Team2(Mengye Gong, Ziyuan Song)

Rudi Interpreter

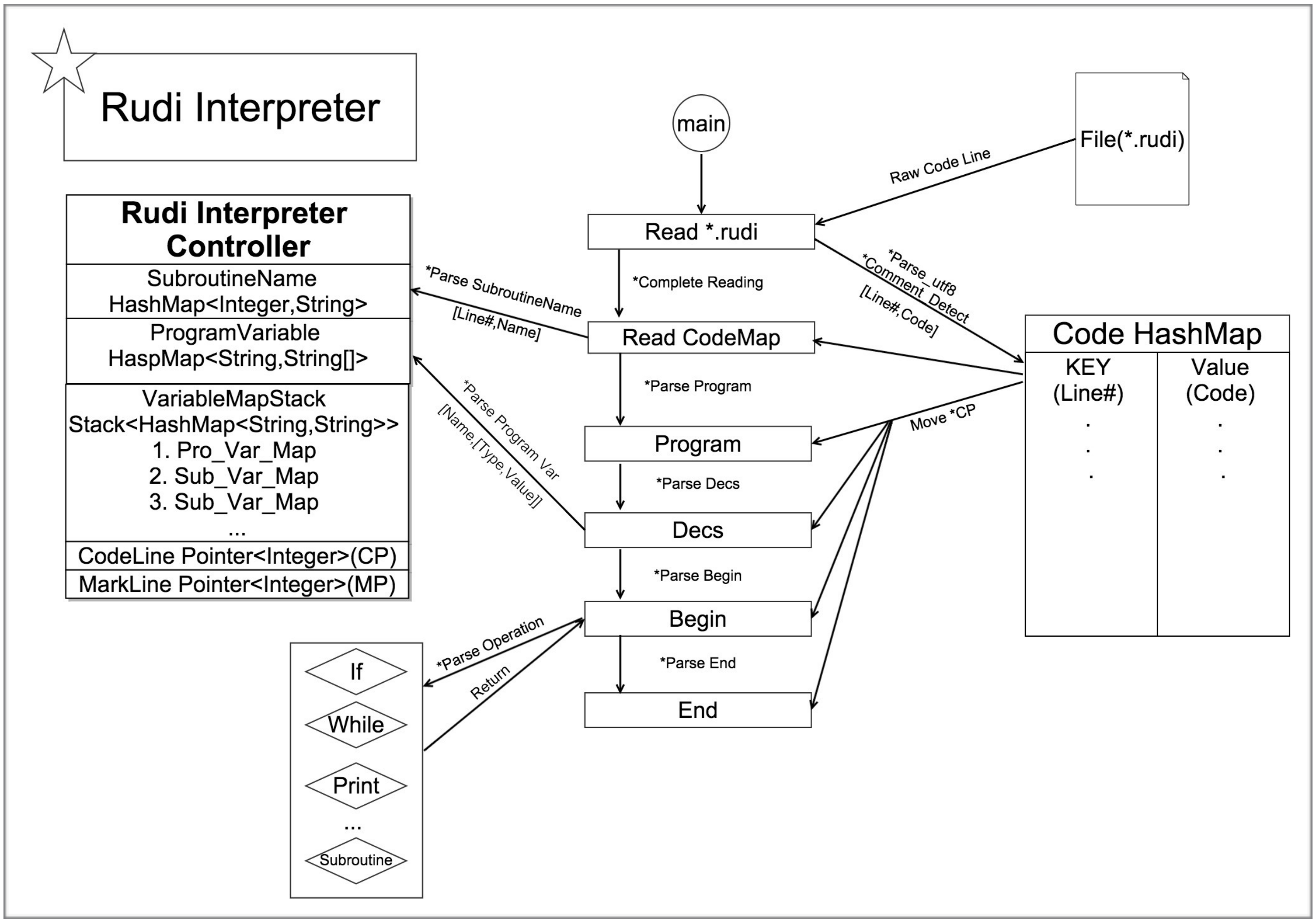
1. Overall Design

The Overall Design of the system has been almost shown in the picture below. There are three main parts in this interpreter. Rudi Interpreter Controller(RIC), Execution Function and Expression Operation. The former two parts can be seen in this structured picture and the last one is a background algorithm about expression calculations and comparisons.

1.1. *Rudi Interpreter Controller*

RIC is the core part of this rudi interpreter. It is designed to manage and control the interpreter system. There are two HashMaps in RIC. The first one is Subroutine Name HashMap(SNHM)used to store all subroutine names and their lines. The other is Program Variable HashMap(PVHM) used to store variable names, types and values. The interpreter only has possess this two HashMaps. The subroutine variable HashMap will be stored and managed by each subroutine objects. CodeLine Pointer(CP) is a pointer to record a line number of code that is being executed or will be. Its movement is implemented by calling RIC's inner interface function. MarkLine Pointer(MP) is another pointer for system to keep a line number that may be used when code executing would like return to this code line.

For the consideration of function nested calling, variable work ranges may be a tough issue. To resolve that subroutines can only use references and variable defined in its declarations and to manage variables conveniently. This system employed a Variable Map Stack(VMS) structure to store the PVHM and variable HashMaps in subroutine. When a subroutine is called, a subroutine object is created, then its variable HashMap will be pushed to VMS. Because PVHM is always be pushed to stacked and never popped until the program process ends, a variable HashMap belonging to an executing subroutine is always on the peak of the VMS. And before a subroutine returns, its variable HashMap will be popped from the peak of the VMS. This stack process is the most essential part of implementations of subroutine nested.



1.2. *Execution Function*

RIC manages the interpreting process and support most data storage structure as a whole. From an executing aspect, there is only single thread main program used in this interpreter as shown in the middle of the upper picture.

After, a main program gives an interface for user to input .rudi file, the interpreter begins to read rudi file from a appointed path. Firstly, the interpreter read codes into a Code HashMap(CHM). Also code execution begins from reading the CHM. The put code in CHM must be parsed by utf-8 format approval and any comments will never be put in CHM. Secondly, interpreter begin to parse each line of code and check if the structure of code is correct. For example, check if there is program and subroutine in this file. And this first traverse will read subroutine name into SNHM if there is any. After parsing the rudi code about structure's correction, it begins to parse and execute codes in the program.

In the decs block, variables will be read and stored in PVH. The variable name is the key. In the begin block, through parsing different keywords to call several functions, like "if then-else", "while", "print", "input", "stop", "assign" and "subroutine call". The program will continue to read and execute until an END comes up.

During these process operations, there is an Operations class to manage static functions like "print", "input", "stop", "assign" and an ComplexFxn class to create objects "if", "while" and "subroutines". The former is set to be static due to its single line execution and the latter is arrange to create objects because there are many loops, ranging validations and condition judging should be taken into consideration. And their attributes should be kept when there is a nested call happening. So the blending of objects and static functions make the interpreter more flexible and efficient.

1.3. *Expression Operation*

When deal with the expression, the first thing we should do is to check whether the string is a valid expression.

First, we should check whether brackets in the expression is in pairs. Single bracket is invalid. If this check is passed. We pick the string in the first pair for more check. This string could have several numbers, characters and operators. First, check whether there is any "." in the string and whether the character before and behind "." is numbers. Because only numbers can show around ".". Then, we should check whether there are spaces between numbers and characters, which is obviously invalid. After this check is passed, we should check whether every operator in this string has correct parameters. This check has different branches according to the type of operators. For example, logic not can only has one parameters, "-" can have either numbers or other arithmetic operators before itself because it can represent either negative number or minus sign.

After all the above checks are passed, we can identify the string is a look-like valid expression. "Look-like" means that the string is only valid in form. It may be still invalid because of some reasons such as operands type error. So we should check the string furthermore.

First, we split the string into pieces of operators and operands, and then check them one by one. If the item is an operand, we push it into a stack and push its type into another stack. If the item is an operator, we should compare the priority between this operator and last operator. If the priority of this operator is lower than the last one, we should pop last operator from the stack which is used to store operator and pop two operands from the operand's stack. Then use these operands and operator to make a computation, and then push the result into operand's stack. During the computation, we should check whether the types of operands are suitable to the operator. Then, we should continue these steps until the priority of this operator is higher than the operator in the top of the operator's stack. In this case, we should push this operator into the operator's stack. The above steps should be execute repeatedly until the operator's stack is empty and then we get the computation result of the string.

Through above steps, we get the computation result of the string in one pair of brackets. We should replace the old string with this result and repeat above steps to check the string in the other brackets and get final result.

2. ADT

HashMaps & Stacks:

The interpreter applies HaspMap to store variable and its attributes as well as subroutine names and its line numbers. And the interpreter use stacks to store HaspMap. The tradeoff of HashMap is easy to search but may be more space to store information. The tradeoff of stack is its FILO feature, but hard to iterate its inner data. So this interpreter use a link list to simulate a stack. It gives out several stack interface and also allow programmers to traverse some HashMap along with a specific principle. Also, when we parse the pair match of brackets, we also use a stack to implement the matching.