BASIMPL MANUAL

Contents

Introduction

BaSimPL Language has been influenced by TinyADA, Python and C languages, learning from all three and imbibing their best and most useful features. The concept of stacks implemented in our language was influenced by that in python, the separation of definition and declaration from TinyADA and the bacis programming construct from C. We have implemented it using Python language. The Compiler and Interpreter have been organized into two separate steps, as they are supposed to be, and the sub-parts of a Compiler – the Parser and the Lexer, have also been divided into two distinct steps to provide a neat flow. The grammar we built has been listed later on in the manual.

Further in-depth details of the BaSimPL language can be found in this manual, or on our Wiki Page:

https://github.com/harsha-kadekar/BaSimPL/wiki

The YouTube link to our presentation video is:

https://www.youtube.com/watch?v=HJjdUxsjX6w&feature=youtu.be

Github repository link –

https://github.com/harsha-kadekar/BaSimPL

Executing the code

To run our code, you need to execute **BaSimPL_Execute.py .** It can be passed multiple arguments

Usuage: **python BaSimPL_Execute.py [inputfile=filename.smpl] [debug=1] [outputfile=filename.bspl]**

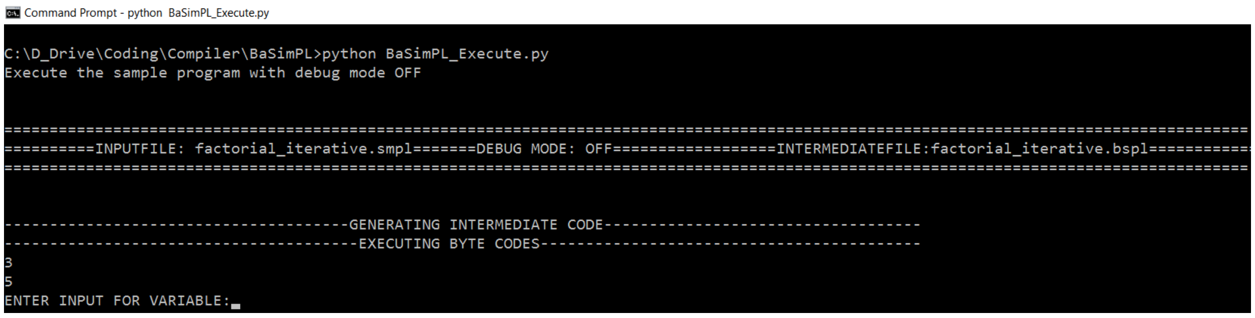
All our source program are having file type as .smpl and all our intermediate byte code files have type .bspl

These files can be opened in the notepad.

All the arguments present inside square brackets are optional. So you can just give

python BaSimPL_Execute.py

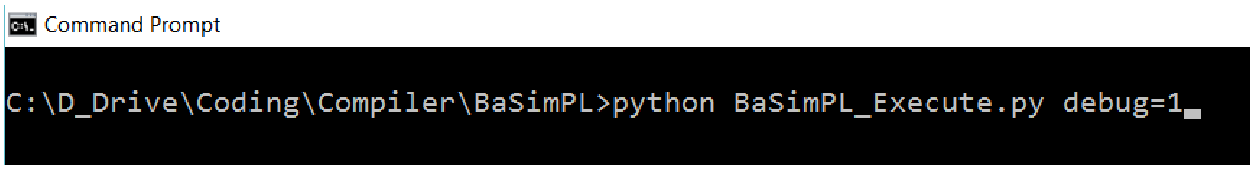
In this case it will just run the test cases that is, it will execute multiple sample programs without debug mode.



```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py
Execute the sample program with debug mode OFF


=========================================================================================================================================
==========INPUTFILE: factorial_iterative.smpl=======DEBUG MODE: OFF==================INTERMEDIATEFILE:factorial_iterative.bspl===========
=========================================================================================================================================


--------------------------------------GENERATING INTERMEDIATE CODE-----------------------------------
---------------------------------------EXECUTING BYTE CODES------------------------------------------
3
5
ENTER INPUT FOR VARIABLE:
```

Python BaSimPL_Execute.py debug=1

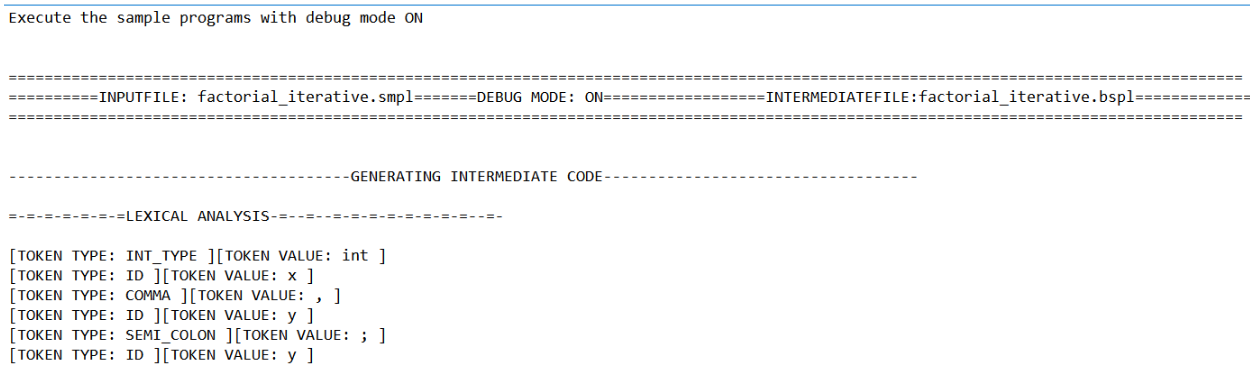
In this case, all the tests will be performed with debug mode ON. So you will get lot of information like intermediate code generated, tokens generated, runtime informations.



Command Prompt

```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py debug=1
```

Tokens generated –



```
Execute the sample programs with debug mode ON


========================================================================================================================================
==========INPUTFILE: factorial_iterative.smpl=======DEBUG MODE: ON==================INTERMEDIATEFILE:factorial_iterative.bspl============
========================================================================================================================================


-------------------------------------GENERATING INTERMEDIATE CODE-----------------------------------

=-=-=-=-=-=-=LEXICAL ANALYSIS-=--=--=-=-=-=-=-=-=-=--=-

[TOKEN TYPE: INT_TYPE ][TOKEN VALUE: int ]
[TOKEN TYPE: ID ][TOKEN VALUE: x ]
[TOKEN TYPE: COMMA ][TOKEN VALUE: , ]
[TOKEN TYPE: ID ][TOKEN VALUE: y ]
[TOKEN TYPE: SEMI_COLON ][TOKEN VALUE: ; ]
[TOKEN TYPE: ID ][TOKEN VALUE: y ]
```

Intermediate Code generated

```
[TOKEN TYPE: SEMI_COLON ][TOKEN VALUE: ; ]
[TOKEN TYPE: SEG_CLOSE ][TOKEN VALUE: } ]

=-=-=-=-=-=-=PARSING-=--=-=-=-=-=-=-=-=-=-=-=-

ALOC GLB.x
ALOC GLB.y
MOVI D0,3
PUSH D0
LEA A0,1(PC)
POP D0
MOV (A0),D0
MOVI D0,5
PUSH D0
LEA A0,0(PC)
POP D0
MOV (A0),D0
FUNCT_BEGIN_Factorial_Iterative:
ALOC LCB.n
LEA A0,13(FP)
POP D0
MOV (A0),D0
ALOC LCB.x
ALOC LCB.val
```

Runtime information –

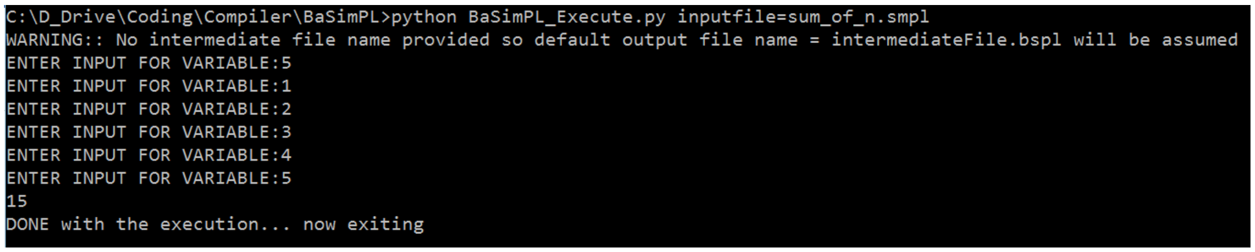
```
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=++DEBUG INFORMATION++=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
D0:     0
D1:     1
A0:     8(FP)
InstructionPointer:     68
Instruction To Be Executed:     PUSH D0
STACK:  []
STACKED RETURN ADDRESS: [134, -999]

GLOBAL VARIABLES:
{}

FUNCTIONS:
{'main': 125, 'hemachandra_fibonacci': 0}

LABELS:
{'IF_END_5': 77, 'ELSE_END_6': 108, 'ELSE_END_4': 109, 'WHILE_BEGIN_2': 29, 'IF_END_3': 59, 'WHILE_END_1': 123}

FRAME POINTERS:
[0, 125]

FRAME ALLOCATIONS:
[{8: 9, 1: 10, 5: 13, 6: 21, 7: 21}, {1: 10}]

GLOBAL STACK VARIABLES:
{}

LOCAL STACK VARIABLES:
[{}, {}]
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
```

python BaSimPL_Execute.py inputfile=sum_of_n.smpl

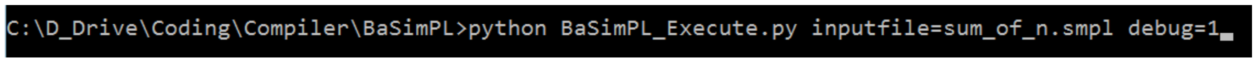
python BaSimPL_Execute.py inputfile=sum_of_n.smpl debug=1

python BaSimPL_Execute.py inputfile=sum_of_n.smpl debug=1 outputfile=sum_of_n.bspl

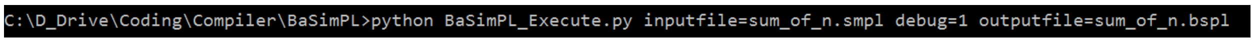
In this we are executing a simple program. Suppose you have written a program in a file sum_of_n.smpl and you want to execute it then use this command. You can enable debug command by adding debug=1 to the above argument. As outputfile is not provided, it will generate byte code in a file called intermediateFile.bspl. If you want to provide the output file name then use outputfile=example.bspl argument to it.



```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py inputfile=sum_of_n.smpl
WARNING:: No intermediate file name provided so default output file name = intermediateFile.bspl will be assumed
ENTER INPUT FOR VARIABLE:5
ENTER INPUT FOR VARIABLE:1
ENTER INPUT FOR VARIABLE:2
ENTER INPUT FOR VARIABLE:3
ENTER INPUT FOR VARIABLE:4
ENTER INPUT FOR VARIABLE:5
15
DONE with the execution... now exiting
```



```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py inputfile=sum_of_n.smpl debug=1
```



```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py inputfile=sum_of_n.smpl debug=1 outputfile=sum_of_n.bspl
```

Source Code

Following are the important source files.

1. BaSimPL_Lexxer.py
   This has classes and functions related to lexical analysis. It uses regular expressions mainly to identifying tokens
2. BaSimPL_Parser.py
3. This has classes and functions for syntax and sematic analysis. In this tokens are parsed to generate the intermediate code.
4. BaSimPL_Execute.py
5. This is the entry point of the program. Based on the argument it will call the interpreter. It also has test functions to test different components.
6. BaSimPL_Compiler.py
   This is our compiler. It uses the objects of lexxer and parser. This finally generates the intermediate byte code file.
7. BaSimPL_DummyRuntime.py
   This is our virtual machine or runtime. It has the structure of our machine. It will execute instructions each line by line.
8. BaSimPL_Interpreter.py
   This utilizes the compiler and runtime to first generate the byte code and finally to execute the generated bytecode.
9. Entry.py
   This represents each entry in the symbol table.
10. SymbolTable.py
    This is our symbol table which has entries based on the program. It is used by the parser while generating the intermediate code.

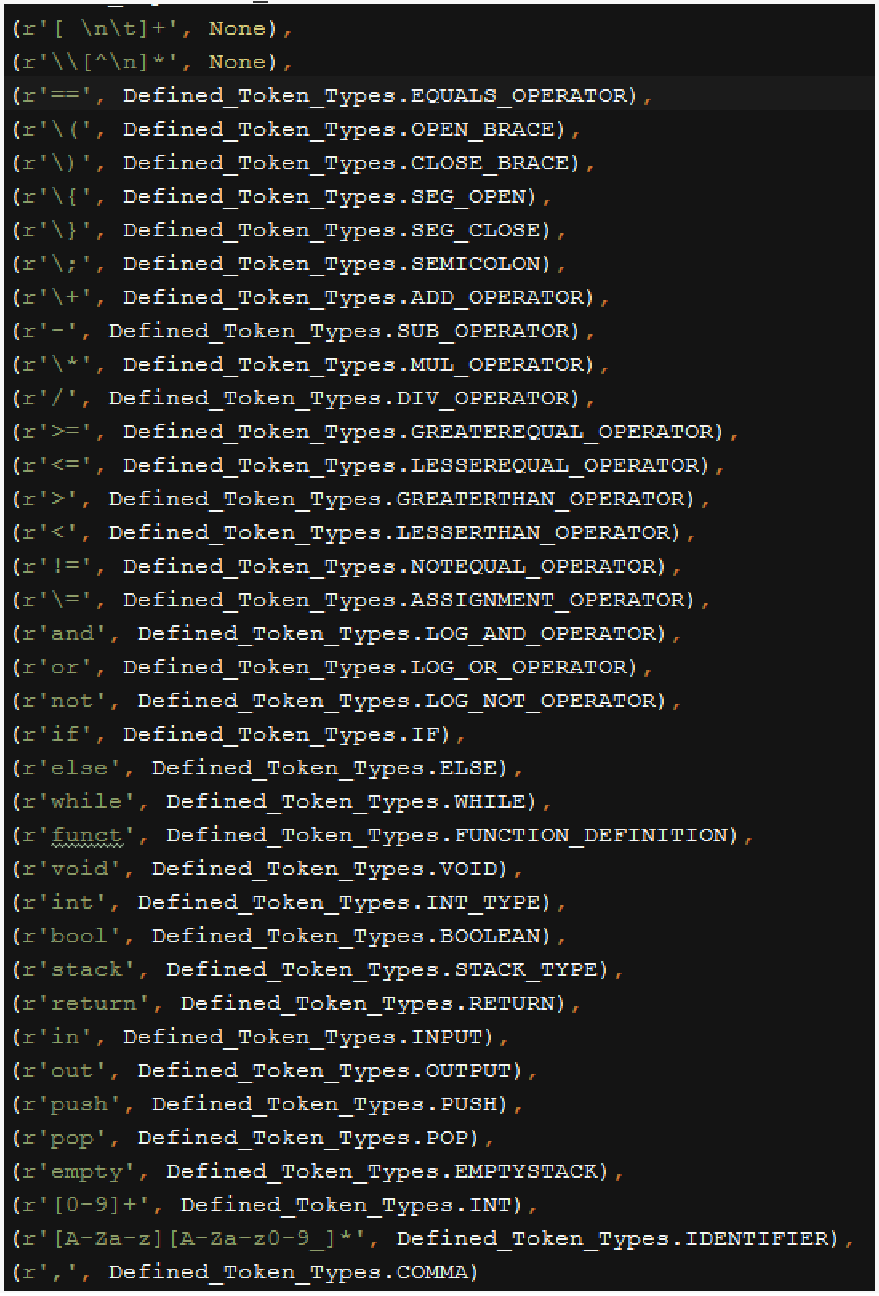
Lexical analysis

This is the list of tokens and its regular expressions –

```
INT = 'INT'

IDENTIFIER = 'ID'

ADD_OPERATOR = 'ADD'

SUB_OPERATOR = 'SUB'

MUL_OPERATOR = 'MUL'

DIV_OPERATOR = 'DIV'

EQUALS_OPERATOR = 'EQUALS'
```

```
ASSIGNMENT_OPERATOR = 'ASSIGN'

GREATERTHAN_OPERATOR = 'GREATER'

LESSERTHAN_OPERATOR = 'LESSER'

GREATEREQUAL_OPERATOR = 'GREATEREQUAL'

LESSEREQUAL_OPERATOR = 'LESSEREQUAL'

NOTEQUAL_OPERATOR = 'NOTEQUAL'

OPEN_BRACE = 'OPEN_BRACE'

CLOSE_BRACE = 'CLOSE_BRACE'

LOG_AND_OPERATOR = 'AND'

LOG_OR_OPERATOR = 'OR'

LOG_NOT_OPERATOR = 'NOT'

IF = 'IF'

ELSE = 'ELSE'

WHILE = 'WHILE'

SEG_OPEN = 'SEG_OPEN'

SEG_CLOSE = 'SEG_CLOSE'

SEMICOLON = 'SEMI_COLON'

COMMA = 'COMMA'

FUNCTION_DEFINITION = 'FUNCT_DEF'

VOID = 'VOID'

INT_TYPE = 'INT_TYPE'

EOF = 'EOF'

RETURN = 'FUN_RETURN'

STACK_TYPE = 'STACK_TYPE'

BOOLEAN = 'BOOLEAN'

EMPTYSTACK = 'STACK_EMPTY'

POP = 'STACK_POP'

PUSH = 'STACK_PUSH'
```

```
INPUT = 'CONSOLE_INPUT'

OUTPUT = 'CONSOLE_OUTPUT'
```



```
(r'[ \n\t]+', None),
(r'\\[^\n]*', None),
(r'==', Defined_Token_Types.EQUALS_OPERATOR),
(r'\(', Defined_Token_Types.OPEN_BRACE),
(r'\)', Defined_Token_Types.CLOSE_BRACE),
(r'\{', Defined_Token_Types.SEG_OPEN),
(r'\}', Defined_Token_Types.SEG_CLOSE),
(r'\;', Defined_Token_Types.SEMICOLON),
(r'\+', Defined_Token_Types.ADD_OPERATOR),
(r'-', Defined_Token_Types.SUB_OPERATOR),
(r'\*', Defined_Token_Types.MUL_OPERATOR),
(r'/', Defined_Token_Types.DIV_OPERATOR),
(r'>=', Defined_Token_Types.GREATEREQUAL_OPERATOR),
(r'<=', Defined_Token_Types.LESSEREQUAL_OPERATOR),
(r'>', Defined_Token_Types.GREATERTHAN_OPERATOR),
(r'<', Defined_Token_Types.LESSERTHAN_OPERATOR),
(r'!=', Defined_Token_Types.NOTEQUAL_OPERATOR),
(r'\=', Defined_Token_Types.ASSIGNMENT_OPERATOR),
(r'and', Defined_Token_Types.LOG_AND_OPERATOR),
(r'or', Defined_Token_Types.LOG_OR_OPERATOR),
(r'not', Defined_Token_Types.LOG_NOT_OPERATOR),
(r'if', Defined_Token_Types.IF),
(r'else', Defined_Token_Types.ELSE),
(r'while', Defined_Token_Types.WHILE),
(r'funct', Defined_Token_Types.FUNCTION_DEFINITION),
(r'void', Defined_Token_Types.VOID),
(r'int', Defined_Token_Types.INT_TYPE),
(r'bool', Defined_Token_Types.BOOLEAN),
(r'stack', Defined_Token_Types.STACK_TYPE),
(r'return', Defined_Token_Types.RETURN),
(r'in', Defined_Token_Types.INPUT),
(r'out', Defined_Token_Types.OUTPUT),
(r'push', Defined_Token_Types.PUSH),
(r'pop', Defined_Token_Types.POP),
(r'empty', Defined_Token_Types.EMPTYSTACK),
(r'[0-9]+', Defined_Token_Types.INT),
(r'[A-Za-z][A-Za-z0-9_]*', Defined_Token_Types.IDENTIFIER),
(r',', Defined_Token_Types.COMMA)
```

Parsing

For parsing we are using recursive decent parsing in a top down way.

Grammar

This is the grammar of our programing language.

program -> topLevelDeclarations*

topLevelDeclaration -> functionDecl | GlobalDataDecl | globalAssignment | globalPushStatement | globalPopStatement | globalStackEmptyStatement

GlobalDecl -> TYPE_DECL declList SEMICOLON

DataDecl -> TYPE_DECL declList SEMI_COLON

functionDecl -> FUNCT_DEF funcReturnType ID OPEN_BRACE PARAMETER_LIST CLOSE_BRACE funBlock

funReturnType -> VOID | INT_TYPE

declList -> ID { COMA ID }

funBlock -> SEG_OPEN declRegion SEQSTATMENTS SEG_CLOSE

declRegion -> DataDecl*

globalAssignment -> ID ASSIGN_OPERATOR INT SEMI_COLON

PARAMETER_LIST -> TYPE_DECL ID { COMA TYPE_DECL ID }

TYPE_DECL -> INT_TYPE | STACK_TYPE

SEQSTATEMENTS -> STATEMENT {STATEMENT}

STATEMENT -> simpleSTATEMENT | compoundSTATEMENT

simpleSTATEMENT -> assignmentSTATEMENT | procedureSTATEMENT | returnStatement | pushStatement | popStatement | stackemptyStatement

returnStatement -> FUN_RETURN ReturnValue SEMI_COLON

ReturnValue -> ID | INT | NULL

compoundSTATEMENT -> ifSTATEMENT | whileSTATEMENT

assignmentSTATEMENT -> ID ASSIGNT_OPERATOR expression SEMI_COLON ifSTATEMENT -> IF OPEN_BRACE expression CLOSE_BRACE SEG_OPEN SEQSTATEMENTS SEG_CLOSE {ELSE SEG_OPEN SEQSTATEMENTS SEG_CLOSE}

whileSTATEMENT -> WHILE OPEN_BRACE expression CLOSE_BRACE SEG_OPEN SEQSTATEMENTS SEG_CLOSE

Expression -> bterm [OR_OPR bterm]*

bterm -> notfactor [AND_OPR notfactor]*

notfactor -> [NOT] bfactor

bfactor -> INT | ID | relation

relation -> simpleExpression { relationalOperator simpleExpression}

simpleExpression -> term { ADDSUB_OPERATOR term}

term -> factor { MULDIV_OPERATOR factor}

factor -> OPEN_BRACE simpleExpression CLOSE_BRACE | ID | INT | procedureSTATEMENT

ADDSUB_OPEATOR -> ADD_OPERATOR | SUB_OPERATOR

MULDIV_OPERATOR -> MUL_OPERATOR | DIV_OPERATOR

relationalOperator -> EQUALS | GREATER | LESSER | GREATEREQUAL | LESSEREQUAL | NOTEQUAL

INT -> [0-9]{[0-9]}

ID -> [a-zA-Z]{[a-zA-Z0-9_]}

OPEN_BRACE -> "("

CLOSE_BRACE -> ")"

SEG_OPEN -> "{"

SEG_CLOSE -> "}"

IF -> "if"

ELSE -> "else"

WHILE -> "while"

ADD_OPERATOR -> "+"

SUB_OPERATOR -> "-"

MUL_OPERATOR -> "*"

DIV_OPERATOR -> "/"

ASSIGNT_OPERATOR -> "=="

EQUALS -> "="

SEMI_COLON -> ";"

FUNCT_DEF -> "funct"

VOID -> "void"

INT_TYPE -> "int"

COMMA -> ","

FUN_RETURN -> "return"

pushStatement -> ID PUSH EXPRESSION SEMICOLON

popStatement -> ID POP ID SEMICOLON

emptyStatement -> ID EMPTY ID SEMICOLON

PUSH -> "push"

POP -> "pop"

EMPTY -> "empty"

globalPushStatement -> ID PUSH EXPRESSION SEMICOLON

globalPopStatement -> ID POP ID SEMICOLON

globalEmptyStatement -> ID EMPTY ID SEMICOLON

Symbol table

Every symbol table has list of entries. Each entry has name, value, type, location, return address, parameter list. Only for functions return address and parameter list will be used. For integer variables type is of INT_VAR where as for stack variable we have type as STACK_VAR.

Apart from entries list, symbol table is linked to previous parent symbol table. Example if I am entering a function, then a new symbol table will be created and that symbol table is linked to global symbol table. By this we are handling multilevel scope blocks.

Interpreter

Runtime/Virtual Machine

Registers:

D0 - General Purpose Register

D1 - General Purpose Register

A0 - Register that holds address

Instructions

MOVI - Used to load an intermediate value to a register. Example if we want to load integer constant 5 to register D0 then we use MOVI D0, 5 . After executing this instruction D0 will have value 5.

PUSH - Used to put value into the stack. Suppose we want to put the value in register D0 to stack then we use PUSH D0.

POP - Used to take a value out of the stack. POP D0, here whatever value was present at the top of the stack will be put in register D0.

LEA - This will load the address to register A0. Suppose we are trying to assign some value to a variable, then that value will be stored in the address allocated to the variable. LEA will help to load the address location of that variable. LEA A0, n(PC)

MOV - This instruction will help to move the value from one register location another another register. Example MOV D0, D1 - This will move the value in D1 to D0. Similarly if we want to move the value in a memory location that is variable to a register also we use MOV. Suppose we want to move the value of variable n to register D0 then, First we need to get the address of the variable n using LEA that is LEA A0, n(PC). Now we use MOV D0, (A0) to move the value to register D0.

CGT - This instruction will be used to check for greater than condition between two registers. CGT D0, D1 - Here if D0 > D1 then value 1 will be stored in register D0, else 0 will be stored in the register D0.

CNE - This instruction will be used to check for not equal to condition between two registers. CNE D0, D1 - Here if D0 is not equal to D1 then value 1 will be stored in register D0, else 0 will be stored in the register D0

CEQ - This instruction will be used to check for equal to condition between two registers. CEQ D0, D1 - Here if D0 is equal to D1 then value 1 will be stored in register D0, else 0 will be stored in the register D0

CGE - This instruction will be used to check for greater than or equal to condition between two registers. CGE D0, D1 - Here if D0 is greater than or equal to D1 then value 1 will be stored in register D0, else 0 will be stored in the register D0.

CLE - This instruction will be used to check for lesser than or equal to condition between two registers. CLE D0,D1 - Here if D0 is lesser than or equal to D1 then value 1 will be stored in register D0, else 0 will be stored in the register D0.

CLT - This instruction will be used to check for lesser than condition between two registers. CLT D0, D1 - Here if D0 is lesser than D1 then value 1 will be stored in register D0, else 0 will be stored in the register D0.

ADD - This instruction will add the values of two registers. ADD D0,D1 - Here we will be adding two registers values D0 and D1 and result will be stored back to D0.

SUB - This instruction will subtract values of two registers. SUB D0,D1 - Here we will be subtraction D0 from D1 that is D0-D1 and result will be stored back to D0.

MUL - This instruction will multiply values of two registers. MUL D0,D1 - Here we will be multiplying D0 and D1 and product will be stored back to D0.

DIV - This instruction will divide values of two registers. DIV D0,D1 - Here D0 will be divided by D1 and quotient will be stored back to D0.

CALL - this instruction is used for calling/executing a function. CALL func1 - here we are calling function func1

BEQ label - This instruction is used for branching. Here if 0 is stored in register D0, then branching will take place and next instruction will be after label. If 1 or any value other than 0 is stored then no branching will occur as a result, next instruction will be immediately after BEQ instruction.

BNE label - This instruction is used for branching. It is opposite of BEQ. If 0 is stored in register D0 then branching will not take placed so next instruction is the instruction which is immediate to the BNE. If 1 or any value other than 0 is stored in D0, then branch will take place and next instruction is the one after label.

JMP label - This instruction is used for branching without condition. When used next instruction will be the instruction following label and not the instruction which is next to JMP instruction.

OR - this will do a logical or of two values. That is D0 || D1. If D0 0 and D1 is 100 then after OR D0, D1 instruction D0 will have 1 as any value other than 0 is considered as logical 1.

AND - this will do a logical and of two values. That is D0 && D1. If D0 0 and D1 is 100 then after AND D0, D1 instruction D0 will have 0. Similarly if D0 100 and D1 is 1 then after operation D1 will have 1 as any value other than 0 is considered as 1.

RET - this will return from function. Before returning it will clear up activation records.

ALOCSTACK - This will allocate the stack either in global namespace or in local namespace.

PUSHST - This will push the element into a stack variable. PUSHST (A0), D0 where D0 has the values to be pushed into the stack. A0 has the address of the stack.

POPST - this will pop the element from the stack variable. POPST (A0), D0 where A0 will have the address of the stack variable. and popped value will reside in D0

EMPTYST - this will check if a stack variable is empty. EMPTYST (A0), D0 where A0 will have the address of the stack variable. and if stack is empty then 1 will be stored in D0 else 0 will be stored in D0.

Interpreter

It is a two phase interpreter. In the first phase it will identify the labels/functions and assigns the addresses to those labels. In the second phase it will execute the actual bytecode. It has a machine stack where we can push or pop numbers during operation. For all the global integer variables, allocation will be done in globalVariables dictionary. Similarly for all the global stack variables, allocation will done in a globalStack dictionary. There is frames stack where for each function call new activation record will be created and pushed into the stack. In that record local integer variables will be allocated. Similarly for each functions we are creating a localStackVariables record and pushed into a separate stack. For every function call a function pointer/environment pointer pointing to the start of the function will be pushed into a framepointer stack. Return address of the caller function will be pushed into a separate ReturnAddress stack. On encountering ret opcode, we are popping all these things and returning back to the normal caller function. We also have an instruction pointer pointing to the current instruction to be executed.

There is a debug flag which will give you detailed information of these things while executing the program.

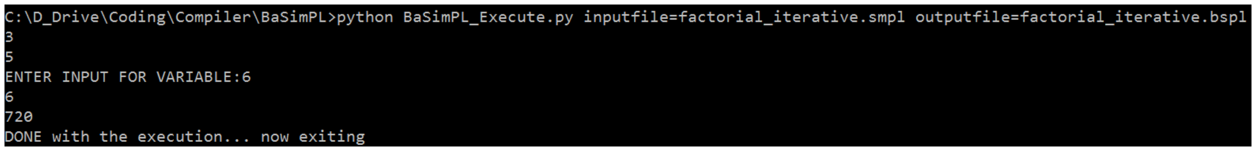
Sample Programs

Factorial_Iterative

This program will calculate the factorial of the given number. Program is present in the file factorial_iterative.smpl . This can be opened in the notepad. Corresponding bytecode can be found in the file factorial_iterative.bspl . To execute this program use this command in the command prompt:

python BaSimPL_Execute.py inputfile=factorial_iterative.smpl outputfile=factorial_iterative.bspl

It will first print 3 followed by 5. Both are global variables. Then it will ask input from the user, value of n whose factorial needs to be found out. Given input will be printed again. It will then print factorial of the value.



```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py inputfile=factorial_iterative.smpl outputfile=factorial_iterative.bspl
3
5
ENTER INPUT FOR VARIABLE:6
6
720
DONE with the execution... now exiting
```

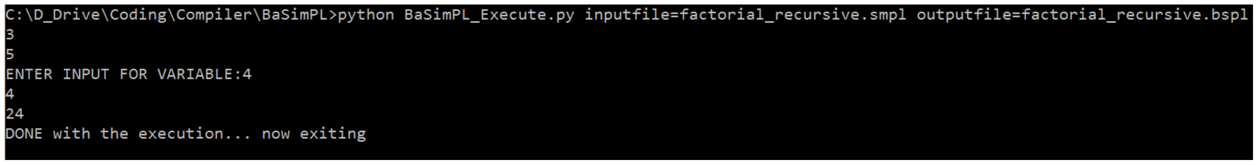
In this program we have demonstrated – global integer variable, local integer variable, Boolean expression, while loop, iterative function execution, parameterized functions, input and output to console.

Factorial_Recursive

This program will calculate the factorial of the given number in a recursive way. Program is present in the file factorial_recursive.smpl . This can be opened in the notepad. Corresponding bytecode can be found in the file factorial_recursive.bspl . To execute this program use this command in the command prompt:

python BaSimPL_Execute.py inputfile=factorial_recursive.smpl outputfile=factorial_recursive.bspl

It will first print 3 followed by 5. Both are global variables. Then it will ask input from the user, value of n whose factorial needs to be found out. Given input will be printed again. It will then print factorial of the value.



```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py inputfile=factorial_recursive.smpl outputfile=factorial_recursive.bspl
3
5
ENTER INPUT FOR VARIABLE:4
4
24
DONE with the execution... now exiting
```

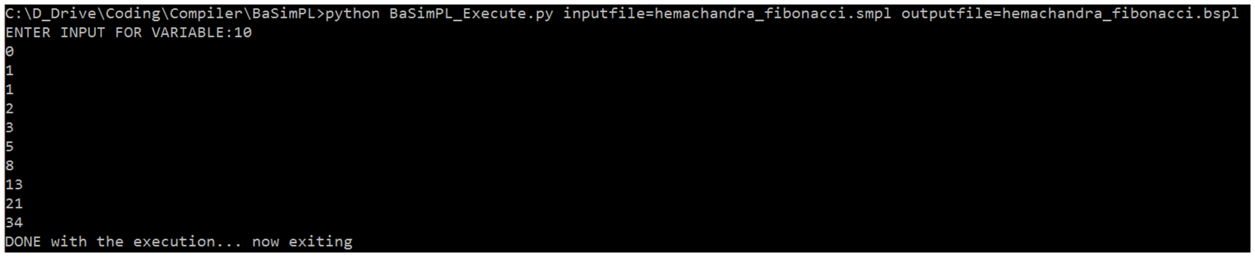
In this program we have demonstrated – global integer variable, local integer variable, Boolean expression, if else constructs, recursive function execution, input and output to console.

Hemachandra_Fibonacci

This program will calculate the Fibonacci numbers. Program is present in hemachandra_fibonacci.smpl. This can be opened in the notepad. Corresponding bytecode can be found in the file hemachandra_fibonacci.bspl. To execute this program use this command in the command prompt:

python BaSimPL_Execute.py inputfile=hemachandra_fibonacci.smpl outputfile=hemachandra_fibonacci.bspl

It will first ask for an integer variable which determines how many Fibonacci numbers needs to be generated. Based on the given input will generate the Fibonacci numbers.



```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py inputfile=hemachandra_fibonacci.smpl outputfile=hemachandra_fibonacci.bspl
ENTER INPUT FOR VARIABLE:10
0
1
1
2
3
5
8
13
21
34
DONE with the execution... now exiting
```

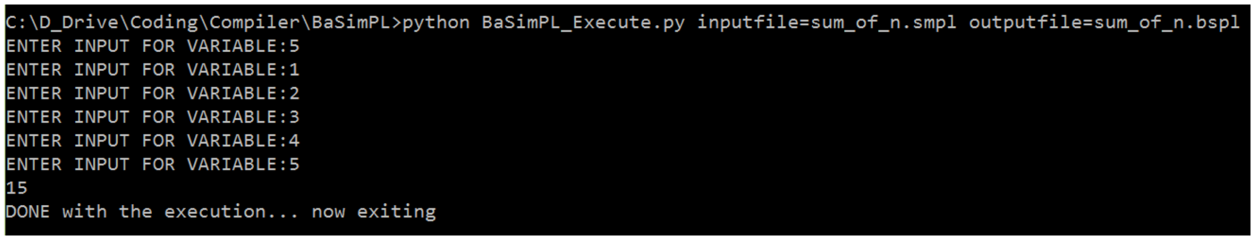
In this we have demonstrated function calls, local variables, if else constructs, in and out, assignment statements, arithmetic expression evaluation.

Sum_Of_N

This program will calculate the sum of n numbers. It uses stack variable. Program can be found in file sum_of_n.smpl . Corresponding bytecode of the program can be found in file sum_of_n.bspl . To execute this program use this command

python BaSimPL_Execute.py inputfile=sum_of_n.smpl outputfile=sum_of_n.bspl

It will first ask an input. This will be your n. Based on the given input it will ask that many input variables. Finally it will give you the sum of those n numbers.



```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py inputfile=sum_of_n.smpl outputfile=sum_of_n.bspl
ENTER INPUT FOR VARIABLE:5
ENTER INPUT FOR VARIABLE:1
ENTER INPUT FOR VARIABLE:2
ENTER INPUT FOR VARIABLE:3
ENTER INPUT FOR VARIABLE:4
ENTER INPUT FOR VARIABLE:5
15
DONE with the execution... now exiting
```

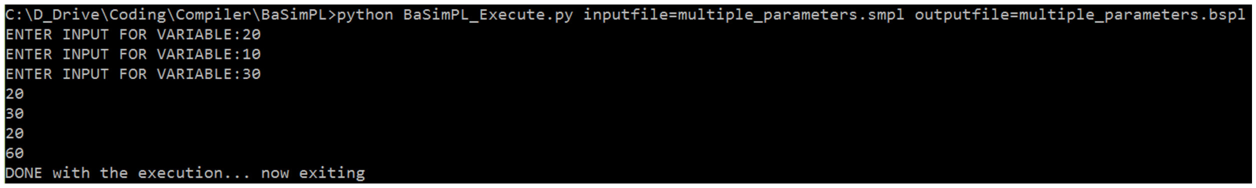
In this we have demonstrated stack variable, local variables, function calls, stack operations, while operations. Relational operators.

Max_Min_Avg_Sum

This program finds the maximum, minimum, average and sum of the given 3 numbers. Program can be found in the file multiple_parameters.smpl . Corresponding bytecode can be found in file multiple_parameters.bspl . To execute the program we can use following command –

python BaSimPL_Execute.py inputfile=multiple_parameters.smpl outputfile=multiple_parameters.bspl

It will first ask three numbers. Then it will show average of 3 numbers, maximum of 3 numbers, minimum of 3 numbers and finally sum of 3 numbers.



```
C:\D_Drive\Coding\Compiler\BaSimPL>python BaSimPL_Execute.py inputfile=multiple_parameters.smpl outputfile=multiple_parameters.bspl
ENTER INPUT FOR VARIABLE:20
ENTER INPUT FOR VARIABLE:10
ENTER INPUT FOR VARIABLE:30
20
30
20
60
DONE with the execution... now exiting
```

In this we have demonstrated function calls. 3 level function calls Main -> Average -> Summation. Multiple parameters to functions: 3 parameters. Logical operations, if else constructs. Arithmetic operations.