Data Processing Language

Aman Chahar (ac3946)
Miao Yu (my2457)
Weiduo Sun (ws2478)
Sikai Huang (sh3518)
Baokun Cheng (bc2651)

Language Reference Manual

1. Introduction

Data Processing language is designed to handle complex file operations and data parsing implementations. We provide easy I/O operations for handling multiple files together with functions like merge, split, copy, write etc. that requires either high level language or very lengthy/cumbersome low level language syntax and complexities. Most file operations involve some of the data processing and thus, we provide flexibility in our language to do them with ease. As most of the reading and writing into files are done line by line, we have designed our language to handle these operations easily by dedicating data types like Line, Para etc. that specifically handle chunk of data together. For the scope of project we are focusing on text files, as these are one of the most common operations done by any useful program. Similar to most of the mainstream languages, we primarily support imperative programming. Syntax is made similar to C/C++ that will be compiled to LLVM code. Extension of the language would be .dp and other details are explained in the following sections.

Some common functionalities:

1) Splitting of a file into 2 or more files

2) Merging of two files in one single file

3) Copying some lines from one text file to another file (by line number)

4) Deleting some lines a file (by line number)

5) Deleting/copying some lines based on query term

6) return size of a file(line), number of letters and size of a line(column)

We want to develop dedicated language that has easy syntax like python and gives efficient performance. Abstraction of files operations and optimized data processing will allow people without much computer science background to easily maintain the data they need. It can be extended to incorporate many optimizations making it suitable for both efficient file and data processing, computing and coding.

2. Lexical Elements

Token are separated using white space characters like space, tab or newline.

2.1 Identifiers

Identifiers are strings defined by sequence of letters and digits. Similar to C language, they should start with an alphabet and may contain underscore. Rules are defined as:

|’0’-’9’	{digit}
|’a’-’z’	{lowercase_letter}
|’A’-’Z’	{uppercase_letter}
|lowercase_letter|uppercase_letter	{letter}
|(letter)(letter|digit|_)*	{identifier}

2.2 Keywords

These keywords are reserved for use in the language and therefore cannot be used as identifiers. These keywords are case sensitive:

int	char	String	Line	para	float	File
return	if	else	for	while	break	
continue						

2.3 Literals

2.3.1 Character Literals

A character literal is expressed as a character enclosed in ASCII single quotes. However, there are few special 2 character literals that start with backslash (‘\’) that escapes the following character. Some of the special character literals are :-

‘\n’ - New Line

‘\t’, - Tab

‘\\’, backslash

\’, escape single quote

\”, escape double quotes

2.3.2 String Literals

A string literal consists of zero or more characters enclosed in double quotes. They can be accessed using [] operator. Strings are defined as mutable internally and any change done by using [] will return new string.

Example: “abc”

2.3.3 Integer Literals

An integer literal may be expressed in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2).

Example: 2016

2.3.4 Floating point Literals

Floating point literals are defined exactly like C.
“A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.” C Language Reference manual

2.4 Comments

Comments are block of code enclosed between /* and */. You can define inline comments using // operator as described in next section. Comments can span across multiple lines.

2.5 Operators

Symbol	Description
=	assign
+,-,*,/	corresponding math operators
++,--	increment/decrement by one
%	modulus operator
==,>,<,>=,<=,!=	comparison operators

“”	string identifier
‘’	character identifier
()	order enforcement operator
[]	index accessing
<>	specify elements in collection, or combine different types
|	Combine results
//	Comment everything in that line following //
/* */	Multiple lines comment

3. Data Types

3.1 Primitive types

int: 32 bit signed integer. Range from -2,147,483,648 to 2,147,483,647. The first bit represents the sign, the others represent the value.

float: 32 bit single-precision floating-point number that follows IEEE 754 standard.

char: 8 bit integer (0-255), is used to represent the character. It can contain any 1-byte long ASCII characters

3.2 Derived types

String: a char array ends with \x0. characters can be accessed through the index. E.g. for a given String k, if you want to access the 7th character in k, you can use k[6].

File: A file handle for file reading or writing. Similar to file type in C

para: An internal collection of <File, int, int>. It’s used to refer a specific part of file. For example, for a given File a, para k=<a,3,5> refers the content between the 3rd and 5th (inclusive) elements in the file.

Line: Line is collection of a string and line number (<String, int>) which is used to keep track of line number in file we are referring and what is the text of that line. We have explicitly defined this datatype because file operations generally require both things.

3.3 Collections

Collections are similar to structures or classes where you can combine multiple types of objects together. It can be defined using <> operator.

Collection<int,string,int> abc;

To refer individual values we use $ operator.

abc$[1] will retrieve first integer from out of the tuple <int, string,int>

3.4 Arrays

Arrays are defined as collection of similar types. Arrays can be accessed directory by using [] operator. Their size can be increased, decreased as required. Appending new element comprises of using + operator

newarray = [array + 0]; (new element should be of similar type)

We have allowed multidimension arrays as well. It can be accessed by nesting [] operator. New arrays can be appended as rows by using :: operator.

Example:-

newarray = [array :: anotherarray] ;

newarray[1][2] will give second row and third column.

To make collection of different types use Collection datatype and then make an array of Collection.

4. Expressions and Statements

4.1 Expression

Expressions are statements that end with semicolon and usually comprises of assignments and function calls.

expression ;
Example
int a = 1;

4.2 if and if else statements

The if statements is used to conditionally execute part of program with true detected by system.

```
// if

if  expression
{
        statement
}
```

```
// if else
if  expression
{ statement 1 }
else
{ statement 2 }
```

if statements detect the true value, if it meets it the statement 1 will be executed. However, it meets false value the statements 2 will be executed. The execution code without a braces.
Here is an actual example:

```
if x == 5
        { save ( output ," Output . txt “);}
```

if x == 5 is true, then the statement save (output ," Output . txt “); will be executed. But if x == 5 is false another example is

```
if x == 5
        {save ( output ," Output . txt “);}
else
        {print ( id  * idf); }
```

above example shows that when if meets false the statements , in the else block, will be executed.

4.3 while statement

The while statement with an ending detection at the beginning of the block.
example:

```
//  while loop
while  expression
          {statement}
```

the while statement detect the expression first. If the expression is true, the statements will be executed. And the statements will be executed repeatedly till the system meets the state will never correspond to the expression.
The example below shows a while statement, which save out the file name with 1 to 5.

```
int file_name = 1
while file_name <= 5
          { /* Do some operations here); */
                    expressions
          }
```

4.4 for and for each statement

The for statement is useful statement to iterate the program states.
Example:

```
//  for loop
for  initialization:termination:increment
{
          statement
}
```

The for statement evaluate the initialization first, then it evaluates the expression increment. Each increment loop, the for statement will detect the increment is correspond to the termination states. If is not meet the termination states, it will execute the statement. However, when it meet the state that correspond to the termination state, it will break out.

Example:

```
for  int file_name = 1 : file_name <= 5: file_name++
          save ( output ," file_name . txt “);
```

above example is the for- version of while loop. The system will execute the 1.txt, 2.txt, 3.txt, 4.txt, 5.txt

for each statement is similar to the for loop. the difference is that the input is array not the expression.

Example:

```
//  for each loop
for  element in  elements
statement
end
```

Example:

```
$File_array = array["1", "2", "3",”4”,”5”];
foreach $File_array
          save ( output ," $File_array . txt “);
```

Above statement output will same the for statement. 1.txt, 2.txt, 3.txt, 4.txt, 5.txt.

5. Functions

Custom functions can be defined using C type syntax

return_type function_name(arguments)

We also (want to) incorporate function overloading concept that allows same function name with different type of arguments. Ability to pass variable parameters can be done using by passing array.

Functions are called by using *call function_name(arguments)*

6. Declarations

6.1 Input & output

6.1.1 File operations

Our language requires handling of files and thus, requires extensive use of file handling operations. Some of the library functions we have are: -

Open(String FileName, String mode)

Returns File pointer

By default open up in Read mode but other modes can be explicitly mentioned: -

“r”:open the file for reading

“r+”:update the file(read and write)

“w”: construct a new file for writing and clear the old file

“w+”: construct a new file for updating

“a”: append

Example:

File a = open(“abc.txt”);

File b = open(“newfile.txt”,”w”);

API:

int close (File)

return 1 on success and 0 otherwise

Closes the File pointer and frees the space.

int delete(File)

return 1 on success and 0 otherwise

Deletes the file associated with file pointer.

int rename(File, String NewName)

returns 1 on success and 0 otherwise

Renames the file associated with the file pointer.

merge(FileA, FileB)

merges two files appending second file to first one.

copy(String initial file ,String new file)

copies the current file to new file.

6.1.2 Formative output

print (formatted string)

Output Stream can be used to specify either File or STDOUT. Desired formatted string can be obtained by combining various variable values, arrays, collection. You can combine using + operator. There is no need to write loops to get all the values.

As data processing requires lot of formatting, we introduce repeat function with print. For repeat, we use $1, $2 as arguments for variables that we want to repeat.

repeat(Repetition, list of variables)

Example:

print(“hello”)

print($1 + “ | “ + $2 + “ \n”).repeat(5, i -> [1:10], j-> [1:2:50])

will result in output: -

```
1 | 1
2 | 3
3 | 5
4 | 7
5 | 9
```

This will help in writing into new files with any format without worrying about writing nested loops in detail.

6.1.3 Formative input

String readline(InputStream, String formatted)

Scan is used for reading files in particular format and obtaining values without the need for formatting explicitly later on. Formatted string is provided similar to what we used in print statement. But to mention the format of the desired action we use %s, %d, %f are used to specify string, integer and floating numbers.

readline(STDIN, “%s,%d,%f”)

6.1.4 Character I/O function

String read(String input, int NumberofChar, InputStream)

returns a string containing length as specified in the arguments.

char readchar(InputStream)

reads a new character from the inputstream and returns it

int writechar(OutputStream, char c)

Similar to readchar but just writes into the output stream one character

int write(OutputStream, String/para)
int write (OutputStream, String[])

Writes all the values in the output stream. If there is array of anything, it iterates and write into file.

As data processing requires lot of formatting, we introduce repeat function with write also. For repeat, we use $1, $2 and so on as arguments for variables that we want to repeat.

repeat(Repetition, list of variables)

```
write(FileA, $1 + “ | “ + $2 + “ \n”).repeat(5, i -> [1:10], j-> [1:2:50] )
```

will result in output: -

```
1 | 1
2 | 3
3 | 5
4 | 7
5 | 9
```

This will help in writing into new files with any format without worrying about writing nested loops in detail. Generally all data processing involves handling lot of intermediate files in particular format that can be read on later on.

deleteline(File, int)

Deletes line from the file

delete (File, para)

Deletes section of text from file

6.2 String functions

String strcpy(String s,int t)

returns a new String with first t bytes of the string s

String strcat(String s,String t)

returns a new String that's the concatenation of the s and t

int strlen(s);

returns the length of a given string

search(File, String)
search (String, String)

return array of indexes where string was found

searchLines(File,String)

returns Line objects which contains that string

split(String,String delimiter)

return array of strings split by delimiter

replace(String, index, newvalue)

returns new string with replaced value. Instead of index value, we can also pass array of index values

7. Statement, program structure and examples

All the code will reside inside functions. Program will start with the function **main()** and end with it. Scope of all variables are defined like in C, where variable scope starts after its definition and ends with the ending of that function.

This example contains same task as we mentioned in proposal but with new syntax

```
1. Program to create a new file with lines that contains keyword "Hillary Clinton" &
"Victory" in first file and "Donald Trump" & "Defeat" in second file.

main()
{
       //Read file 1 and file 2
       File fileA = open("Democratic_IOWA_cacuses_Analysis.txt");
       int sizeA = length(fileA);               //Compute number of lines

       File fileB = open("Republican_IOWA_cacuses_Analysis.txt");
       int sizeB = length(fileB);              //Compute number of lines

       //Para dataype that stores file pointer, start line number and last line number
       Para par1 = <fileA, 0, sizeA>;
       Para par2 = <file2, 0, sizeB>;

       //String arrays are used to store multiple results
       // | (Pipe) operation can combine results given they return same type

       Line[] firstFile = searchLines(par1, "Hillary Clinton") | searchLines(par1,
"Victory");
       Line[] SecondFile = searchLines(par2, "Donald Trump") |  searchLines(par2,
"Defeat");

       // Merge combines two files which takes input as either <fileName, collection of
lines> or <filename, paragraphs>
       File output = merge(<fileA,firstFile>, <fileB,SecondFile>);

       //Save the output into output file
       save(output,"Output.txt");
}
```

2. Program to implement TF-IDF algorithm (Information retrieval)

```
       // similar to C/C++ syntax of defining function
float compute_tf(par doc1, String keyword)
{
       String[] keyword_lines = search(doc1,keyword);
       int occurrences=0;
       for line in split(keyword_Lines,”\n”)
       {
              occurrences = occurrences + search(line,keyword).size();
       }
              return occurrences/words(doc1);
}

float compute_idf(para[] docs, String keyword)
{
       int occurrences =0;
       for doc in docs
       {
              if doc.contains(keyword)
              {
                     occurrences++;
              }
       }
       //Math library to compute log/similar operations
       return Math.log(docs.size()/occurrences);
}
```

```
main()
{
       String dir_path = "folder_path";
       //Initialize keyword is used to initialize any data type

       declare para[] files;

       // For loop to iterate over contents
       // dir library function to get all files in a folder
       for file in dir(dir_path)
       {
              files.add(<file, 0, size(file)>);
       }
       // call keyword followed by function name calls that function
       float tf = call compute_tf(files[1],"Compilers");
       float idf = call compute_idf(files,"Compilers");
       print(id * idf); //  print on console

}
```