Nested loops: iter should update on each loop
    equals("outer=0; inner=0; loop(2){outer=iter; loop(2){inner=iter}}; outer+inner", {}, 2,
{"outer": 1, "inner": 1})


# added by Maxim Mamotlivi
def test_evaluate_for_statement():
    print("test evaluate_for_statement")

    equals("sum=0; for(i=0; i<5; i=i+1) {sum=sum+i}", {}, None, {"sum": 10, "i": 5})
    equals("sum=0; for(; i<5; i=i+1) {sum=sum+i}", {"i": 0}, None, {"sum": 10, "i": 5})
    equals("sum=0; for(i=0; ; i=i+1) {sum=sum+i; if(i==2) {break}}", {}, None, {"sum": 3, "i":
2})
    equals("sum=0; for(i=0; i<5;) {sum=sum+i; i=i+1}", {}, None, {"sum": 10, "i": 5})
    equals("sum=0; for(i=0; i<5; i=i+1) {if(i==2) {continue} sum=sum+i}", {}, None, {"sum": 8,
"i": 5})

    # for loop with return inside function
    code = """
        function f() {
            sum=0;
            for(i=0; i<5; i=i+1) {
                sum=sum+i;
                if(i==3) {return sum}
            };
            return 999
        };
```

```
    f()
  """
  # annoyingly long ast
  equals(code, {}, 6, {LONG AST OMITTED FOR BREVITY})
```

“for” and “loop” tests from feature-test.t

```
// ------------------------------------------------------------
// Loop Idiom
// ------------------------------------------------------------
print("Testing loop idiom...");
count = 0;
loop(10) {
    count = count + 1
}
assert count == 10;

// ------------------------------------------------------------
// For Loops
// ------------------------------------------------------------
print("Testing for loops...");
count = 0;
for (i=1; i<10; i=i*2) {
    count = count + 1
}
assert count == 4;
```

Results and Evaluation

Success!

Both loop constructs were implemented and function as intended.

Metrics

- All tests pass.
- Stable parser with no crashes on valid input.
- Substantially minimize necessary loop boilerplate via “loop” idiom

Lessons Learned

- Parser modifications require careful grammar design.
- Operating within an established testing paradigm has enormous advantages.

Future Work and Recommendations

Remaining Issues

- Limited error handling for malformed loop syntax.
- “Loop” statement’s infinite mode may need more explicit termination controls.
- Professor Delozier frequently discussed implementing Tail-Recursion, however this seems a considerable task.

Future Extensions

- Add the “switch” statement.
- Continue improving error messages for better debugging.

User Manual

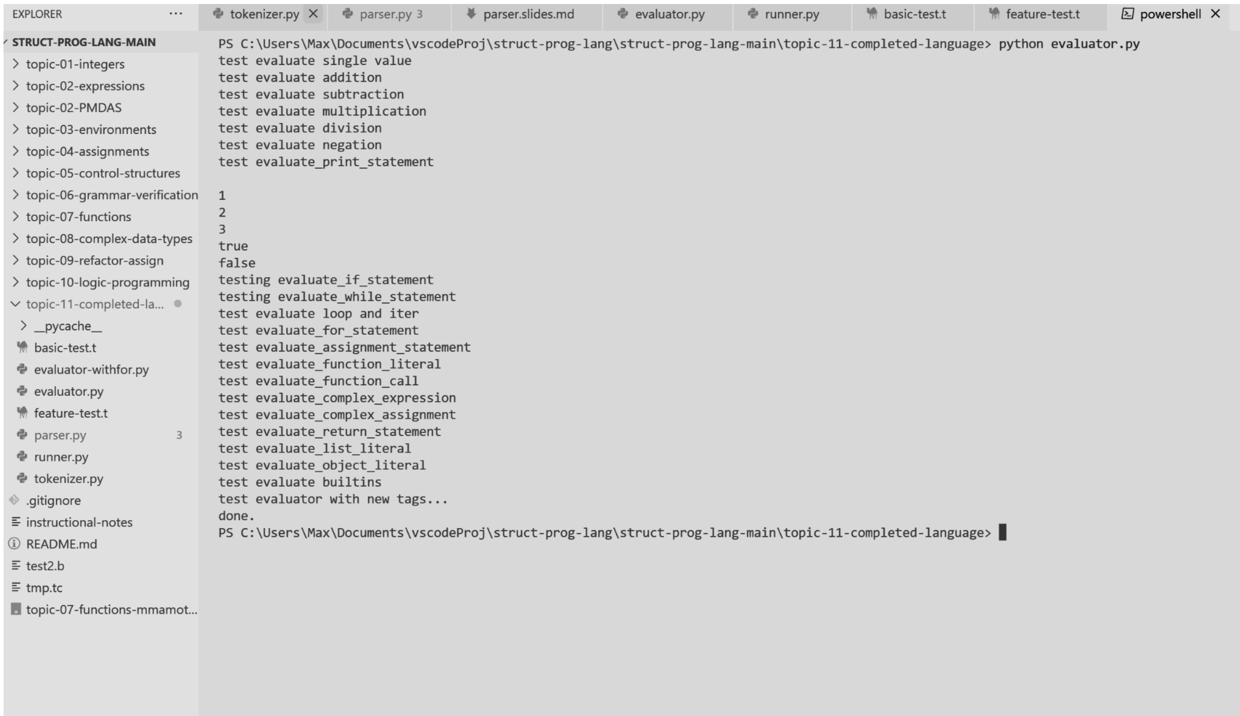
Installation/Running

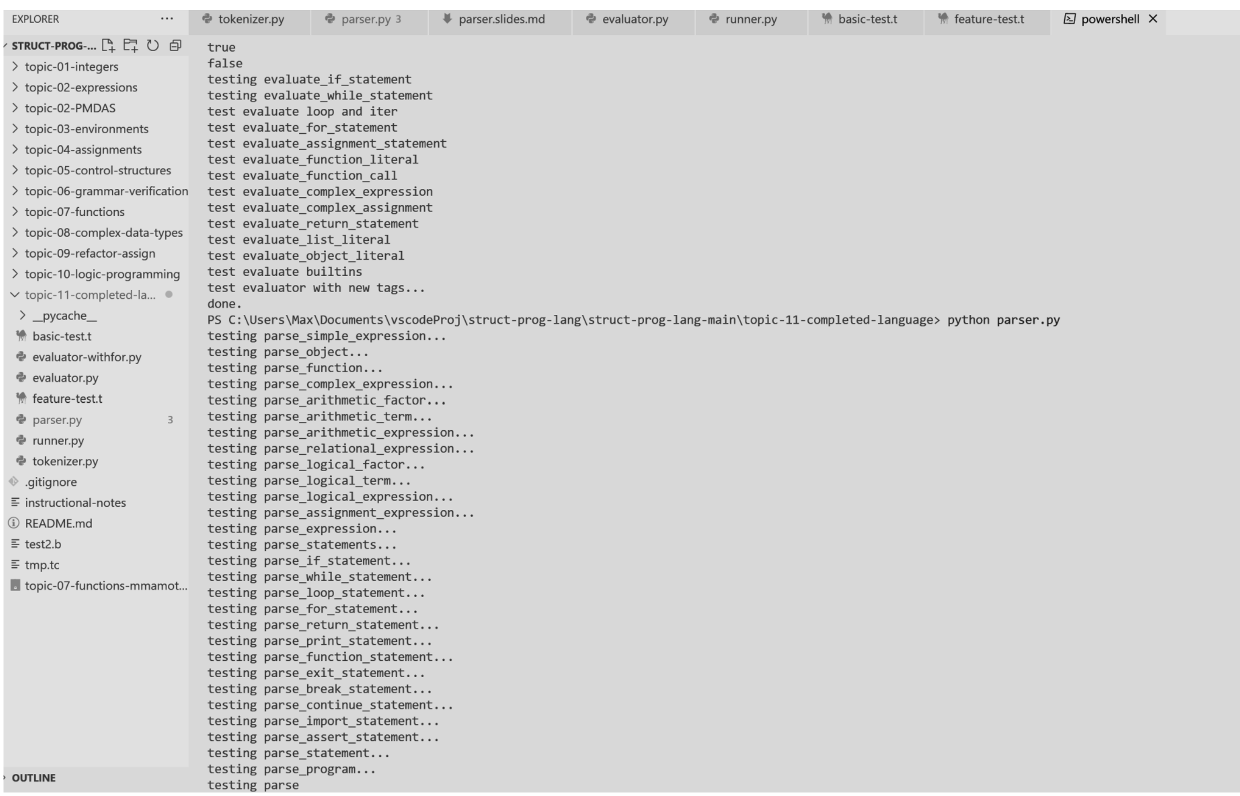
1. Access original repository: https://github.com/gregdelozier/struct-prog-lang
2. Access modified version of the above: https://github.com/mm3717/CS33101-StructureOfProgLangProject
3. Run: python tokenizer.py
4. Run: python parser.py
5. Run: python evaluator.py
6. Run: python runner.py feature-test.t

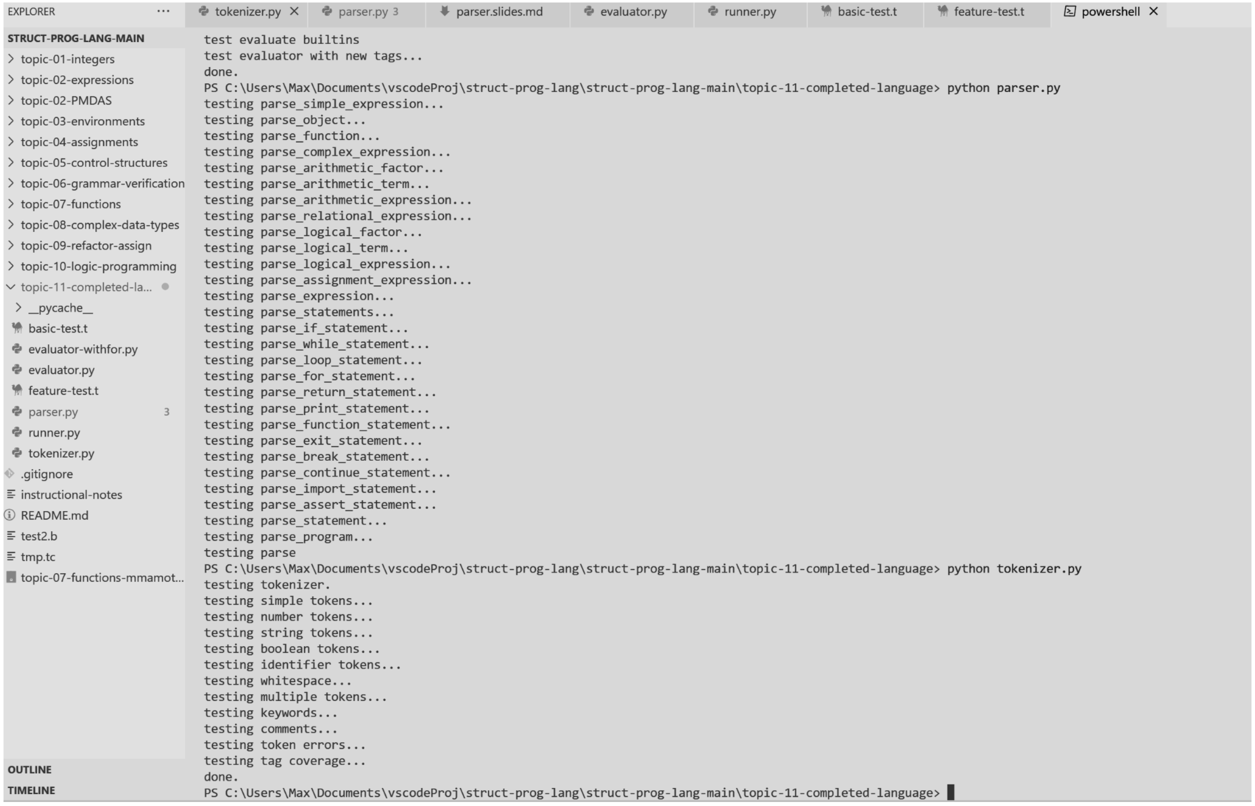
Production Deployment

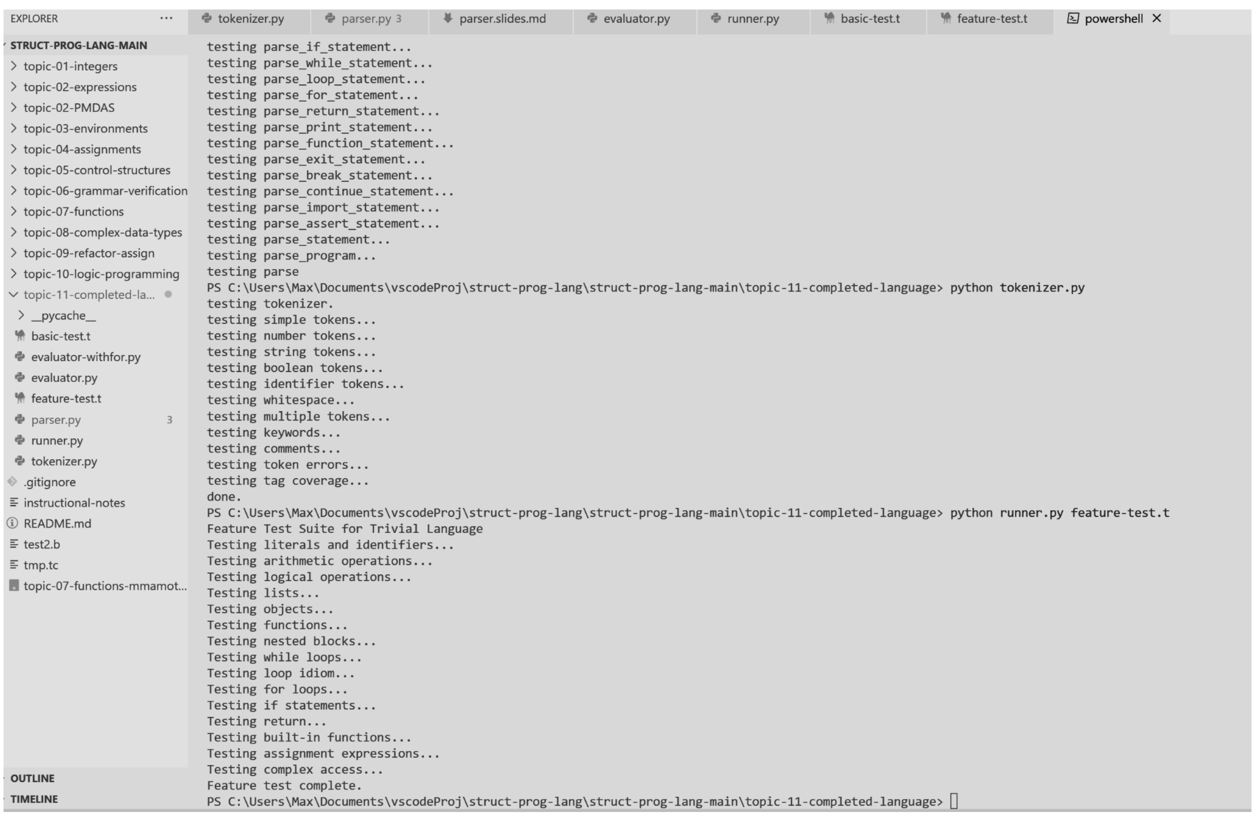
Available locally via repository.

Screenshots






EXPLORER
tokenizer.py
parser.py 3
parser.slides.md
evaluator.py
runner.py
basic-test.t
feature-test.t
powershell
STRUCT-PROG-LANG-MAIN
topic-01-integers
topic-02-expressions
topic-02-PMDAS
topic-03-environments
topic-04-assignments
topic-05-control-structures
topic-06-grammar-verification
topic-07-functions
topic-08-complex-data-types
topic-09-refactor-assign
topic-10-logic-programming
topic-11-completed-la...
__pycache__
basic-test.t
evaluator-withfor.py
evaluator.py
feature-test.t
parser.py 3
runner.py
tokenizer.py
.gitignore
instructional-notes
README.md
test2.b
tmp.tc
topic-07-functions-mmamot...
OUTLINE
TIMELINE
test evaluate builtins
test evaluator with new tags...
done.
PS C:\Users\Max\Documents\vscodeProj\struct-prog-lang\struct-prog-lang-main\topic-11-completed-language> python parser.py
testing parse_simple_expression...
testing parse_object...
testing parse_function...
testing parse_complex_expression...
testing parse_arithmetic_factor...
testing parse_arithmetic_term...
testing parse_arithmetic_expression...
testing parse_relational_expression...
testing parse_logical_factor...
testing parse_logical_term...
testing parse_logical_expression...
testing parse_assignment_expression...
testing parse_expression...
testing parse_statements...
testing parse_if_statement...
testing parse_while_statement...
testing parse_loop_statement...
testing parse_for_statement...
testing parse_return_statement...
testing parse_print_statement...
testing parse_function_statement...
testing parse_exit_statement...
testing parse_break_statement...
testing parse_continue_statement...
testing parse_import_statement...
testing parse_assert_statement...
testing parse_statement...
testing parse_program...
testing parse
PS C:\Users\Max\Documents\vscodeProj\struct-prog-lang\struct-prog-lang-main\topic-11-completed-language> python tokenizer.py
testing tokenizer.
testing simple tokens...
testing number tokens...
testing string tokens...
testing boolean tokens...
testing identifier tokens...
testing whitespace...
testing multiple tokens...
testing keywords...
testing comments...
testing token errors...
testing tag coverage...
done.
PS C:\Users\Max\Documents\vscodeProj\struct-prog-lang\struct-prog-lang-main\topic-11-completed-language>


EXPLORER
tokenizer.py
parser.py 3
parser.slides.md
evaluator.py
runner.py
basic-test.t
feature-test.t
powershell
STRUCT-PROG-LANG-MAIN
topic-01-integers
topic-02-expressions
topic-02-PMDAS
topic-03-environments
topic-04-assignments
topic-05-control-structures
topic-06-grammar-verification
topic-07-functions
topic-08-complex-data-types
topic-09-refactor-assign
topic-10-logic-programming
topic-11-completed-la...
__pycache__
basic-test.t
evaluator-withfor.py
evaluator.py
feature-test.t
parser.py 3
runner.py
tokenizer.py
.gitignore
instructional-notes
README.md
test2.b
tmp.tc
topic-07-functions-mmamot...
OUTLINE
TIMELINE
testing parse_if_statement...
testing parse_while_statement...
testing parse_loop_statement...
testing parse_for_statement...
testing parse_return_statement...
testing parse_print_statement...
testing parse_function_statement...
testing parse_exit_statement...
testing parse_break_statement...
testing parse_continue_statement...
testing parse_import_statement...
testing parse_assert_statement...
testing parse_statement...
testing parse_program...
testing parse
PS C:\Users\Max\Documents\vscodeProj\struct-prog-lang\struct-prog-lang-main\topic-11-completed-language> python tokenizer.py
testing tokenizer.
testing simple tokens...
testing number tokens...
testing string tokens...
testing boolean tokens...
testing identifier tokens...
testing whitespace...
testing multiple tokens...
testing keywords...
testing comments...
testing token errors...
testing tag coverage...
done.
PS C:\Users\Max\Documents\vscodeProj\struct-prog-lang\struct-prog-lang-main\topic-11-completed-language> python runner.py feature-test.t
Feature Test Suite for Trivial Language
Testing literals and identifiers...
Testing arithmetic operations...
Testing logical operations...
Testing lists...
Testing objects...
Testing functions...
Testing nested blocks...
Testing while loops...
Testing loop idiom...
Testing for loops...
Testing if statements...
Testing return...
Testing built-in functions...
Testing assignment expressions...
Testing complex access...
Feature test complete.
PS C:\Users\Max\Documents\vscodeProj\struct-prog-lang\struct-prog-lang-main\topic-11-completed-language>