

## Lab 10: Interpreter — MiniLang

CS1234: Small-Scale Application Development

### Background

When you write code in any programming language, something has to *understand* and *execute* it. An **interpreter** does exactly that — it reads source code line by line and executes each statement immediately. In this lab, you will build a simple interpreter in C for a tiny language called **MiniLang**.

### Problem Statement

(5 marks)

You are required to implement an interpreter in C that reads and executes a MiniLang program passed via a file. Your interpreter processes the file **line by line**, executing each statement as it is encountered.

**Note:** You are **guaranteed** that all input files are valid and error-free. Every variable will be declared before use, no variable will be declared more than once, and division by zero will never occur. You do **not** need to handle any error cases. Also, all computation results are guaranteed to stay within the bounds of `int`, so you do not need to handle overflow/underflow.

MiniLang supports five types of statements:

#### 1. Variable Declaration

```
let <var>
```

Declares an integer variable. A variable must be declared before it is used.

#### 2. Assignment

```
<var> = <value>
```

Assigns an integer literal to a declared variable.

#### 3. Arithmetic Expression

```
<var> = <operand1> <op> <operand2>
```

Evaluates a two-operand arithmetic expression and stores the result in `<var>`. Each operand is either a variable name or an integer literal. Supported operators: `+`, `-`, `*`, `/`.

**Division:** Use C integer division semantics (truncation toward zero). For example, `q = 7 / 3` stores 2, and `q = -7 / 3` stores -2.

#### 4. Unary Increment / Decrement

```
<var>++  
<var>--
```

Increments or decrements the value of a declared variable by 1, **in place**. Only **postfix** forms (`++` and `--`) are supported. A statement that uses a unary operator **will not** contain any other

operator — the two forms are mutually exclusive on any given line. Arbitrary spaces are allowed around the variable and operators, e.g. `count++`, `count ++`, and `count ++` are all valid.

## 5. Print Statement

```
print <var>
```

Prints the current value of the variable to `STDOUT`, followed by a newline.

## Input Format

---

Your program accepts the path to a MiniLang source file as a command-line argument:

```
./a.out program.mn
```

- Each **non-empty** line contains exactly one complete statement
- A newline terminates a statement
- Variable names may contain lowercase letters, uppercase letters, and underscore (`-`)
- Maximum variable-name length is **64 characters**
- All values are integers (possibly negative)
- There will be at most **100 variables** in any program
- Leading/trailing spaces are allowed on every line
- Spaces/tabs may appear between tokens (including unary statements)
- Empty lines or lines containing only spaces may appear and should be ignored
- **All input is guaranteed to be valid** — no error handling is required

## Output Format

---

For every `print` statement, output the value of the variable on a new line.

## Sample Test Cases

---

### Test Case 0

Input (`p0.mn`):

```
let a
let b
let c
a = 10
b = 3
c = a + b
print c
c = a * b
print c
```

Output:

```
13
30
```

**Explanation:**  $c = a + b$  evaluates to  $10 + 3 = 13$ . Then  $c = a * b$  evaluates to  $10 \times 3 = 30$ . Each `print` outputs the current value of `c`.

### Test Case 1

**Input** (p1.mn):

```
let total
let delta
total = 6
delta = total - 4
print total
print delta
let product
product = total * delta
print product
```

**Output:**

```
6
2
12
```

**Explanation:** `delta = total - 4` gives  $6 - 4 = 2$ . `product = total * delta` gives  $6 \times 2 = 12$ . Variables can be declared at any point before they are used.

### Test Case 2 — Unary Operators

**Input** (p2.mn):

```
let a
let b
a = 5
a++
print a
b = 3
b--
print b
```

**Output:**

```
6
2
```

**Explanation:** `a++` increments `a` from 5 to 6. `b--` decrements `b` from 3 to 2. The updated value is stored back in the variable immediately.

### Constraints

| Parameter                    | Limit                       |
|------------------------------|-----------------------------|
| Maximum number of variables  | 100                         |
| Maximum variable-name length | 64 characters               |
| Maximum line length          | 256 characters              |
| Maximum lines in input file  | 300                         |
| Integer values               | fit within <code>int</code> |

## Notes & Submission Guidelines

---

- Run `check.sh` to test against public test cases.
- Run `submit.sh` to submit.
- **Memory note:** Free any dynamically allocated memory before exit. Failure to do so will result in a **0.5 mark deduction**.

## Extras (Ungraded)

---

### Extension 1: Error Handling

Extend your interpreter to detect and report the following errors, printing the corresponding message and halting immediately:

| Situation                                     | Error Message                          |
|-----------------------------------------------|----------------------------------------|
| Variable used before declaration              | Error: undeclared variable <var>       |
| Variable declared more than once              | Error: redeclaration of variable <var> |
| Assigning to an undeclared variable           | Error: undeclared variable <var>       |
| Using an undeclared variable in an expression | Error: undeclared variable <var>       |
| Applying ++ or -- to an undeclared variable   | Error: undeclared variable <var>       |
| Division by zero                              | Error: division by zero                |

### Extension 2: if Statements

Support conditional statements of the form:

```
if <var> <op> <value> : <statement>
```

Execute <statement> only if the condition evaluates to true.

### Extension 3: Chained Expressions

Allow expressions with two operators:

```
z = a + b - 3
```