Group Project: A Command-Line-Based Interpreter for SIMPLE Programs

Deadlines

1. Team up on Blackboard/Groups before 23:59, September 23, 2022. Afterwards, we will randomly team up students with no group.
2. Archive the source code, the tests, the presentation slides, and the final report in PDF into a ZIP file, and submit it on Blackboard before 23:59, November 19, 2022.
3. Submit the design review report on Blackboard before 23:59, November 25, 2022.

1 Overview

The goal of this project is to develop a command-line-based interpreter for programs written in the SIMPLE programming language. SIMPLE is a programming language similar to Java, but it supports a very limited number of data types, operators, and statement types, and the interpreter will enable users to define and interpret SIMPLE programs.

In a nutshell, SIMPLE supports only two data types, namely `bool` and `int`: Values of type *bool* include only `true` and `false`, while values of type *int* range between `-99999` and `99999`, both inclusive. That is, calculation results of *int* type that are larger than 99999 or smaller than -99999 are always rounded to 99999 and -99999, respectively. In total, 15 different operators, namely `+`, `-`, `*`, `/`, `#`, `~`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `&&`, `||`, and `!`, and seven types of statements, namely variable definitions, skips, assignments, conditionals, loops, prints, and blocks, are supported in SIMPLE; Each statement in a SIMPLE program is identified by a unique label, while each variable or expression is identified by a unique name.

2 Requirements

The interpreter should provide the following capabilities:

[REQ1] (1.5 points) It should support defining variables.
Command: `vardef` *lab typ varName expRef*
Effect: Defines a new variable definition statement with a unique label *lab*. When this variable definition statement is executed, a variable with the unique name *varName* and of type *typ* is defined and initialized to the value of the expression reference *expRef*, where *typ* is either `bool` or `int`. An expression reference is either a literal, a variable name, or an expression name.
Example: `vardef vardef1 x int 100`
Note: All variable/expression names and statement labels in SIMPLE programs are identifiers, which 1) may contain only English letters and digits, 2) cannot start with digits, and 3) may contain at most eight characters; `int`, `bool`, `true`, `false`, and all command names (e.g., `vardef` and `execute`) are SIMPLE keywords, and they cannot be used as variable/expression names or statement labels.

[REQ2] (1.5 points) It should support defining binary expressions.
Command: `binexpr` *expName expRef1 bop expRef2*
Effect: Defines a new binary expression with a unique name *expName*. The new expression's left operand is an expression reference *expRef1*, its binary operator is *bop*, and its right operand is an expression reference *expRef2*. The following ten binary operators can be applied to `int` expression references: `+`, `-`, `*`, `/`, `>`, `>=`, `<`, `<=`, `==`, `!=`; The following four binary operators can be applied to `bool` expression references: `&&`, `||`, `==`, and `!=`.
Example: `binexpr exp1 x * 20`

[REQ3] (1.5 points) It should support defining unary expressions.
Command: `unexpr` *expName uop expRef1*
Effect: Defines a new unary expression with a unique name *expName*. The new expression's unary operator is *uop*, and its operand is an expression reference *expRef1*. The following two unary operators can be applied to *int* values: `#` and `~` (equivalent to unary `+` and `-` in Java). The following one unary operator can be applied to *bool* values: `!`.
Example: `unexpr exp2 ~ exp1`

[REQ4] (1.5 points) It should support defining assignment statements.
Command: `assign` *lab varName expRef*
Effect: Defines a new assignment statement with a unique label *lab*. When this assignment statement is executed, the value of expression reference *expRef* will be assigned to the variable with name *varName*. The variable should have been defined by a `vardef` statement.
Example: `assign assign1 x exp2`

[REQ5] (1.5 points) It should support defining print statements.
Command: `print` *lab expRef*
Effect: Defines a new print statement with a unique label *lab*. When this print statement is executed, the value of the expression reference *expRef* will be printed between a pair of square brackets.
Example: `print print1 exp2`

[REQ6] (1.5 points) It should support defining skip statements.
Command: `skip` *lab*
Effect: Defines a new skip statement with a unique label *lab*. When executed, a skip statement does nothing.
Example: `skip skip1`

[REQ7] (1.5 points) It should support defining block statements.
Command: `block` *lab statementLab1 ... statementLabn*
Effect: Defines a new block statement with a unique label *lab*. The block statement contains a list of statements with labels *statementLab1*, ..., *statementLabn* ($n > 0$). When this block statement is executed, the statements in the block statement are executed in sequence.
Example: `block block1 assign1 skip1`

[REQ8] (1.5 points) It should support defining conditional statements.
Command: `if` *lab expRef statementLab1 statementLab2*
Effect: Defines a new conditional statement with a unique label *lab*. When this conditional statement is executed, the statement with label *statementLab1* is executed if the expression reference *expRef* evaluates to `true`; Otherwise, the statement with label *statementLab2* is executed.
Example: `if if1 exp5 block1 print1`

[REQ9] (1.5 points) It should support defining loop statements.
Command: `while` *lab expRef statementLab1*
Effect: Defines a new loop statement with a unique label *lab*. When this loop statement is executed, the value of the expression reference *expRef* is evaluated repeatedly: If it evaluates to `true`, the statement with label *statementLab1* is executed; Otherwise, the loop terminates.
Example: `while while1 true block1` (note that the statement with label `block1` will be repeatedly executed for ever when this while statement is executed)

[REQ10] (1.5 points) It should support defining SIMPLE programs.
Command: `program` *programName statementLab*
Effect: Defines a new SIMPLE program with a unique name *programName*, and the program has the statement labeled *statementLab* as its body. When the program

is executed, the statement in its body is executed accordingly.
Example: `program program1 while1`

[REQ11] (15 points) It should support executing a defined SIMPLE program.
Command: `execute` *programName*
Effect: Executes the defined program named *programName*.
Example: `execute program1`

[REQ12] (3 points) It should support listing a defined SIMPLE program to the console.
Command: `list` *programName*
Effect: Prints out the list of commands that defines the program with name *programName*.
Example: `list program1`

[REQ13] (3 points) It should support storing a defined SIMPLE program into a file.
Command: `store` *programName path*
Effect: Stores the defined program with name *programName* into the file at *path*.
Example: `store program1 d:\prog1.simple`

[REQ14] (3 points) It should support loading a defined SIMPLE program from a file.
Command: `load` *path programName*
Effect: Loads the defined program from *path* and names it as PROGRAMNAME.
Example: `load d:\prog1.simple program1`

[REQ15] (1 points) The user should be able to terminate the current execution of the interpreter.
Command: `quit`
Effect: Terminates the execution of the interpreter.

The interpreter may be extended with the following *bonus* features:

[BON1] (10 points) Support for three commands to debug SIMPLE programs.

- **Command:** `debug` *programName*
  Effect: Starts/Continues executing the program named *programName* in debug mode till completion or a breakpoint.
  Example: `debug program1`
- **Command:** `togglebreakpoint` *programName statementLab*
  Effect: Sets/Removes a breakpoint at the statement labeled *statementLab* inside the program with name *programName*. During debugging, the execution of the program will suspend when encountering the breakpoint, i.e., the statement with label *statementLab* becomes the next statement to execute.
  Example: `togglebreakpoint program1 block1`
- **Command:** `inspect` *programName varName*
  Effect: When the execution of program named *programName* is suspended during debugging, this command can be invoked to print out the value of the variable named *varName* within a pair of angle brackets, if the variable is in scope.
  Example: `inspect program1 x`

[BON2] (5 points) Support for instrumenting[1] SIMPLE programs.
Command: `instrument` *programName statementLab pos expRef*
Effect: Prints out the value of *expRef* in a pair of curly braces right before/after the statement with label *statementLab* from the program named *programName* is executed, where *pos* should be either *before* or *after*. Note that multiple instrument commands can be used to print out multiple strings at a single statement location.
Example: `instrument program1 block1 after 5` will cause the interpreter to

[1] https://www.youtube.com/watch?v=VaZrulbL6Kc

print out "{5}" *after* each time the statement labeled *block1* from program named *program1* is executed.

The requirements above have been left incomplete on purpose, and you need to decide on the missing details when designing and implementing the interpreter. For example, you need to use your best judgment to gracefully handle situations where a command is in bad format or the operation required by a command cannot be performed successfully, e.g., because a name is already used or not defined yet. Poor design in handling those situations will lead to point deductions.

Hint: When interpreting a SIMPLE program, or the statements contained in the program, the interpreter needs to keep track of the values of the variables defined in the program. For that purpose, the interpreter needs to 1) maintain a data structure to associate variables with their values and 2) update those values based on the effects of the statements when they are executed.

3 An Example Interaction Session with the Interpreter

A short, but complete, example interaction session with the interpreter is shown in Figure ?? to give you a concrete idea about how the interpreter is typically used. In the figure,

1) the leading `>` symbol on each line is the prompt character of the interpreter,
2) `<enter>` indicates the `Enter` key that the user presses to complete a line of input, and
3) the comments start with '@'.

Note that inputting a command will only cause the expression or statement being defined. The expressions and statements defined in a program will not be interpreted/executed until the execution of the program is triggered by an invocation to the `execute` command. For example, inputting command `vardef vardef1 int x 0` will only cause a variable definition statement to be defined. The statement is not interpreted until the command `execute printeven` is invoked.

```
>vardef vardef1 int x 0<enter>                                    @ vardef1: int x = 0;
>b-expr exp1 x % 2<enter>                                                @ exp1: x % 2
>b-expr exp2 exp1 == 0<enter>                                      @ exp2: x % 2 == 0
>print print1 x<enter>                                              @ print1: print x
>skip skip1<enter>                                                      @ skip1: skip
>if if1 exp2 print1 skip1<enter>                @ if1: if(x%2==0) then prt1 else skp1
>b-expr exp3 x + 1<enter>                                               @ exp3: x + 1
>assign assign1 x exp3<enter>                                     @ assign1: x = x + 1;
>block block1 if1 assign1<enter>                                @ block1: {if1 assign1}
>b-expr exp4 x <= 10<enter>                                            @ exp4: x <= 10
>while while1 exp4 block1<enter>                  @ while1: while(x<=10){if1 assign1}
>block block2 vardef1 while1<enter>   @ block2: int x = 0; while(x<=10){if1 assign1}
>program printeven block2                                          @ printeven: block2
>execute printeven          @ executes the program, printing out [0][2][4][6][8][10]
```

Figure 1: Defining and executing a SIMPLE program `printeven` that prints out even numbers between 0 and 10, both inclusive.

The SIMPLE program in Figure ?? can be rewritten into the program shown in Figure 2 using a syntax similar to Java.

```
int x = 0;
while(x <= 10){
  if(x % 2 == 0) then
    print x;
  else
    skip;
  x = x + 1;
}
```

Figure 2: Program `printeven` from Figure ?? written in a Java-like syntax.

Given the program `printeven` defined in Figure ??, Figure 3 demonstrates how the program can be debugged using corresponding commands. Moreover, if command "`instrument printeven`

`assign1 after x`" is invoked before the first `debug` command, the debugging process will also print out the value of `x` each time it is increased, producing an output "`{1}`", "`{2}`", ..., "`{11}`".

```
>...<enter>
>program printeven block2<enter>                                    @ printeven: block2
>togglebreakpoint printeven if1<enter>                    @ sets a breakpoint at if1
>debug printeven<enter>           @ executes the program in debugging mode till if1
>inspect printeven x<enter>                                          @ prints out "<0>"
>debug printeven<enter>         @ continues the execution in debugging mode till if1
>inspect printeven x<enter>                                          @ prints out "<1>"
>togglebreakpoint printeven if1<enter>               @ removes the breakpoint at if1
>debug printeven<enter>     @ executes the program in debugging mode till completion
```

Figure 3: Debugging program `printeven`.

4 Group Forming

This is a group project. Each group may have 3 to 4 members. Please team up on Blackboard/Groups before 23:59, September 23, 2022. Afterward, students with no group will be randomly grouped, and requests for group change are only approved if written agreements from all members of the involved groups are provided before or on September 7, 2022. No group change will be allowed after September 7, 2022.

5 Code Inspection

The inspection tool of IntelliJ IDEA[2] checks your program to identify potential problems in the source code. We have prepared a set of rules to be used in grading (`COMP2021_PROJECT.xml` in the project material package). Import the rules into your IntelliJ IDEA IDE and check your implementation against the rules. Note that unit tests do not need to be checked against these rules.

The inspection tool is able to check for many potential problems, but only a few of them are considered errors by the provided rule set. 2 points will be deducted for each *error* in your code reported by the tool.

6 Line Coverage of Unit Tests

Line coverage[3] is a measure used in software testing to report the percentage of source code lines that have been tested: The higher the coverage, the more thoroughly we have tested a program.

You should follow the Model-View-Controller (MVC) pattern[4] in designing the interpreter. Put all Java classes for the model into package `hk.edu.polyu.comp.comp2021.simple.model` (see the structure in the sample project) and *write tests for the model classes*.

During grading, we will use the coverage tool of IntelliJ IDEA[5] to get the line coverage of your tests on package `hk.edu.polyu.comp.comp2021.simple.model`. Your will get 10 base points for a line coverage between 90% and 100%, 9 base points for a coverage between 80% and 89%, and so on. You will get 0 base points for a line coverage below 30%. The final points you get for line coverage of unit tests will be calculated as your base points multiplied by the percentage of your requirements coverage. For example, if you only implement 60% of the requirements and you achieve 95% line coverage in testing, then you will get 6 = 10 * 60% points for this part.

7 Design Review

The 3-hour lecture time in Week 13 will be devoted to the peer review of your design. The review will be conducted in clusters of X groups (X is to be decided later); Each group needs to review the designs of the other groups in its cluster and fill out the review reports. In a review

[2] https://www.jetbrains.com/help/idea/code-inspection.html
[3] http://en.wikipedia.org/wiki/Code_coverage
[4] https://en.wikipedia.org/wiki/Model-view-controller
[5] https://www.jetbrains.com/help/idea/code-coverage.html

report, a group needs to point out which design choices from another group are good, which are questionable, which should have been avoided, and then explain the reasons.

Details about the organization of the design review and the format of the review report will be announced before the review session.

8 Project Grading

You can earn at most 100 points in total in the project from the following components:

- Requirements coverage (in total 40 points, as listed in Section 2)
- Code quality (10 points for good object-oriented design and 10 for good code style, as reported by the code inspection tool)
- Line coverage of unit tests (10 points)
- Presentation (7 points)
- Design review report (8 points)
- Final report (15 points)
- Bonus points (15 points)

  Note: Bonus points can be used to reach, but not exceed, 100 points.

Individual grading: In the project, each group member is expected to actively participate and make fair contributions. In general, members of a group will receive the same grade for what their group produces at the end. Individual grading, however, could be arranged if any member thinks "one grade for all" is unfair and files a request to the instructor in writing (e.g., via email). In individual grading, the grade received by each group member will be based on his/her own contributions to the project, and each member will need to provide evidence for his/her contributions to facilitate the grading.

9 General Notes

- Java SE Development Kit Version 17[6] and IntelliJ IDEA Community Edition Version 2022.2.1[7] will be used in grading your project. Make sure you use the same versions of tools for your development. Note that you need to set "File|Project Structure..." in IntelliJ IDEA as the following:
  - Project SDK: JDK 17
  - Project language level: 8 - Lambdas, type annotations etc.
- Your code should not rely on any library that is not part of the standard Java SE 17 API.
- The project material package also include three other files:
  - `SampleProject.zip`: Provides an example for the structure of the project directory. Feel free to build the interpreter based on the sample project.
  - `IntelliJ IDEA Tutorial.pdf`: Briefly explains how certain tasks relevant to the project can be accomplished in IntelliJ IDEA.
  - `COMP2021_PROJECT.xml`: Contains the rules to be used in code inspection.

If you use other IDEs for your development, make sure all your classes and tests are put into an IntelliJ IDEA project that is ready to be tested and executed. 50 points will be deducted from projects not satisfying this requirement or with compilation errors!

[6] https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html
[7] https://www.jetbrains.com/idea/download/other.html

10 Project Report Template

Project Report
Group XYZ
COMP2021 Object-Oriented Programming (Fall 2022)
Member1
Member2
Member3

1 Introduction

This document describes the design and implementation of an interpreter for SIMPLE programs by group XYZ. The project is part of the course COMP2021 Object-Oriented Programming at PolyU.

2 A Command-Line-Based Interpreter for Simple Programs

In this section, we describe first the overall design and then the implementation details of the interpreter.

2.1 Design

Use class diagram(s) to give an overview of the system design and describe in general terms how different components fit together. Feel free to elaborate on design patterns used and/or anything else that might help others understand your design.

2.2 Requirements

For each (compulsory and bonus) requirement, describe 1) whether it is implemented and, when yes, 2) how you implemented the requirement as well as 3) how you handled various error conditions.

[REQ1] 1) The requirement is implemented.
2) Implementation details.
3) Error conditions and how they are handled.

[REQ2] 1) The requirement is not implemented.

[REQ3] ...

3 User Manual

In this section, we explain how the interpreter works from a user's perspective. *Describe here all the commands that the system supports. For each command, use screenshots to show the results under different inputs.*