CMPE150 2025 Fall Homework 5
Japanese Programming Language (JPL)

- Read **all instructions** very carefully.
- There may be minor updates to the homework. Follow the Moodle announcements carefully.
- Name your solution file: **hw5.py**. Submit only this file without zip.
- Your code should run in Python version > 3.10

Important: You are not allowed to use any function/library/class/structure that was not covered in the classroom. You are basically constrained to the use of for/while/if-else statements, lists, strings, f-strings, dictionaries, file operations (including pickle dump). But you cannot define a function inside a function, you cannot use the eval() function, you cannot use regular expressions, you cannot use complex data types, etc.

You are NOT allowed to use **try except** statements in this homework.

I recommend you turn off all support functionalities of large language models/coding interfaces. Do not copy-paste any code from them. You might get help from LLMs or your friends in a natural language (in English or Turkish) on the algorithmic part, but do not request any code segment directly. Write down the code yourselves, even without using the auto-complete functionality.

Large number of example inputs will be provided in the following link:
https://drive.google.com/drive/folders/1cwUzQZZqQ1h8CPYxzK_m8NtleXdWbZL5?usp=sharing

In this homework, you are given the task of compiling and executing a program written in a new programming language, JPL. Your program should be able to handle a given program segment. Summary:

- Your program should be named **hw5.py**.
- Given the command line argument `-compile <input-file> <obj-file>`, your program will read `<input-file>`, compile it and generate a binary file `<obj-file>`. A non-empty `<input-file>` will always be available.
-
- Given command line argument `-execute <obj-file> <output-file>`, your program will read `<obj-file>`, which you previously generated, execute it and write the output into the text file `<output-file>`.
- You can assume that there is no error in the command line arguments. `<input-file>` and `<obj-file>` are in the same directory when your script is run with the `-compile` and `-execute` arguments, respectively.

BACKGROUND

1. Compilation — Translating Source Code into a Lower-Level Form

Compilation is the process of translating human-readable source code (e.g., Java, C, C++, JPL) into machine/executable code.

What happens during compilation?

1. **Lexical Analysis:** The compiler breaks the source code into tokens (keywords, identifiers, operators).
2. **Syntax Analysis (Parsing):** It checks if the code follows the language grammar (e.g., semicolons, braces, correct sentence structure).
3. **Semantic Analysis:** Ensures logical correctness (e.g., variable types match, functions exist, proper usage).
4. **Code Generation:** Converts the validated code into machine code/bytecode.

What happens in this step?

The compiler reads your source code (e.g., `.c`, `.cpp`, `.java, .jpl`) and converts it into a lower-level representation. Depending on the language, the output may be:

- Machine code (native executable) - C, C++
- Bytecode (for a virtual machine) - Java, C#, Python's .pyc
- Intermediate representation - Many modern compilers including JPL

What the compiler checks:

- Syntax errors (`missing ;`, unmatched `{}`)
- Undeclared variables
- Type mismatches (e.g., assigning a string to an int)
- Missing return statements
- Wrong method signatures
- Type errors (adding string to int, wrong function arguments)
- Scope errors (undefined variables)

Output:

- An executable file (e.g., `a.out`, `.exe`)
- OR bytecode (e.g., `Main.class`)
- OR intermediate code (e.g., **`<obj-file>`** in JPL)

Key point:

❗ **If compilation fails, `<obj-file>` is not generated, execution cannot begin.**

If the compiler finds problems during these steps, it stops and reports **compile error**.

---

2. Execution — Running the Compiled Code

Once compilation succeeds, the second stage begins.

The program is now executed by:

- CPU (native machine code, e.g., C/C++)
- Virtual Machine (bytecode run by JVM or .NET CLR)
- Interpreter (line-by-line execution, e.g., Python)
- Executer (from your intermediate representation in `<obj-file>` in JPL)

What happens during execution:

- Variables are created in memory
- Functions are called
- Loops run
- Input/output occurs
- Runtime errors may appear:
    - `ZeroDivisionError`
    - `NullPointerException`
    - Array out of bounds
    - File not found
    - Running out of memory

Characteristics

- The program starts normally, but might **crash during execution**.
    - Such a crash was not detected by the compiler because the code was syntactically and semantically valid.

Key point:

Runtime errors occur **after the program successfully compiles**, but **while it is running**.

❗ **Errors that appear during execution are runtime errors, *not* compile errors.**

Another Summary:

- Your program should be named **hw5.py**.
- Given the command line argument `-compile <input-file> <obj-file>`, your program will read `<input-file>`, compile it.
    - If there is no compile error, it generates a binary file `<obj-file>`.
    - If there is a compile error, `<obj-file>` is not generated. Instead your program should report the compile error by raising "Compile error" with the line number.
    - Not that the program should be parsed line-by-line in a sequential order. In case there are multiple compile errors, the one in the earliest line should be reported.
- Given command line argument `-execute <obj-file> <output-file>`, your program will read `<obj-file>`, which you previously generated, execute it, and write the output into the text file `<output-file>`.
    - Your program will be executed until the end or until a runtime error occurs.
    - In case of a runtime error, a "Runtime error" is raised with the line number. Note that even if there is a runtime error, the output until that runtime error should be written into the `<output-file>`.
    - Not that the program should be parsed line-by-line in a sequential order. In case there are multiple runtime errors, the one in the earliest line should be reported.

A sample `<input-file>` is as follows:

```
Puroguramu o hajimeyo .
<stat>
<stat>
<stat>
...
Puroguramu o oware .
```

The program should start with `Puroguramu o hajimeyo .` and end with `Puroguramu o oware .` In between, there should be zero or more statements (`<stat>`). There can be no empty line. Trailing, leading empty lines are not allowed. Spaces at the end of the lines are not allowed.

`<stat>` is one of the following:

- `<var> wa <type> de aru .`
- `<var> no atai wa <expr> de aru .`
- `<expr> o print suru .`

where

`<var> wa <type> de aru .`	is used to declare a variable with the corresponding type. A variable can be declared only once. In other words, the same variable name cannot be declared multiple times.
`<var> no atai wa <expr> de aru .`	is used to assign the evaluated value of an expression to a variable.
`<expr> o print suru .`	is used to store the evaluated expression into the `<output-file>` file

`<var>` is the name of a variable. Variable name is **case-insensitive** and composed of only letters in the English alphabet, with a maximum length of 10 letters. Note that only variable names are case-insensitive. All other tokens, constants, etc are case-sensitive.

`<var>` must be declared before usage. `<var>` must be declared before its assignment. It must match the declared type.

`<type>` is one of the following:

- `seisu`
- `moji-retsu`

where `seisu` and `moji-retsu` correspond to integers and strings, respectively. `seisu` can be 10 digits max. `moji-retsu` can be 10,000 characters max.

The default value of an integer is 0 and a string is an empty string.

`<expr>` is one of the following:

- `<str-expr>`
- `<num-expr>`

where `<str-expr>` and `<num-expr>` correspond to string expression and numeric expression, respectively.

`<str-expr>` is one of the following:

- `<str-const>`
- `<var>`
- `<str-expr> tasu <str-expr>`
- `kaikakko <str-expr> tojikakko`
- `<num-expr> kakeru <str-expr>`

`tasu`, `kaikakko`, and `tojikakko` correspond to concatenation, open-paranthesis, and close-paranthesis. `kakeru` is used to replicate the `<str-expr>` `<num-expr>` times. `<var>` is the variable name of the string type.

Note that single-layer parentheses is allowed but no nesting.In other words, nested parentheses is not allowed.

Precedence rules: parentheses behave as if they have the highest precedence. `kakeru` has higher precedence than `tasu.` Otherwise, the string expressions are evaluated from right to left.

`<str-const>` is a sequence of characters, enclosed by the hyphen (`-`) character. It can contain max 10,000 characters. It does not contain the hyphen (`-`) character. The following are example valid `<str-const>`:

- `-hello world-`
- `-Hello ksjaksja-`
- `-kjskldja '>"|?|"?|"}&^*&^*%$7678326879-`

`<num-expr>` is one of the following:

- `<num-const>`
- `<var>`
- `<num-expr> tasu <num-expr>`
- `<num-expr> kakeru <num-expr>`
- `kaikakko <num-expr> tojikakko`

where `tasu`, `kakeru`, `kaikakko`, and `tojikakko` correspond to plus, times, open-paranthesis, and close-paranthesis. `<var>` is a variable name of integer type.

Precedence rules: parentheses behave as if they have the highest precedence. `kakeru` and `tasu` have the same precedence. The numeric expressions are evaluated from right to left.

Note that single-layer parentheses is allowed but no nesting. In other words, nested parentheses is not allowed.

`<num-const>` is a constant positive integer of max 10 digits. In standard English-language number formatting, the comma is used as a thousands separator. In our specific JPL number formatting, we will use commas to separate numbers into 4-digit groups, corresponding to 万 (10^4) and 億 (10^8) grouping:

- e.g.: `1,0000,0000` or `99,9995,2934` or `1123,0000` or `5,2934` or `3934` or `234` or `0`
- No plus (+) or minus (-) is allowed. Only digits with comma (,) separator are allowed.

Some compile error example cases:

- Any syntax error.
- If a variable is declared multiple times (remember that variable names are case insensitive).
- Using an undeclared variable.
- If a variable uses JPL keywords. These keywords are:
    - `wa`, `o`, `no`, `atai`, `de`, `aru`, `print`, `suru`, `tasu`, `seisu`, `moji-retsu`, `kakeru`, `kaikakko`, and `tojikakko`
    - `puroguramu`, `hajimeyo`, `oware`

- If the variable is not properly named (if non-English letters/characters are used or the variable names is longer than 10 characters).
- In case constant integer overflow (more than 10 digits).
- In case of constant string overflow (more than 10,000 chars).
- If a constant string contains a hyphen (−).
    - Note that any character except hyphen is allowed in strings.
- Commas in integer constants must follow the required 4-digit grouping rules shown below. Some more details:
    - Multiple commas allowed.
    - A number like 1,234 is invalid.
    - Commas should always appear at the correct 4-digit boundaries.
    - 0001 or 01 are not valid. (The number should not start with 0 unless its value is 0).
    - Numbers containing fewer than 4 digits must not have commas.
    - The number cannot end with a comma
- If a variable is given a value of the wrong type.
    - e.g. a variable is defined as an integer and is given a value of string type.
    - e.g. a variable is defined as a string and is given a value of integer type.
- There should be a single space in between each token in each line.
    - Tokens include `<stat>, <var>, <type>, <expr>, <str-expr>, <num-expr>, <str-const>, <num-expr>, <num-const>`
    - Tokens also include all JPL keywords.
    - Tokens also include dot (.) at the end of the sentences. Therefore, there should be a single space before dot (.). There should be no space after the dot.
    - Note that nested parentheses are not allowed. In other words, Any form where `kaikakko` appears inside another `kaikakko … tojikakko` is not allowed. For example:
        - `kaikakko x tasu y tojikakko kakeru 2` is allowed
        - `3 kakeru kaikakko x tasu y tojikakko` is allowed
        - `kaikakko x tasu y tojikakko kakeru kaikakko a tasu b tojikakko` is allowed
        - `kaikakko kaikakko x tasu y tojikakko tojikakko` is not allowed
- If there is zero space or more than one space character between tokens.
- If the expressions are not properly formed. For example:
    - If the parentheses do not match.
    - If the plus and times operators take incorrect types.
    - If there is a type error at any point in the expression.
- Follow the following rules:
    - str tasu str → ok
    - str tasu num → error
    - num tasu str → error
    - str kakeru num → error
    - str kakeru str → error
    - num kakeru str → ok

Runtime Errors in case:

- If an integer is generated more than 10 digits during runtime.
- If a string is generated more than 10,000 characters during runtime.

Output:

Compile step:

- Write output to file `<output-file>`. Each print statement should print-out the corresponding integer or string in a single line. Output should not include trailing spaces.
- If there is a compile error, no `<obj-file>` should be generated. Your program should raise an exception and quit using the following code:
    - `raise Exception(f"Compile error {line_no=}")`
- Unless there is a compile error, `<obj-file>` should be generated.

Execute step:

- You need to read from `<obj-file>` and execute the program. You can assume `<obj-file>` should be there. All operations (including print statements) should be executed until
    - The end of the program, or
    - A runtime error occurs.

    - In case runtime error occurs, your program should raise an exception with
        - `raise Exception(f"Runtime error {line_no=}")`

Some Examples:

Assume you have the following input.jpl in the current directory:

```
Puroguramu o hajimeyo .
x wa seisu de aru .
x no atai wa 7,9519,8784 de aru .
Puroguramu o oware .
```

Running your hw5.jpl with the following command line arguments:

`-compile` `input.jpl` `hw5.obj`

should generate `hw5.obj`

Then running your hw5.jpl with the following command line arguments:

`-execute` `hw5.obj` `output.txt`

should generate an empty `output.txt` file:

```

```

Now we add a print statement.

Assume you have the following input.jpl in the current directory:

```
Puroguramu o hajimeyo .
x wa seisu de aru .
x no atai wa 7,9519,8784 de aru .
x tasu 1 o print suru .
Puroguramu o oware .
```

Running your hw5.jpl with the following command line arguments:

`-compile` `input.jpl` `hw5.obj`

should generate `hw5.obj`

Then running your hw5.jpl with the following command line arguments:

`-execute` `hw5.obj` `output.txt`

should generate the following `output.txt`

```
7,9519,8784
```

Now, a new example with an additional line in the input file:

Assume you have the following input.jpl in the current directory:

```
Puroguramu o hajimeyo .
x wa seisu de aru .
x no atai wa 7,9519,8784 de aru .
x tasu 1 o print suru .
x tasu x o print suru .
Puroguramu o oware .
```

Running your hw5.jpl with the following command line arguments:

`-compile` `input.jpl` `hw5.obj`

should generate `hw5.obj`

Then running your hw5.jpl with the following command line arguments:

`-execute` `hw5.obj` `output.txt`

should generate the following `output.txt`

```
7,9519,8784
```

and also `raise Exception("Runtime error line_no=5")`

Now let us have a syntax error:

Assume you have the following input.jpl in the current directory:

```
Puroguramu o hajimeyo .
x waa seisu de aru .
x no atai wa 7,9519,8784 de aru .
x tasu 1 o print suru .
x tasu x o print suru .
Puroguramu o oware .
```

Running your hw5.jpl with the following command line arguments:

`-compile` `input.jpl` `hw5.obj`

should not generate `hw5.obj`. It should `raise Exception("Compile error line_no=2")`

Reminders:

(1) Parentheses Rule:

Only *single-level* parentheses are allowed, but they may appear multiple times in an expression, and they can contain any valid expr — but no pair may appear inside another pair.

✓ allowed:

```css
( a tasu b ) kakeru c
a tasu ( b kakeru c ) tasu d
( a tasu b ) tasu ( c kakeru d )
kaikakko a tasu b tojikakko tasu kaika…toji...
```

✕ forbidden:

```css
( a tasu ( b tasu c ) )
```

(2) Parentheses treated as highest precedence:

Evaluation order:

1 → expressions inside parentheses

2 → kakeru (right-to-left)

3 → tasu (right-to-left)

(3) Expression types inside parentheses:

Parentheses can contain *any valid expr* (string or numeric), but they must conform to the type of the surrounding context.

For example:

```nginx
x wa seisu de aru .
x no atai wa kaikakko 5 tasu 3 tojikakko de aru .
```

✓ ok

```nginx
x wa seisu de aru .
x no atai wa kaikakko -hello- tasu -world- tojikakko de aru .
```

✕ compile error (string expr inside numeric assignment)

(4) Mixed-type operations:

Following your rules:

- str tasu str → OK
- str tasu num → compile error
- num tasu str → compile error
- str kakeru num → compile error
- str kakeru str → compile error
- num kakeru str → OK (string replication)
- num kakeru num → OK
- str kakeru num → compile error

(5) Numeric evaluation right-to-left:

```css
a tasu b tasu c
```

= a + (b + c)

```css
a kakeru b tasu c kakeru d
```

= a * (b + (c * d))

(6) String evaluation right-to-left (with precedence):

```r
a tasu b tasu c  =>  a tasu (b tasu c)
a kakeru b tasu c  => a kakeru b  tasu c
```