Jiang Wu jw3026
Ningning Xia nx2120
Sichang Li sl3484
Tingting Ai ta2355
Yiming Xu yx2213

Project Proposal

Stint: A Language Breaks Barrier Between String and Integer

1. MOTIVATION

In real-life applications, we often encounter the situation in which a mixture of characters and numbers needs to be processed. In traditional programming languages, such as C and Java, reading or modifying a single word/number in a string has been made quite complicated. For example, Java requires an extension, i.e. the String class, to perform the parsing and extracting tasks, and when it comes to a mixture of character words and numbers, the problem will be even more complex. While some scripting languages, e.g. Python, pretend to make the task easier, the lengthened running time and weak error-checking mechanisms provided by the compiler overweigh the convenience.

Motivated by these factors, this project tries to design a language that provides easy ways to process strings directly and conveniently by a bunch of built-in functions, specified by certain operators.

2. INTRODUCTION

The programming language *Stint* can process textual data more efficiently and effectively, aiming at providing solutions for the complex manipulations and conversions between the most common-used data types: *string* and *int*.

There are several key features of Stint while programming:

- Intelligently distinguish between *string* and *int*.
- Provide a simple and direct way to do mathematic operations for numbers in string without extracting them and transferring data type.
- Define more effective text-manipulation via operators. Meanwhile, maintain traditional string features and functions.
- Make input and output functions more convenient.

In practice, *Stint* can be applied to pre-process initial textual data and text file, and save formatted data to specified file for further use.

3. PRELIMINARY DESIGN

3.1 General Syntax

- **Naming**
  The name of variable can only include 26 alphabets in both lower and upper case (a-z, A-Z), digits (1-9), and underline (‘ _ ’). In addition, the first character can only use 26 alphabets in lower case.

- **Data Type**

  The only available data type is string.

  - *Declaration*

    User does not need to indicate data type when define a variable. Directly using "assignment" operator (' = ', see 2.1 for detail) is fine.

  - *Expression*

    Quotation marks (' " " ') are used to express string by adding them to both sides. Noted that when string contains only digits, quotation marks are no more necessary (e.g. 123 can be directly assigned to a variable).

  - *Access*

    The language provides three ways to access part of the string. First, user can access a single character or sub-string of any length at any position like other languages like C++ or Java. What's more, they can access any indicated set ("set" is defined as a range of continuous characters or digits. E.g. in string "abc123de34", "abc" and "de" are string sets, and "123" and "34" are number sets) of string or number. (See detail of access way in 3.2)

  - *Index*

    The index will start from 1 in both string and integer cases.

- **End-of-Sentence**

  Semicolon (' ; ') is used to indicate the end of one sentence.

- **Comments**

  Double-slash (' // ') is the only comment sign, and it only works for single line.

- **Keyword**

Keyword	Description	Expression
return	Terminate program	
std	Indicate the program should input/output to/from standard stream	<< *std var*
open	Open corresponding file	*open* "doc_name.txt"
close	Close corresponding file	*close* "doc_name.txt"
while	Define a loop	*while* (*condition*) {*statement*}
true	Boolean	
false	Boolean	

3.2 Operator

Name		Sign	Expression	Description	Example
Assignment		=	*expr1* = *expr2*	Assign the value of *expr2* to *expr1*	s = “abc”;
Position Indicator		@	@ *pos*	Indicate specific position for operation (use with other operators)	See in other operators’ examples
String Operator	Insert	+	*expr1* + *expr2*	Default: append *expr2* to the **end** *expr1* Use @: insert *expr2* into *expr1* at position indicated by @	*“ab” + “c” -> “abc”* *“ab” + “c” @ 1 -> “acb”*
	Delete	-	*expr1* - *expr2*	Default: delete the **first** *expr2* from *expr1* Use @: delete the **first** expr2 starting from the position indicated by @ from *expr1*	*“cabc” – “c” -> “abc”* *“cabc” – “c” @ 2 -> “cab”*
Integer Operator		.+ *	*expr1* .+ *expr2*	Default: do calculation for the **first** set of integer Use @: do calculation for the ***n*th** set of integer indicated by @	*“ab12c56” .+ “1” ->* “ab13c56” *“ab12c56” .+ “1” @ 2 -> “ab12c57”*
Character Extractor		[ ] [,]	[*index*] [*start*, *length*]	Get the nth character in bracket, or get the string starting from *start* with *length*	s = *“abcd”*; s[1] -> *“a”* s[2,2] -> *“bc”*
Set Extractor	(string)	< >	< *index* >	Get the ***n*th** set of string (number ignored)	s = *“abc12ed34”*; s <2> -> *“ed”*
	(integer)	.< >	.< *index* >	Get the ***n*th** set of number (string ignored)	s = *“abc12ed34”*; s .<2> -> *“34”*
I/O	Output	<<	<< *destin var*	Output the string in *var* to the *destin* (file or *std*)	<< “test.txt” s << *std* s
	Input	>>	>> *source var*	Get a line from *source* and store into *var*	>> “test.txt” s >> *std* s

* Integer Operator actually include add (‘ .+ ’), minus (‘ .- ’), multiply (‘ .* ’), divide (‘ ./ ’), and comparison (‘ .== ’, ‘ .< ’, ‘ .> ’). Here only use add as example in the table.

4. Application Case & Sample Code

One typical scenario for using our language is when we need to process text mixed with numbers, for example, an inventory table. Suppose we have a .txt file, in it each line records the information of a certain kind of good. There might be the name of the good, the price and the number remains. See the Table blow:

Model: 11’ Macbook Air 999$ Remain: 104
Model: 13’ Macbook Air 1199$ Remain: 82
Model: 13’ Macbook Pro 1199$ Remain: 196
Model: 15’ Macbook Pro 1499$ Remain: 54

In this Table the domain and property follow a certain kind of format, specifically A:B. But there is no guarantee. Because in reality the file might from someone's causal hand input. So the pattern matcher in Java or C++ sometimes doesn't work when you try to modify something. We designed a more efficient way to implement the modifying demand.

Each line is a mixture of characters and numbers. We often need to modify the number of items as the number is constantly changing. Using our language we can modify the number of items in a string directly, using command like "s = s .+ 1 @ 3". Or we can modify the price using command "s = s .+ 50 @ 2"

Our language is strong enough to deal with more complicated question, since we can operate all characters and numbers directly regardless of their position in a string. For example, we need to add some extra information because there's a sale, say, back to school sale. We can do it like this:

```
s = s .- 100 @ 2;
s <2> = s <2> + "Back to School Sale!";
```

With the simple two commands, the string s now:

```
Model: 13' Macbook Pro Back to School Sale!  1099$  Remain: 196
```

As we only have one variable type, the whole process of modifying a string becomes much simpler than using C++ or Java. No declaration, no stream buffer, no reader or anything to that nature.

Below is the whole demonstration of how we read an inventory file and output a modified version to another file by the language *Stint*.

Assume we got a file mentioned above, and now it comes to back-to-school days and we want to lower the price of each item by 100 dollars and add some text to it. We can simply write the following program.

```
open "data.txt";                    //open a file to read
open "data2.txt";                   //open a file to write
while ( >> "data.txt" line ) {      //read one line by each time

        //subtract 100 from the second integer in the string
        line .<2> = line .<2> - 100;
        //append to the second string token
        line <2> = line <2> + "Back to School Sale!";

        << "data2.txt" line;        //write to a file
        << std line;                //write to standard output
}
close "data.txt";                   //close a file
close "data2.txt";
return;                             //terminate the program
```

After executing this program, we will get the following lines from both data2.txt and standard output (monitor).

```
Model: 11' Macbook Air Back to School Sale! 899$  Remain: 104
Model: 13' Macbook Air Back to School Sale! 1099$  Remain: 82
Model: 13' Macbook Pro Back to School Sale! 1099$  Remain: 196
Model: 15' Macbook Pro Back to School Sale! 1399$  Remain: 54
```

It contains only 11 lines of code and it's very neat. If written in java or C++, it contains a lot more than that.