Destin Krepps Cmsc313 AssemBullet: Prof Potteiger

This is the name of the language I’ve created that’s based from assembly

Content-

**Language documentation** (intro and how to write an AssemBullet program)

**VM documentation** (how to work the vm and how its code works)

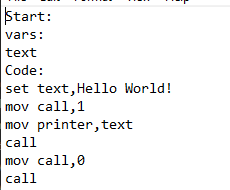
Language Documentation

Intro:

The name of the language is based off how the call function works. Because you load the call command before invoking the call, it resembles actions of a gun. You load the call then fire it by calling ‘Call’ itself.

How to begin creating the program.

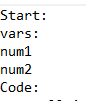
‘Start:’ must lead every program to be recognized by the runtime environment.

For example -> (img1) 

After start is called

You can initialize your variables by leading the section with ‘Vars: ‘ but it is not necessary

You write in your variable names line by line (I will go into how this system works in the VM code later)

For example-> (img2)  this will initialize two variables ‘num1’ and ‘num2’.  
\*you do not need a vars section this is optional, and you may place this section wherever as long as its not within another section\*

‘Code:’ is a mandatory section and must be prefaced with that on a line

Within code you can use the following commands

**Mov** (this will move something from the second param into the first)

(‘Mov parameter1,parameter2’) - parameter 1 must be a register (call, printer, r1-10,) printer is what will be outed in the system write sys call

- parameter 2 can be register r1-10, a variable initialized previously, a method name, or an input of your choice (this will be considered a string and if used for int properties it will be dealt with within the VM)

Notes on mov (please structure this with the with a space after the mov command and a comma and no spaces between the parameters), and moving a 0, 1, or method name, will load/cock the call register.

**Set** (this command is specifically for variables within the vars section)

(‘set parameter1, parameter2’) -parameter1 must be a variable name previously initialized

- parameter2 is a string of your choice and if you’re using this for an int purpose it will be dealt with within the vm code

Structured the same way as mov though the parameter two may have spaces (no commas however as this was the delimiter)

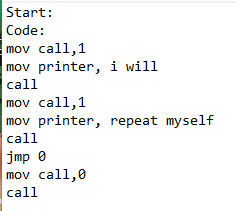
**Inp**  (This is the system call for scanning input and will accept input into variables previously initialized )

(‘inp paremeter1’) - parameter 1 must be a variable name previously initialized in the vars section

(this will be stored on the data space right after the name of the var provided as a system to keep track of variable names and what they hold because C++ sucks and you can run text strings as code)

**Jmp** (this will jump to a specific instruction (instructions are on each line of code section) index, each instruction is held in a stack like data structure and can be indexed based off of the line they written in for example the first instruction is index 0 the next line is index1, after you jump to that index you will rerun through all the code up until jump was called then you will continue as if jump was not there.)

(‘jmp 0’) - parameter1 must be an integer index that does not exceed the number of lines before the command call

 (index starts at 0 and starts first line after code:)

0

1

2

3

4

5

6

7

8

**add & sub** (this will add two parameters together and will store the sum or difference in the first parameter)

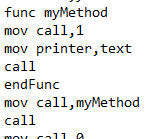
(‘add parameter1,parameter2’) || (sub parameter1,parameter2)

- parameter1 must be a variable or general-purpose register because the calculation must be stored

-parameter2 can be a variable, general purpose, or an integer (will be converted from string);

**func** (This is how a function is called anything within the function will not be ran till the method header is called using the call command. The method must be enclosed by a ‘endFunc’ command)

(‘func parameter1’) - parameter 1 is the method name/header

for example -> (img3)  this method will run once called and will print whatever the text variable is.

**Call** (This command pulls the trigger on whatever was loaded within the command from the previous mov instruction within it)

(‘Call’) - no parameters Call will be used as a finalization on what was inside

What can be called : ‘0’ – system exit, if 0 is inside of call the program will exit

‘1’ – system write, if 1 is called it will print whats inside the printer register for example of usage see ‘img1’

‘a method name’ – this invokes the method called to be used and the method name will be the parameter of the ‘func’ command . for example (mov call,method1) is what you can load it with.

Language Notes:

As the text file is being parsed it is important to follow the syntax closely as to not receive errors when running the file on the vm. ‘Start:’ and ‘Code:’ must be present in every assembullet program and is .txt

**VM documentation**

Intro: Im really happy with this program, it was one of those you’re a given a lot of room for creativity, and although I ran into issues in my choice of c++ like you cant name variables from strings from text files I used a data system instead to keep track of things like that it was fun and frustrating. Im happy with my work and im happy its over as well lol.

**How to work the environment:**

When you start the VM (run the c++ code) you will be prompted with a welcome and an asking of the file you’d like to run ( you will write for example HelloWorld.txt). If you’ve entered a valid File it will parse the file and translate it and run it based on its content similar to an assembler and assembly. If you’ve entered an invalid file you will be met with an error message saying you did something wrong and maybe you forgot your .txt or .Start at the beginning of the file.

**How the environment works behind the scenes/code explanation (in order of appearance):**

**Prework [**

-Started by including every library I may need for this program to work

-A prototype called reparse that takes in a single instruction and runs it as an instruction is created early in the file and will be used later with the call command to run a function later and not right away so the code wont always be read line by line if a function appears, also if jmp command appears you want to go back to where you jumped and reload the old files this is done with this method taking in instructions form the instruction data holder that’s indexed from a vector.

- I then used 3 vectors of string type because the txt file pulls everything as a string , this is for global vars. in the variables vector it takes from the txt files sets them to first index and initializes them with “” empty quotations on the second index

* instructionTracker – this vector indexes the instructions so I can keep track of them and is essentially my RIP and will be used for jmp instructions later
* methods – this vector holds all the names of functions if they appear within the program
* Variables – this vector holds the name of the variable in index[i] and holds the data for it in index[i+1], I do this because c++ cant take in strings and dynamically name variables with them so this system will allow me grab variables based off the names given by the user later.

- Then we created an array stack to hold local memory and parameters as needed, this stack is used a lot in this program grabbing parameters from commands and making it easer to use them

-create a couple of Booleans one to end program other to remove a previous jump

-Initialize registers, ‘call’ to call things, ‘printer’ to print things, ‘input’ to input things, and general r1-r10

**]**

**Entering main (variable setting from assem file first after prompt user for the file in question)[**

**-**welcome user and prompt them for the file and set that variable to textFile to use for instream later,

- create a string called instructions and another for varSearch that does the same thing and set each line to them as you go

- if the file beings with ‘Start:’ its assumed to be a valid program if not tell user invalid file

- if is valid look for ‘vars:’ section

-if vars section is found get each line and initialize it to the variables vector by pushing back the variable name in the assem file and a blank space in the next index to initialize it and can be changed later

- if the ‘vars:’ section isn’t found it ends the first read of the file (vars can beat on top or bottom of code section because if another section or the end of file is found after ‘vars:’ it ends the first read )

**]**

**Second read through for code section now that variables may have been initialized[**

**-**instead of using varSearch as my instream use instruction and load lines into that variable

-if the instruction is ‘code:’ you’ve found the code section begin running command lines

-string command is the first item of the full instruction so if mov printer,helloWorld is the instruction the command is mov and we use that to identify how to handle it in c++

**call**

-first we check for the call instruction which is always by itself if the call instruction is found we first check to see if a function is loaded in call because we have to use an external c++ function to run this outside of line by line run I will explain this later when explaining that c++ function reparse (). Ill mark the explanation by a green

What is loaded in call?

* 0 – sets endRead to true stops the reading of the file and ends the c++ program gracefully simulates a exit syscall
* 1- prints what’s within printer register and simulates a syscall for writing to the terminal
* Method header- uses the instruction variable to search for the method name in the method vector then search the code using the instructionTracker (RIP) for the function header then you’re going to reparse the code within the function that was called using reparse line by line until you reach the line endFunc then you continue regular read through.

**func**

- if call wasn’t the instruction we check if it’s a function by looking for func command, if found were gonna cut the func command off the instruction and add the parameter which is the function name to methods vector to be used later when caller is looking for the function

- we also want to push the instructions from the function onto the tracker although we don’t want to actually run them until called to we push it onto the instructionTracker data holder (RIP) and then grab the next line until we reach the function closer instruction ‘endFunc’

**mov**

- if its not function we check for the mov command if found we check the cut off the mov instruction and check the first parameter using a comma delimiter we push the parameters onto the stack

* if the param 1 was call were going to be loading call and we set call whatever the second parameter is
* if param 1 is printer we set the printer to either a varaiable or you can hard write the printer text in for param 2
* if param 1 is a general register we check if param 2 is a register or variable set param 1 to what param 2 is if not just set the register to whatever is written in for param so you can write a number or any string into a register as well (comparing param 2 to registers is what takes up a good chunk of code space)

-after performed the mov we want to pop the two parameters off the stack as were not using them anymore by two pop() calls

**Set**

**-** if the command was a set command, we know were working with variables we do this because the initialized variables earlier were only set to its next index which is just empty quotes

- this is where we get a little creative because you cannot dynamically create variables with their names being strings, you can do this in other languages but c++ works at compile time unlike python which I can see now was recommended for a reason

- we grab the parameters and push the parameters onto the stack we then search the variables vector for names of the variables (i) and set the location to a integer that we got by using a find function we then want to set that variable by using the next index (i+1) and set that to parameter 2 from assem program

- were done with these params so we pop them off

**Inp**

- if input command is found we push the input parameter onto the stack

- this parameter must be a variable name initialized previously

- we then compare the variables vector to the parameter from the stack to find the location of the variable in the vector

- we get the input from the user

- we set that input to the location we found earlier in the variables vector +1 as it goes with the system we created

-pop the parameter off the stack as were done with it

**jmp**

-if the command is jmp it refers to a jump process for control flow

- we grab the parameter which refers to an index of a instruction

-we then track the instruction were going to by accessing it from the vector (RIP)

-we check if weve jumped before if we haven’t then we preform the jump, this is so we don’t infinitely loop back to this jump again and again. So if this jump has been preformed we set jumped to true

- next were going to reparse(old instructions), this goes to where we want to and re reads it and instruction by instruction we go back through and perform the commands until we reach the jump command we originally started with

- once we get back to the jump command we started with we ignore it and continue with the rest of the code

\* this emulates the assem jump function and does looping and recursion\*

**Add || Sub**

**-** im grouping these next two commands asthey are practically the same but signs are switched

- if the command add or subtract is found we push the params onto the stack

- the first parameter must be a register/variable because we store the total or difference in it

- the second parameter can be a register, a variable, or hard coded in the assem file as a string

- if param 1 is a register we check if param 2 is also a register if it is we convert first register and second register to ints using stoi() which is string to int, we do the calculation and store the total/diff back to the first parameter using to\_string(int) which turns an int to a string

- if param 2 is not a register we just do the math from the code

-the reason we can use variables so that we may do math on user inputs we set two variables to user input using the inp command then do math on them

-if it is two variables we find the first variable name mark its location find the second variable name mark its location as well then we use the variables location +1 to do the math then store it in the first variables name location+1

- pop the parameters off the stack as were done with them

**Else**

- and if there is no Start: or the files not right name we end the program

**]**

**Reparse method[**

- this method takes in a string as its parameter which will be an instruction that the program wants to run

-so we do the exact same things as above the only difference is, is that we use this for functions so we can run them when called and not line by line

- we also use jump functions here because we cannot restart the file completely based off index so instead I use this method to get around restarting completely just to jump to a location;

- this method takes in the command and runs the line after its done it goes back to the code and most likely the code will grab the next line and reparse again.

**]**

**End Notes**

Im sure my life would’ve been much easier had I used another language, but I am really happy with it.

If youre unsure about anything on this project feel free to email me [dkrepps1@umbc.edu](mailto:dkrepps1@umbc.edu).