December 5, 2016
Final Project - RUDI Interpreter
Computer Science Principles for Practicing Engineers
CMU SE 17630-D
Lattanze/Rosso-Llopart

Project Objective

The task for this project is to build an interpreter which acts as a line by line processor for the programming language RUDI. The interpreter should read from a specified input file which is written in the RUDI language and execute the code. The syntax for RUDI is specified in the CSPE RUDI Interpreter course pdf.

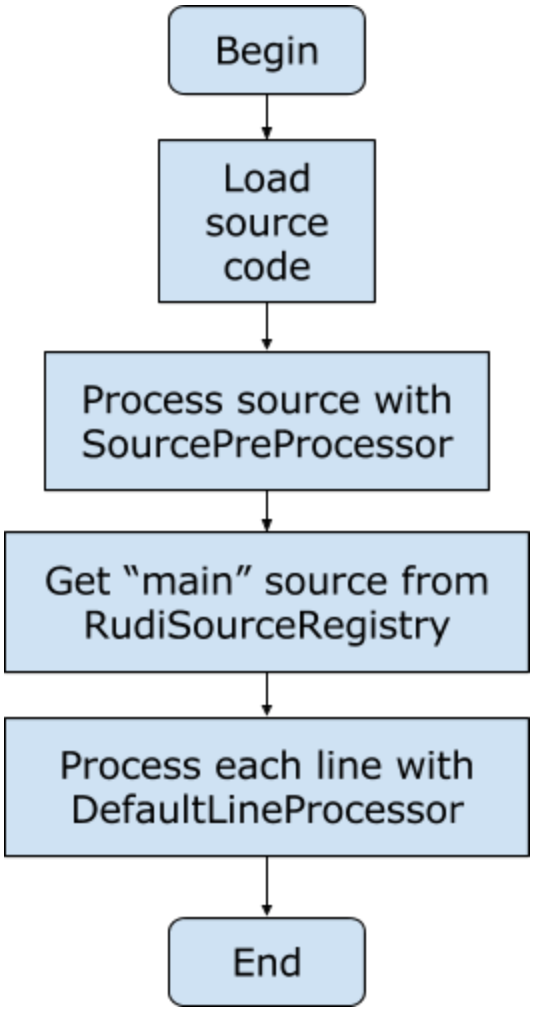
Assumptions

1. All opening brackets '[' and ending brackets ']' must always occupy their own line (except when contained within strings).
2. Only one set of brackets may occur within the DECS section of the program. Any other occurrences of brackets can only occur between the BEGIN and END statements of the program or between the BEGIN and RETURN statements of a subroutine declaration.
3. Variables and subroutines cannot be declared with the same names as any reserved words. Reserved words include any RUDI statements or constants (e.g. PROGRAM, WHILE, or CR) as well as the key word MAIN which is inherent to the operation of the source code of the interpreter.
4. Tokens within any expression *must* be space-delimited. For example (1 :ne: 3) is legal syntax, however (1 :ne: 3) would be illegal because the '(' and '1' tokens are not space-delimited. (1 :ne:3) would likewise be illegal because the tokens ':ne:' and '3' are not space-delimited.
5. The interpreter supports arithmetic between float and integer types, however a variable which has been declared as an integer cannot be reassigned to a float type as a result of such arithmetic during program execution.
6. Subroutines do not have to be declared as being functions of any existing variables declared within the primary DECS section of the program. For example, for subroutine declaration foo(a), a user can call foo(x) during program execution and any previously declared variable x will simply be passed to the foo() subroutine with the new label 'a' within that subroutine's scope. Any modifications to 'a' which occur within foo() will also modify variable 'x' when returning to the main body of the program.

Interpreter Design

The overall structure of our interpreter is as follows. The application will first load the source code from the file into an original "raw" RudiSource class object (a RudiSource object

stores an array of strings, where each string is a line of source code. The Rudisource also stores a global offset integer, which tells it which line of the original source code its first string element originates from). From there, the raw RudiSource is sent to the SourcePreProcessor line processor. First the SourcePreProcessor parses the lines and strips out all comments, then concatenates individual lines (post-stripping) back together. The processor then makes sure that all brackets are matched within the code by counting '[' and ']' characters that exist and making sure there count is equal at the end of the code. Finally, the processor parses the raw source code into multiple RudiSource's which are stored in a registry called the RudiSourceRegistry. The RudiSourceRegistry is a hash map that maps RudiSources to various key functions. The key for the original source code is called the rawSource, and the key for the main body of the program is called Main. Any subroutines declarations found inside of the original code result in the creation of RudiSources with keys with the same name as the subroutine declaration. Each RudiSource stores and initializes its global offset once it is created, while the rawSource's global offset remains uninitialized. Once the SourcePreProcessor is finished, it grabs the Main RudiSource from the RudiSourceRegistry, and sends each line one by one through the DefaultLineProcessor, which will evaluate each line and determine what to do. Here is a flowchart for the entire process of the interpreter:



The DefaultLineProcessor holds the majority of the program's logic. It follows a chain-of-responsibility design pattern by holding a list of line processors which each achieve a separate task. Each line of code will go through the chain until it finds a line processor that can process it. This promotes loose coupling of our system, and supports the open-closed principle of object oriented programming, allowing us to extend the system to handle new commands by adding a new LineProcessor without needing to modify the existing system.

While each line is being processed, the line processors have access to a RudiContext object which is created for each RudiSource (For all other RudiSources besides Main, the corresponding RudiContext is created at the same time as the initialization of the RudiSource. For the RudiContext initalization for the Main RudiSource, see the Program line processor details outlined below), which holds the state of execution of the code as well as any variables or parameters which have been initialized within the RudiSource's scope. This includes the current processing mode (declaration, execution, skip, or comment), parent context (if one exists), the RudiSource being processed, a hash map registrar for variables and parameters, and the current bracket depth. The RudiContext also stores control flow information, including control types such as WHILE, IF, and ELSE expressions. The RudiContext is used by the line processors to determine if a given line can be processed by any individual processor, such as the DecsLineProcessor which only operates while declaring new variables in declaration mode. The variable and parameter registrar VariableRegistrar allows line processors to access and modify variables as needed during the program execution.

The RudiContext can be accessed through a singleton instance of the RudiStack ADT. This is a stack of RudiContexts. A new RudiContext is pushed onto the stack when processing if-then structures, when loops, and subroutine calls. This was done so the structures can contain their own state, which allows for things like nested if's and subroutine calls. This also allows for a separation of scope for variables between parent and child contexts. RudiContexts are always processed from the stack in a LIFO order. Because each RudiSource contains its global offset from the original code error reporting can be done accurately by adding the relative line of any errors to the source's global offset if it exists. This line number is then also referenced in the rawSource RudiSource in order to find the original line of code to print to the user. When a statement is finished being processed, it is popped from the stack and the line processors resume where they last left off on the previous RudiContext if it exists.

A major component used by many line processors is the ExpressionResolver. The ExpressionResolver evaluates expressions by converting the infix expressions to postfix through an implementation of the shunting-yard algorithm, and then runs through the postfix algorithm in order to get a final value that represents the expression. This is used during variable assignments, print, if, and while statements, and subroutine calls. The ExpressionResolve relies on all tokens within expressions being space-delimited in order to be able to evaluate separate tokens, and uses a hash map in order to convert from RUDI comparison and arithmetic operators to their corresponding functions in Java.

Line Processors

All line processors have two functions, canProcess() and doProcess(). canProcess() checks the current RudiContext to see if it should attempt to process the current line, while the doProcess() function contains the processing instructions and is executed if canProcess() returns a “true” boolean. Below is a list of all implemented line processors in their order of execution, and their corresponding purpose:

EmptyLine - This processor checks to see if the current line is empty. If it is the line processor moves to the next line of the RudiSource and increments the line count.
Program - This processor checks to see if the PROGRAM reserved word is present. If so the program is in the Main RudiSource, and a corresponding RudiContext is created and pushed onto the call stack to be used by the other line processors.
Subroutine - This processor checks to see if the current line is a subroutine declaration. If it is then the current RudiContext being processed is in the first line of a subroutine call, and the line is skipped.
Decs - This processor checks for the DECS reserved word, and if it is found changes the RudiContext to declaration mode.
StartingBracket - This processor checks for the ‘[‘ character. If one is found it increments the RudiContext bracket depth.
EndingBracket - This processor checks for the ‘]‘ character. If one is found it decrements the RudiContext bracket depth.
SkipLine - This processor checks to see if the RudiContext specifies that the lines are within an IfThenElse statement or While loop. If they are the processor parses the lines into a true branch for the If statement and a false branch for the Else statement. The false branch contains an empty array of strings if the Else statement is absent or a While statement is being evaluated. The conditional of the statement is then evaluated, and the corresponding branch is used to create a RudiSource and RudiContext which are pushed onto the stack. In the case of While statements this process is repeated until the conditional evaluates to false.
IfThenElse - This processor changes the RudiContext to notify the SkipLine processor that it is within an If statement.
While - This processor changes the RudiContext to notify the SkipLine processor that it is within a While statement.
VariableDeclaration - This processor checks to see if the RudiContext is in declaration mode. If so it reads lines and initializes each variable as integer, float, or string and stores them in the VariableRegistrar.
Begin - This processor checks to see if the BEGIN reserved word is present. If it is it changes the RudiContext to execution mode.
End - This processor checks to see if the END reserved word is present. If it is it checks to see if the current RudiSource is the Main RudiSource, and if it is it ends the program. Otherwise it throws an error.
Return - This processor checks to see if the Return reserved word is present. If it is it checks to

see if the current RudiSource is a subroutine RudiSource, and if it is it ends the processing of the current RudiContext. Otherwise it throws an error.
Print - This processor checks to see if the PRINT reserved word is present. If it is it attempts to print the following stringm, expression, or variable (by checking to see if it is mapped to the VariableRegistar) and if it cannot it throws an error.
Input - This processor checks to see if the INPUT reserved word is present. If it is it checks to see that a declared variable follows the reserved word, and then prompts the user for input to store within the variable.
CallSubroutine - This processor checks to see if the current line is a call to a subroutine. If it is the corresponding RudiSource/RudiContext are placed onto the stack or an error is thrown if the key for the subroutine does not exist. This processor also places any variables of the subroutine as parameters to the RudiContext as it is placed onto the stack.
VariableAssignment - This processor checks text on the line to see if it is in the format of a variable assignment If it is it checks the text to the left of the '=' character against the VariableRegistrar to see if it exists. If it does exist in the registrar it attempts to assign the expression to the right of the '=' character to the variable accordingly, and throws an error if it cannot.
Stop - This processor checks to see if the STOP reserved word is present. If it is the program is terminated immediately.

ADTs

This interpreter utilizes hash maps, stacks, arrays, and a linked-list queue.
Hash maps are used to store RudiSources and their corresponding RudiContexts into the RudiSourceRegistry. Other hash maps are also used to store variable and parameter values inside of the VariableRegistrars for each RudiContext. Stacks are used both the postfix expression solver of the ExpressionResolver and the RudiStack. Arrays of strings are used in each RudiSource to store the lines of code and in the portion of the ExpressionResolver which tokenizes a given expression in order to store the tokens.

A queue is used in the ExpressionResolver which parses the expression tokens with the shunting-yard algorithm. A queue is needed since we need to preserve the order of the items popped off the operator stack before pushing them onto the output stack. A linked-list implementation of a queue is used since the actual number of items to be stored in the queue is unknown. This implementation also gives an (O(1)) time complexity for its enqueue and dequeue operations as an added benefit.

The line processors are kept within an array of LineProcessor objects stored inside the DelegatingLineProcessor. The number of processors we are going to configure is known, so the array provides the advantage of an O(1) lookup time. Also, because the number of processors

to configure isn't huge, allocating a contiguous chunk of memory isn't a concern.

Program Running Instructions

To use the interpreter, you must have Java 1.8 installed. The provided zip file contains a run directory with scripts. Here are instructions/examples for running the interpreter on Windows and OSX operating systems:

Windows

- From the run directory, execute: rudi file.rud
- To run from any directory, add the run directory to your path
    - set PATH=%PATH%<PROJECT_FOLDER>\run

OSX

- From the run directory, execute: ./rudi file.rud
    - To avoid ./, add the current directory to your path
        - export PATH=$PATH:.
- To run from any directory, add the run directory to your path
    - export PATH=$PATH:<PROJECT_FOLDER>/run