Data Processing Language

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

Language name using initials: - BMWSA

Introduction

motivation

With the advent of Big Data era, file operations and data parsing has become more crucial. There is a need for a language that can perform more complex operations not only taking less time to process but also making it easy for programmers to write an efficient data processing application with minimal effort. Even simple algorithms like sorting on large files can take huge time. There are lots of languages like Python, R that are becoming popular in data processing domain as they let users perform complex operations with minimal effort. But being very high-level languages they tend to perform poorly (less than optimal) in spite of better processors. In many domains like Finance, stock markets trading etc. low level languages like C/C++ are still used where time plays crucial time for their business.

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.

Language description

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)

Data types:

Keywords	Description
int	32-bit integer
char	character
String	array of characters that end with \x0
Line	Line represented in a File File operations generally require reading line by line
para	a collection of <File, int, int>
float	float number
File	file handle

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
|	Combine results
//	Comment everything in that line following //
/* */	Multiple lines comment

Examples of declarations:

```
int a;  // declare an integer named ‘a’
int function k(File b, String c)
       // Code in function
       return x; // return the value you want
end
 /* declare a function ‘k’ which takes File b and string c as its parameters and the
function returns an integer.*/
declare Collection<types> //initialize a collection of a specific type
```

Library Functions

Function signature	**Syntax**	**Description**
open(String,String) returns File pointer	File a = open(“abc.txt”)	Opens the file in read mode by default. Modes w, a are available for write and append
readline(File) return String or EOF	String line = readline(FileA)	Reads a line from previous location of file pointer
write(File, String) write(File, para) write(File, collections<line>)	write(FileA,”Write this down”)	Takes arguments as file and String that has to be written. If it is para (paragraph type) containing starting and end line number it copies

		complete segment. If collection of lines is given as argument, then only those lines are copied
size(File) Returns int	int totalLines = size(FileA)	Returns number of lines in the File
split(File, int) returns two file pointers (Can be extended to splitting multiple files) split(File,array)	Collections<File> files = split(FileA, 10)	Returns file pointers to two or more files containing content as requested by user
merge(File,File) merge(Collections<File>) merge(para, para) returns merged file pointer	File output = merge(FileA, FileB)	Merges the two files and returns file pointer to new file. It can also accept collection of files that will be merged or some sections of files.
delete(File, int) delete(File, para) returns new file pointer	File output = delete(FileA, 10)	Deletes particular line or section of lines and returns the pointer to that line
close(File)	close(FileA)	Frees the File pointer
search(file, String) search(String, String) returns collection<position>	int occurrences = search(lineA, “compilers”);	It returns the line numbers where keyword is present or position if string is provided
copy(File) returns File pointer to new file	File newFile = copy(FileA)	Makes a copy of file and returns the pointer to it
save(File, String)	save(FileA,”output.txt”)	Save the current file on disk with second argument as file name

Custom 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 or collection.

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

Additional Keywords

declare : To declare any variable without the need of assignment and initialization
call : To call the function following this keyword
end : Used to mark the end of a block. For example end of control statements, function definitions etc.

Collections<type> is similar to C++ vectors or ArrayList in java that allows multiple sized arrays. It can contain any type of elements but must be declared before. It contains functions like add, size, insert etc.

control flow:

```
//Comment (single line)
/* start of the block comment
*/ end of the block comment

// if  else
if   expression
        statement
else
        statement
end

// if
if expression
        statement
end

// for loop
for initialization:termination:increment
        statement
end

// for each loop
for element in elements
        statement
end

// while loop
while expression
        statement
end
```

Code Example

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.

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

//Collection is like vectors/ArrayList that can append any type
// | (Pipe) operation can combine results given they return same type

Collection<Lines> firstFile = search(par1, "Hillary Clinton") | search(par1, "Victory")
Collection<Lines> SecondFile = search(par2, "Donald Trump") |  search(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)

```
String dir_path = "folder_path"
//Initialize keyword is used to initialize any data type

declare Collections<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)>)
end
// 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

// similar to C/C++ syntax of defining function
```

```
float function compute_tf(par doc1, String keyword)

	Collection<Lines> keyword_lines = search(doc1,keyword);
	int occurrences=0
	for line in Lines
		occurrences = occurrences + search(line,keyword).size();
	end
	return occurrences/words(doc1)
end

float function compute_idf(Collection<par> docs, String keyword)
	int occurrences =0
	for doc in docs
		if doc.contains(keyword)
			occurrences++
		end
	//Math library to compute log/similar operations
	return Math.log(docs.size()/occurrences)
	end
end
```