PROJECT 1

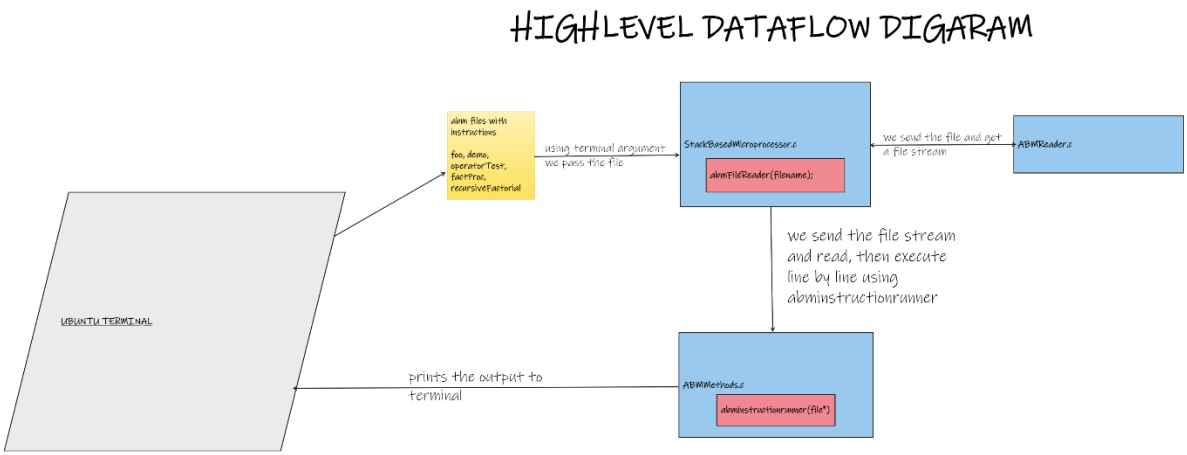
Sai vishnu Anudeep Kadiyala
001450445

OCTOBER 5, 2023

TABLE OF CONTENTS:

System Documentation:

a. High-level dataflow diagram.



As mentioned above in the diagram, the terminal takes the filename and the main function from StackBasedMicroprocessor pick the name and it calls abmFileReader method to get a file pointer which is then sent to abminstructionrunner in ABMMethod.c to execute each line from instructions.

b. List of Routines and descriptions:

1. ABMMethods.c

Routine Name	**Routine Description**
abminstructionrunner	Takes instruction FILE, reads it into a Pair of line number, keyword, command and uses it call abmkeywordhelper to execute.
abmkeywordhelper	Used to run through the given instructions read from file and execute accordingly.

trimleadtrailspaces	Used to remove forward and backward spaces from a given line from file.

2. ABMReader.c

Routine Name	**Routine Description**
abmFileReader	reads a given file and returns FILE
PrintabmFile	used to print a file

3. IndexKeywordCommandPair.c

Routine Name	**Routine Description**
initializePair	initialize pair making the current 0
addPair	it takes a line number, keyword and command to store in pair.
getlinenumber	get line number of a given keyword and command when both keyword and command match the parameter.

4. Stack.c

Routine Name	**Routine Description**
initialize	initialize the stack with size 0
isEmpty	returns if empty or not.
IsFull	returns if full or not.
PushIntoStack	pushes the parameter passed item into the stack
PopStack	pop the top of the stack and returns it.
PeekStack	peeks the top of the stack and returns it.

5. VariableManager.c

Routine Name	**Routine Description**
initializeContainer	initializes new container and assigns its number
insertIntoContainer	Inserts given variablename and its value into container.
updateContainerbyaddress	takes a variable address and its value and updates the value in the map
FindInAboveScope	takes a variable name and search for this above scope and returns its value, *This is used when passing parameters.
FindInContainer	find the given variable name in current container.
FindInContainerbyaddress	find the give variable address in given container and return its value
getaddressfromContainer	gets the variable and returns address of the variable name from container
NewScope	if it is new scope, we create a new container now.
DeleteScope	delete the current scope, used when we have been done with a scope to free space.
MakeReturnablesAccesible	takes number of returnable variables in the scope and appends those in the above scope making them accessible to the scope above.

6. Variablearray.c

Routine Name	**Routine Description**
initializeStrings	initializes the 2d array
isEmptyStrings	check and returns if the 2d array are empty
isFullStrings	check if the 2d array is full or not
append	takes data and append it into the 2d array
atindex	takes an int and returns what's stored in that location. Not used in abminstruction runner.
addressofdata	returns address of data's index to store it in map corresponding to its value.

7. Addresstovaluedict.c

Routine Name	**Routine Description**
initializeMap	initialzes the map by making its size 0
insert	takes a key value and int value and checks if key already exists, if exists update it by new value else create a new key and insert value.
find	finds the given key in map and returns its value.

8. StackBasedMicroprocessor.c

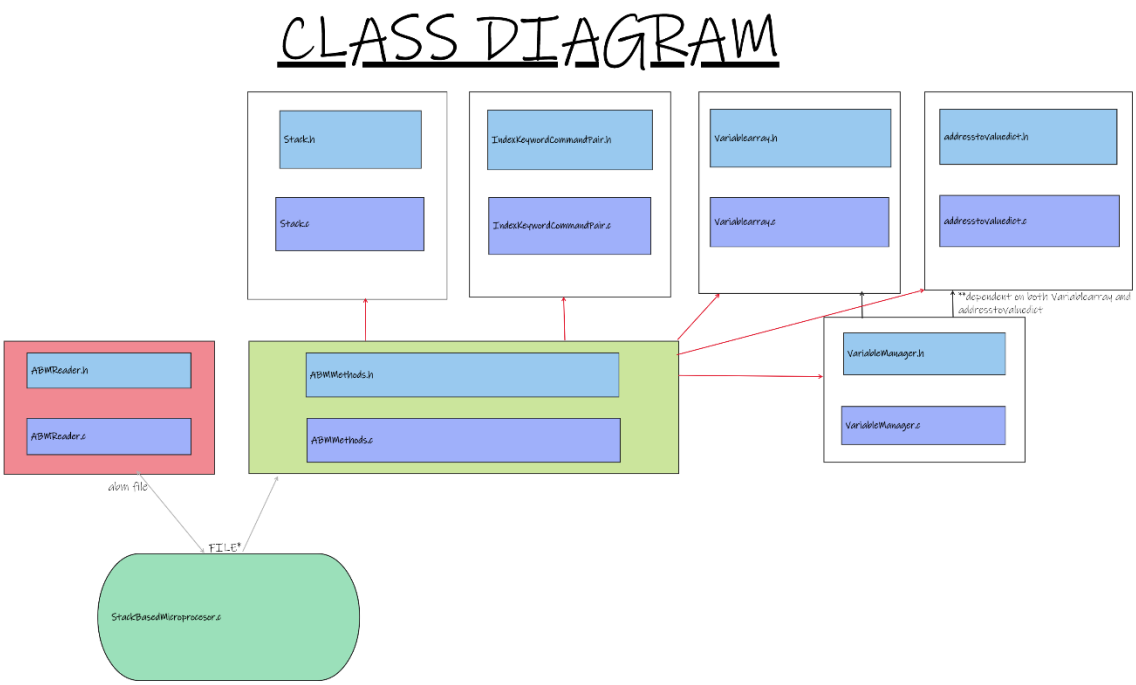
Routine Name	**Routine Description**
Main Function	Calls abmFileReader gets FILE* from it and abminstructionrunner

c. Implementation Method:

I have used structs to make *Stack, IndexKeywordCommandPair, Map, Variablearray and VariableManager* that help in running the instructions from abm file.

1. *IndexKeywordCommandPair*: A struct that holds a line number, a keyword and command that is read from an abm file.
2. *Stack*: used in basic execution of keyword-command pair and when *call* is evoked it is used to store the last control index.
3. *Variablearray*: uses a 2D array to store variable names in them.
4. *Map*: a basic key to value map to store variable addresses and values
5. *VariableManger:* It layered over 2D array (Stringarray from *Variablearray.h*), and a Map (Map from *addresstovaluedict*) to store variables and manages its scope.

These help in running the instructions perfectly and returning desired output.



As the above class diagram shows, I have headers for all the structs that get called in ABMMethods, and both ABMMethods and ABMReader header gets called in StackBasedMicroprocessor.

StackBasedMicroprocessor takes the filename as input from terminal and passes it to ***ABMReader*** where I read the file and return it. Using the Returned file, I call abminstructionrunner from **ABMMethods**. Here this function breaks the file down into IndexKeywordCommandPair. Also, creates a labellocation Map where all the labels with their line number are stored. Now with this pair and labellocation, a Stack and VariableManager container are passed to abminstructionhelper inside ABMMethods where it loops through the pairs one by one and basing on Keywords it performs corresponding actions.

For keyword in stack Manipulations, we tend to call the corresponding methods from Stack.c to perform the action.

For keyword is Arithmetic, Logical and Relational Operators we pop the top two elements of passed stack and perform the operation. Whenever show is encountered, we print the command passed, and if print is encountered, we print the top of the stack.

Before we move to subroutine control, a little Note about the scope of variables:

For rvalue:

1.if only begin[s] is/are evoked then we can access the above scope and get the value of given variable

2.if Begin & Call is/are evoked then we should only have access to current scope to find given variable.

*3.Mathamatically everytime ** n(begin) > n(call) ** we can access the scope above current and find a variable else NO*

For lvalue:

1. under any circumstance of begin/call/return when we make a new variable it is always in the current scope

2. if we have encountered call and then encountered return, in begin/end everything after this is accessible its above scope so we count them and make them accessible.

So according to the above-mentioned logic, we decide which scope to work from Variable manager and every time we encounter a begin or call, we increment count of them and every time we end or return, we decrement the count which is used to determine the scope of variables to access. Also, once the call is returned, we then count the number of returnable variables made before end is encountered, and we make those variables accessible to previous scope and delete the newly made begin/end scope as we have encountered end. This has been the logical implementation of subroutine calls.

Now moving to Control flow, while reading the file we mark the labels and their locations in a map so every time we encounter label, we just check we have the location of label, and for goto we jump to the given target label accessing its location from map. On gotofalse, if the top of stack is 0 then we jump to the label, and for gototrue, if the top of stack is nonzero then we jump to the label.

Also, Abminstructionhelper creates a new stack to store the index when a call is called to make sure that when the call returns it passes the control back.

In this way when a file is passed and returns the output of the given instructions in terminal.

Test Documentation:

a. How Did I Test My Program:

I have used given operatorTest.abm, foo.abm, demo.abm, factProc.abm and recursiveFactorial.abm to validate my algorithms efficiency.

Below are the screenshots of outputs for each file. To mention they have matched exactly with each of the .out files provided showing the provided solutions credibility.

i. *operatorTest.abm:*

ii. *foo.abm:*

```
"Consider the CALLER the routine which
is calling the CALLEE"

before foo r is:
2
_____________________________________
p is a formal parameter and
r is an actual parameter
therefore the call may be seen as
foo( r );
and function foo may be seen as
foo( int p )
_____________________________________
in foo r is local.
therefore  r is:
0
_____________________________________
in callee foo, the value of p is:
2
_____________________________________
value of p in caller function is:
0
```

iii. *demo.abm:*

```
This code illustrates parameter passing strategy.

before function work:
value of x is:
0
value of f is:
5
_____________________________
the call to function work may be seen as
work( f, x );
and function work may be seen as
work( INOUT int ff, INOUT int xx )
_____________________________________
after function work:
value of x is:
1
value of f is:
6
```

iv. *factProc.abm:*

```
factProc.abm ( Computes 5 factorial
using a loop )

function call to fact may be seen as
fact( f, n );
function fact prototype may be seen as
fact( INOUT t; IN i )

5 factorial is:
120
```

v. *recursiveFactorial.abm:*

```
Computes Factorial
Factorial of
5
equals
120
```

b. Bugs and errors in code:

Currently the code doesn't have any bugs or errors as mentioned above it gives the desired output perfectly for the given input of instructions.

c. Executable Version of solution:

Due to too many headers and .c files, I was unable to paste it here, please check it in my submission or my github repository maintained by me which will be public after the deadline.

[OCT 5th, 2023]

d. Difficulties and solutions involved in creating solutions:

Having an encounter with multiple errors and difficulties, I would like to mention that variable scope has been the standalone difficulty that had to be handled. And building a solution involving customized structs made the work much easier, I made Variable Manager which handles variable creation, update and manages their scope. Apart from that the familiarity with language C is not profound for me making it a little time consuming to solve, create, or debug.

Algorithm and datastructures:

a. Data Structures:

   i. *IndexKeywordCommandPair*: a struct that has an int line number, char arrays keyword and command which makes this struct an Index Pair. This is used to store the file as this way of reading it makes it easier to collect labels, jump to target, understand scope.

   ii. *Stack*: a struct using a char array as stack to pop/push/peek elements. Is used as the given abm functions depend on stack-based operations. Apart from this I used it in storing the last index if a call is evoked to jump back once returned.

   iii. *Variablearray*: uses a 2D array to store variable names in them. This is the efficient way as I can get the address of variable name, and as string in C are char* arrays I used 2D array.

   iv. *Map*: a basic key to value map to store variable addresses and values. This also is the most feasible option for the way I was solving as this makes it easier for me to read find the address of a variable name and return its value.

   v. *VariableManger:* It layered over 2D array (Stringarray from *Variablearray.h*), and a Map (Map from *addresstovaluedict*) to store variables and manages its scope. This is used to make a new scope leading to a new 2D array and map every time new scope is made. This helps me to manage both 2D array and Map together and decide accessibility of a variable according to the scope.

The above-mentioned structures are mostly derived from arrays and customized to their desired characteristics. They all are chosen intuitively to aid solving the given problem in the most modular way I can.

User Documentation:

a. What operation system used: UBUNTU - 22.04.2 LTS

b. How to compile your code:

After having all the .h and .c files,

In ubuntu terminal use:

gcc Stack.c ABMMethods.c StackBasedMicroprocessor.c ABMReader.c Variablearray.c addresstovaluedict.c IndexKeywordCommandPair.c VariableManager.c -o Project1

which compiles my code.

c. What other applications are required to run: NONE.

d. How to Run your program with parameters:

After compiling my code as mentioned above,

if you root into **Project1** and enter the abm filename that must be executed

./Project1 "file-name" *=> ex: ./Project1 abmfiles/demo.abm*

You should now see the desired output printed in the terminal.

e. Other requirements: NONE.