American Journal of Applied Sciences, 10 (4): 403-413, 2013
ISSN: 1546-9239

doi:10.3844/ajassp.2013.403.413 Published Online 10 (4) 2013 (http://www.thescipub.com/ajas.toc)

Javascript Interpreter Using Non Recursive Abstract Syntax Tree Based Stack

[1]Sambit Kumar Patra, [2]Binod Kumar Pattanayak and [2]Bhagabat Puthal

[1]Common Engineering Group, MAHINDRA COMVIVA, Bangalore, India
[2]Department of Computer Science and Engineering, ITER, Siksha 'O' Anusandhan University, Bhubaneswar, India

Received 2013-03-05, Revised 2013-04-24; Accepted 2013-05-13

ABSTRACT

In a Mobile device, apart from the battery and memory, the execution time is the key design constraint for executing the scripts of complex and unstructured JavaScript in the web-browser. Abstract Syntax Tree (AST) is a better option for mobile code as it is compiled only once. Due to very recursive nature of the AST, its traversal is going to be inherently recursive. Since use of recursion is out of scope, therefore the ultimate decision would be to emulate the recursive behavior using a set of stacks. We design an algorithm for a non recursive AST based stack, a lightweight interpreter which interprets and evaluates the complex scripts of JavaScript in the allocated time period.

Keywords: Non-recursive Stack for Mobile Device, Script Interpreter, JavaScript Interprete, JavaScript Compiler

1. INTRODUCTION

In the Script Engine architecture, the compiler component generates the AST and Symbol Tables (ST). The interpreter executes the AST tree with reference to ST. The other possible alternative is that the compiler generates the byte-code. Traditional byte-code generation involves 2 stages of compilation. At first, it generates AST and then byte-code from the AST. Many times, it has been observed that by using the jQuery libraries of JavaScript, the scripts are compiled but not executed. Considering the memory limitation of the mobile devices and the limitation of execution, it is preferable to generate AST node and execute as and when required rather than converting all AST nodes to respective byte-codes. However AST node is recursive in nature which can block the high priority mobile management operations such are "CALL" and "SMS".

In this study we have designed the non-recursive AST based stack algorithm to interpret the JavaScript in a predefined time period with asynchronous manner. The data structure of the algorithm has been defined in Data Structure section. In System Architecture section we define interpreter architecture. The evaluations of AST from the instruction stack are evaluated in Algorithm section. The detail asynchronous behaviours are discussed in Asynchronous Behaviour section. In evaluation section we have verified our algorithm with test scripts of ECMA objects from OMA-ESMP test cases (Open Mobile Alliance-ECMA Script Mobile Profile. We have also ported the script engine with devices (a) Moto RAZR v3 (brew 3.15), (b) Qtopia (Linux OS), (c) Samsung (Windows) and (d) Nokia Series (Symbian OS).

A lot of research works have been conducted relating to interpreters. Ortiz (2008) presents S-expression Interpreter Framework (SIF) based on the interpreter design pattern and written in the Ruby programming language in order for language design and implementation, which can be used for demonstration of advanced language concepts and various programming styles. A comparison of two versions of an interpreter for Java programming language is performed in the study of Hills *et al*. (2011), where the authors chose the versions such as visitor pattern and interpreter pattern and the comparison is carried out with respect to maintenance and execution efficiency of implementation of Java programming language. Design of an interpreter with a virtual hardware management facility is detailed by Diessel and Malik (2002), which overcomes the Field-

Corresponding Author: Sambit Kumar Patra, Common Engineering Group, MAHINDRA COMVIVA, Bangalore, India

Programmable Gate Array (FPGA) resource limitations and enables implementation of large systems with small FPGA chips. Design and implementation of a query language interpreter with object oriented specification for bibliographic information retrieval is presented by Fisl *et al*. (1998) that uses an internet client application in Java programming language. Effect of mis-predictions during execution of the indirect branch instructions on an interpreter is addressed by Wien (2003). Effects of "recursive make" related to UNIX related programs are discussed by Miller (2008). Strotz and Wold (1960) provide a synthesis of recursive vs. non-recursive systems with respect to interpretability of a parameter. Typed Command Language (TyCL) an implementation of the Tcl language that is aimed at producing better results during compilation, is presented by Buss (2011). A debuggable interpreter design pattern is included in the work of Vrany and Bergel (2009) that specifies the coexistence of multiple debuggers in order to accept new debugging operations and at the same time being easy to implement. The calculational design of a generic abstract interpreter for a simple imperative language is detailed by Cousot (1999).

2. PROBLEM DEFINATION

The objective of the script interpreter is to design the JavaScript interpreter for embedded devices as per the European Computer Manufacturers Association (ECMA) by reducing memory consumption, reducing CPU cycle consumption, generic in nature, executing in an allocated time period and ease of portability to any devices. JavaScript is cross platform, object oriented, lightweight and standalone. The choice of stack-based interpretation comes not out of choice but out of compulsion. For a typical platform like feature phone where stack size and memory available are low, features like recursion are proscribed. Since AST based interpretation is chosen, due to very recursive nature of the AST, its traversal is going to be inherently recursive. But since use of recursion is out of scope, therefore the ultimate decision would be to emulate the recursive behavior using a set of stack.

The idea is to emulate the way recursion really works in the existing machine architectures. It involves usage of a Runtime Stack in the Data Segment. The Runtime Stack consists of Stack Frames where each stack frame refers to a function call. Similar behavior has to be emulated in the form of a stack using linked-list; we can use the same name Runtime Stack for this.

Again, traversing the AST will be a typical post-order traversal, which also must be implemented without recursion, for which we may use a stack, which we will call as Instruction Stack. At the same time, we need to save the Environment or say Execution Context in typical compiler language, which gives the current state of the interpretation and other details. As we move from one execution context to other, we may require to push them one after the other in a stack called Execution Stack, so that we can come back to the previous execution context with a mere pop.

3. SYSTEM ARCHITECTURE

The Script Interpreter's typical states and the transitions between states are represented in **Fig. 1**. The Script Engine Controller invokes the script interpreter on request from the consumer. The script interpretation occurs in the context of the consumer. Script Interpretation occurs on need basis, i.e., as and when the consumer need to invoke the script interpreter is invoked and the script is executed in the form of interpretation. The word interpretation assumes that the script is already compiled, but that may not often be the case. There will be instances where the interpreter has to invoke the script compiler to compile the scripts and then interpret.

The various states of the interpreter are:

3.1. Uninitialized

The Script Engine is yet to initialize this Component. This is when the consumer is yet to make a request to the Script Engine.

3.2. Initialized

The Script Engine initializes the script interpreter component by an 'initialize' call. The interpreter gets initialized along with its sub components. The precondition is that the script compiler should be initialized. Initialization mainly refers to the allocation of various resources such as memory, coupling (registration) among various components.

3.3. Interpreting

The script interpreter is invoked by an 'interpret AST' call by the Script Engine. So interpreter needs to interpret a script function. The function may be an internal one (within the script interpreter context) or it can be an external one (within consumer context).

3.4. Connected to Consumer

This is the most important state in the script interpreter State transition scenario. This state is a resultant of a 'connect' call from the Script Engine, where the SIP runs in the context of the Consumer, or typically executing a functionality defined/stated by the consumer.

3.5. Suspended

For a typical phone environment, this is a state ought to be considered. The Consumer via Script Engine forces the SIP with a 'Suspend' call, when the consumer itself goes for a Suspension state. At this point of time, the SIP saves the current state in persistent memory and remains suspended until resumed further with a 'resume' call to go back to its previous state.

3.6. Disconnected

The SIP is out of the context of the Consumer with a 'Disconnect' call. Now the SIP can either go to initialized state with an 'operation over' call or to a stopped state with a 'Reset' call.

3.7. De Initialized

This is the end state of the SIP. Essentially the SIP is de-initialized at this point. De-initialization would mean freeing up resources, decoupling.

3.8. Algorithm

The Script Interpreter (SIP) is a component of the Script Engine. The main function of SIP is as follows:

- Interprets the Abstract Syntax Tree (AST) generated by the Script Compiler, using following operations:
  - Non recursively Traverses the IST (Interpretive Syntax Tree (AST+ST)) in appropriate order
  - Evaluate the AST Nodes/Sub trees using a stack in synchronization with the Symbol table and Scope information
  - Fires execution commands for the Consumer
- Handles event from the Consumer
- The Script Interpreter works with the Interpretive Syntax tree i.e., we can say annotated Abstract Syntax Tree (AST) with Symbol Table (ST) information. The IST is optimized for efficient traversal while interpretation and is perfectly semantically checked

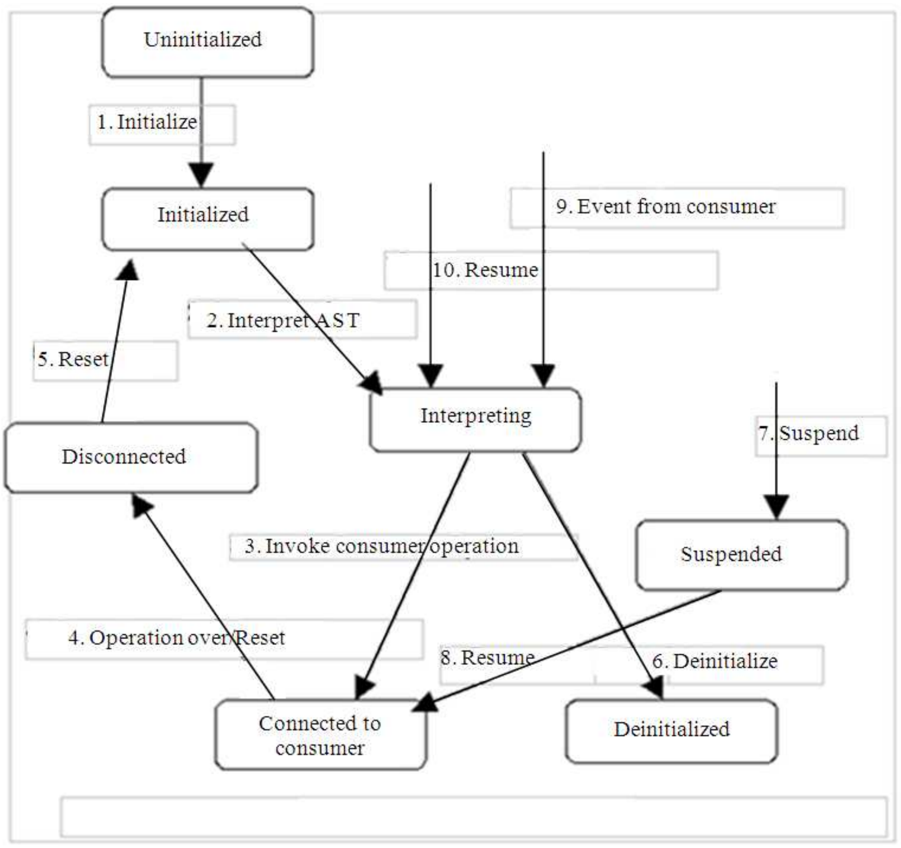

Fig. 1. Script interpret state diagram

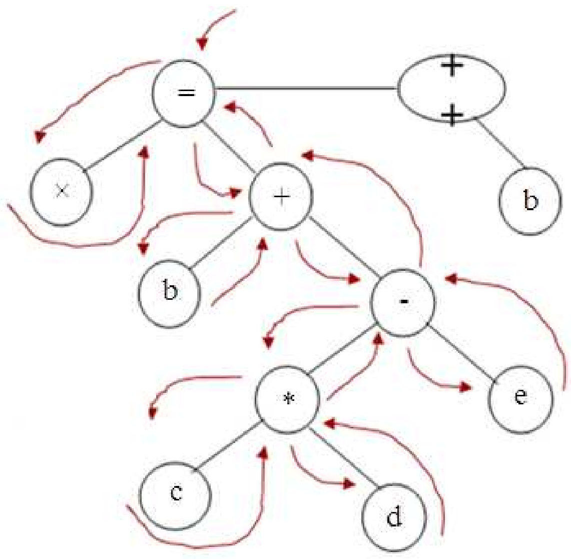

Fig. 2. A typical AST and its traversal

As the stack-based non-recursive interpretation method is chosen for the AST traversal, the AST traversal method plays a crucial role during the interpretation. For simple arithmetic expressions, normally the post order method is preferred, where the rule is to visit the root node at the end, after visiting left node followed by the right. Post-order traversal holds good as long as it's a simple arithmetic operation, but when it comes to the AST of a structured programming language like Java Script with so many programming constructs such as for, while, try-catch, a normal post-order traversal won't fit to the purpose. Hence, a modified post-order traversal method has been conceived. Here we describe about the way the AST is traversed for different types of programming constructs.

Consider the following expression:

x = b + (c * d) – e ; b++

From **Fig. 2**, the two statements form a statement list, one starting with the first node with element "=" and the next node with element "++". Clearly "=" being a binary operator contains two children "x" and "+" and "++" being a unary node contains a single child "b". Similarly "+" is a binary operator containing two children "b" and "-", "-" is binary operator containing two children "*" and "e" and "*" is also a binary operator containing two children "c" and "d".

For a traversal, a stack called "Instruction Stack" is used. This single stack is responsible for holding the temporary AST nodes that are pushed and also the evaluated result node.

Table 1 shows a simple expression evaluation using an Instruction Stack (IS) to hold the nodes and temporary results. The evaluation logic depends on the type of programming construct being evaluated. Instruction Stack is the key to the execution of the traversal; keeping the information about the way a node is pushed onto the Instruction Stack helps in the correct evaluation of the AST. More constructs and their traversal methods are discussed later.

3.9. The Instruction Stack

Each of the elements in the instruction stack is represented in **Fig. 3**. The entries of each element are the AST Node and the way it's pushed to the stack. The way of pushing is important from the AST traversal point of view. In order to facilitate the C Array implementation, the stack top is chosen to be equal to 0 in the beginning. On pushing, the stack top is incremented by one; making the first element corresponding to zero-th element in the Array. The Max Stack Size can be configured depending on the width of a typical expression.

The structure of Instruction stack is as follows:

```
typedef struct _st_instruction_stack
{
        AST                 *ASTNode;
        short int           uhPushMethod ;
}INST_STACK[MAX_STACK_SIZE] ;
```

The AST root node of the script block is first pushed to the instruction stack before the traversal. The traversal algorithm construct is as follows:

```
PROCEDURE    START_INTERPRETE
                        (INST_STACK   *IS,  TIME
TIME_DURATION)
{
        START_TIME = GET_THE_SYSTEM_TIME
( ) ;
                while (hTop>0&&TIME_DURATION
> 0 )
        {
                END_TIME                      =
GET_THE_SYSTEM_TIME () ;
                STACK_POP ( IS,  &ast, &pushType)
;
                N_NODE= GET_NEXT_NODE(ast);
                If  (IS_LIST_TYPE(  ast->eNodeType
))
                {
                STACK_PUSH(IS,           N_NODE,
E_PUSH_AS_ROOT);
                }
                If(              IS_LEAF_NODE(ast-
>eNodeType))
                {
        PROCESS_LEAF_NODE (ast, pushType);
```

Table 1. Evaluation of expression using instruction stack

	Table column head	
AST node	Popped result stack	Push with description
(=)	=	“=” is initially pushed to the IS.
(=)--(++)	x (As Left)	On popping “=”, its observed that the
/	= (As Root)	next statement to it is need to executed, after the
/	++(As Root)	execution of “=”. Hence “++” is pushed as Root.
(x)		Now the node under consideration i.e. “=”
		is having a left child. Upon seeing the
		left child, the left child is pushed as
		Left after pushing the “=” node again back to the IS.
(=)--(++)	+ (As Right)	On popping “x” its observed that it’s a leaf node,
/ \	X (As Left)	which means that the node is a left hand
/ \	= (As Root)	expression for its parent. Now when such
(x) (+)	++(As Root)	a Left Leaf is popped, the IS is peeped to get its
		parent. If the parent is having a right child
		(It MUST have) and if it’s a non
		leaf then it’s pushed to the IS,
		after the node under consideration “x” is pushed again.
(=)--(++)	b (As Left)	Similar to Step 2
/ \	+ (As Right)	
/ \	X (As Left)	
(x) (+)	= (As Root)	
/	++(As Root)	
/		
(b)		
(=)--(++) B	- (As Right)	Similar to Step 3
/ \	b (As Left)	
/ \	+ (As Right)	
(x) (+)	X (As Left)	
/ \	= (As Root)	
/ \	++(As Root)	
(b) (-)		
(=)--(++)	* (As Left)	Similar to Step 2
/ \	- (As Right)	
/ \	b (As Left)	
(x) (+)	+ (As Right)	
/ \	X (As Left)	
/ \	= (As Root)	
(b) (-)	++(As Root)	
/		
/		
(*)		
(=)--(++)	c (As Left)	Similar to Step 2
/ \	* (As Left)	
/ \	- (As Right)	
(x) (+)	b (As Left)	
/ \	+(As Right)	
/ \	X (As Left)	
(b) (-)	= (As Root)	
/	++(As Root)	
/		
(*)		
/ \		
/ \		

Table 1. Continue

(c) (d)		
(=)----C	c*d(As Left)	Here's a deviation to Step 3, since its found that c's
(++)	-	sibling i.e. the right child “d” is a leaf node.
/ \	(As Right)	Now “c” is already evaluated as a leaf node,
/ \	b (As Left)	and we have got its sibling as a leaf node as well.
(x) (+)	+(As Right)	Hence the operation “*” (c and d’s parent)
/ \	X (As Left)	is to be carried out using these
/ \	= (As Root)	two leaf nodes “c” and “d”.
(b) (-)	++(As Root)	
/ \		
/ \		
(c*d) (e)		
		So c is popped, followed by its parent “*”
		and multiplication operation is carried
		out on left leaf node “c” and “*”’s right node “d”.
		The result c*d is formed as leaf node is
		again pushed the way “*”
		was pushed i.e. as a Left
(=)---(++) c*d	c*d-e (As Right)	Similar to Step 8
/ \	b (As Left)	
/ \	+ (As Right)	
(x) (+)	X (As Left)	
/ \	= (As Root)	
/ \	++(As Root)	
/ \		
(b) (c*d-e)		
(=)----(++) c*d-e	b+c*d-e (As Right)	Here the node “c*d-e” popped is a Right
/ \	x (As Left)	leaf node, if this is the case then its parent must
/ \	= (As Root)	be a left leaf node, here in this case it’s “b”,
(x)	++(As Root)	which is popped as the left hand expression,
(b+c*d-e)		now again the parent (which must be a binary
		operator) is popped. In this case its “+”.
		Hence addition operation is done with
		one operand as the left leaf node “b”
		and the other the right leaf node “c*d-e”.
		The result b+c*d-e is pushed the
		way “+” was pushed
(++)	b+c*d-e x’ (As Root)	Similar to Step A
	++(As Root)	Here the operation “=” is carried out
		using x as the left hand Expression,
		and b+c*d-e as the right hand expression.
		The result x’
		(whatever assigned) is pushed again
		to IS the way “=” was pushed
(++)	x’++(As Root)	The node popped was pushed as Root
		and it’s a leaf node, so its ignored.
		…..
		Repeat Step 1

```
}
        else
        {
            PROCESS_NON_LEAF_NODE (ast);
        }
        TIME_DURATION := TIME_DURATION – (END_TIME – START_TIME);
```

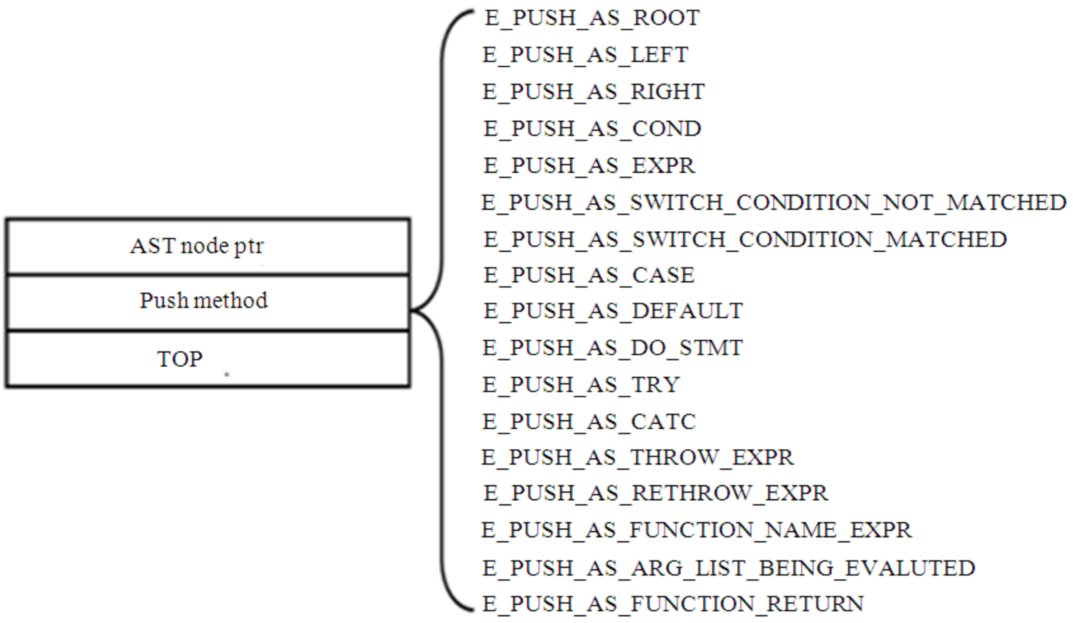

Fig. 3. Instruction stack elements

```
        }
}
PROCEDURE IS_LIST_TYPE (E_AST_TYPE eNodeType)
{
        If (eNodeType == E_CONSTLIST || eNodeType == E_UNARYLIST || eNodeType == E_BINARYLIST || eNodeType == E_TERNARYLIST) {
                return TRUE ;
        }
        return FALSE ;
}
PROCEDURE IS_LEAF (E_AST_TYPE
        ENODETYPE
{
        Switch (ENODETYPE)
        {
                case E_ CONST :
                case E_NUMBER:
                case E_STRING:
                case E_RESULT:
                case E_REG_EXP : return TRUE;
                default: return FALSE ;
        }
}
PROCEDURE STACK_PUSH
                (AST *ast, E_PUSH_TYPE pushType)
{
        If(hTop < MAX_STACK_SIZE)
        {
                IS->ast = ast ;
                IS-> uhPushMethod = pushType;
                hTop++ ;
        }
        else {
                sipError();
                exit (1);
        }
}
PROCEDURE STACK_POP
                (AST **ast, E_PUSH_TYPE *pushType)
{
        If(hTop > -1)
        {
                *ast = IS->ast ;
                *pushType = IS-> uhPushMethod ;
                hTop-- ;
        }
        else {
                sipError();
                exit (1);
        }
}
PROCEDURE STACK_PEEP
```

```
                    (AST **ast, E_PUSH_TYPE
*pushType)
{
        If(hTop  > -1)
{
                *ast = IS->ast;
                *pushType = IS-> uhPushMethod
        }
        else {
                sipError();
                exit (1);
        }
}
PROCEDURE PROCESS_NON_LEAFNODE
                    (AST                ASTNODE,
E_AST_TYPE  ASTTYPE)
{
        switch (ASTTYPE)
        {
                case
E_AST_SWITCH:SWITCH_NODE (ASTNODE) ;
                case                E_AST_TRY
        :TRY_NODE(ASTNODE) ;
                case                E_AST_WITH
        :WITH_NODE(ASTNODE) ;
                case                E_AST_FORIN
        :FORIN_NODE(ASTNODE) ;
                case            E_AST_FUNCTION:
FUNCTION_LIT(ASTNODE) ;            case
E_ATYPE_CATCH:
                case E_ATYPE_REG_EXP          :
                case E_ATYPE_DOT_OPR          :
                default:
                {
                    if                          (
IS_BINARY(ENODETYPE))
                    {

        PROCESS_BINARY_NODE (ASTNODE);
                    }
                    else            if          (
IS_AST_UNARY_TYPE(ENODETYPE))
                    {

        PROCESS_UNARY_NODE (ASTNODE);
                    }
                    else)
                    {
        PROCESS_TERNARY (ASTNODE ) ;
                    }
                }
        }
}
```

```
PROCEDURE PROCESS_LEAF_NODE
        (      AST      *ast      ,      E_PUSH_TYPE
HOW_IT_WAS_PUSHED )
{
        switch (HOW_IT_WAS_PUSHED)
        {
                case
E_PUSH_AS_FUNCTION_EXPR_RESOLVED:
                        FUNCTION_EXPR (  ast ) ;
                break ;
        case
E_PUSH_AS_FUNCTION_NAME_EXPR:
                        FUNCTION_NAME_EXPR(
ast  ) ;
                break ;
        case E_PUSH_AS_LEFT:
        PROCESS_LEFT_LEAF_NODE ( ast ) ;
                break ;
        case E_PUSH_AS_RIGHT:

        PROCESS_RIGHT_LEAF_NODE ( ast ) ;
                break ;
        case E_PUSH_AS_COND:
                        CONDITION_EXPR( ast ) ;
                break ;
        case E_PUSH_AS_ROOT:
                        RETURN_VALUE        (ast-
>pbBranch);
        -
-
default:
        }
}
```

3.10. Processing a Left Leaf Node

```
PROCEDURE PROCESS_LEFT_LEAF_NODE
                                        (AST
*LEFT_NODE);
{
        STACK_PEEP              (&PARENT_NODE,
&PARENT_PUSHED);
        STACK_GET_RIGHT_CHILD
                        (&PARENT_NODE,
&RIGHT_NODE);
        If      (IS_PARENT_NODE_BINARY_TYPE
(PARENT_NODE))
 {
                If                  (IS_LEAF_NODE
(&RIGHT_NODE))
                {
                 OPERATE          (PARENT_NODE-
>eNodeType,
                    LEFT_NODE,
                    RIGHT_NODE,
                    &RESULT_NODE);
            STACK_PUSH (RESULT_NODE,
PARENT_PUSHED);
            }else
            {
              STACK_PUSH (LEFT_NODE,
PUSH_AS_LEFT) ;
              STACK_PUSH (RIGHT_NODE,
PUSH_AS_RIGHT);
            }
    }else{
            -
            -
    }
}
```

On Popping the left leaf node, its parent is checked. Note that we can assert that its parent must be a non-block binary node. Now we have to check if the parent is having a right child or not. If the right child is a leaf one, then both left and right nodes are ready to be evaluated and are the two operands for the parent operator. In that case, the parent is popped and the operation is performed using two operands i.e. left-leaf and right-leaf nodes. The result of the operation is again pushed to the stack in the way the parent is popped. If the right child is a non-leaf node, then the current left leaf node is pushed again and the right child is pushed as PUSH_AS_RIGHT.

3.11. Processing Right Leaf Node

```
PROCEDURE PROCESS_RIGHT_LEAF_NODE
            (AST *LEFT_NODE);
{
    STACK_POP (&LEFT_NODE,
&LEFT_PUSHED);
STACK_POP (&PARENT_NODE,
&PARENT_PUSHED);
If
(IS_PARENT_NODE_BINARY_TYPE(PARENT_NO
DE))
{
     OPERATE(PARENT_NODE->eNodeType,
    LEFT_NODE,
    RIGHT_NODE,
    &RESULT_NODE);
            STACK_PUSH (RESULT_NODE,
PARENT_PUSHED);
    }else{
            -
            -
            }
}
```

On popping the right leaf node, we can assert that it must have been pushed after pushing its left sibling. In that way, the stack order must be such that the operator node is on the top of it followed by the right node on top of the left node. Once the right leaf node is popped, we can simply perform two pops to get the left leaf sibling and its parent operator node respectively. Now, the operation can be performed and the result is pushed the way the parent operator node was popped.

3.12. Asynchronous Behavior of Script Interpretation

Case 1: From a feature-phone's perspective, the execution of the Java Script by the script engine cannot be blocking. It must work in a suspend-resume manner. Suspension of the execution might come when a high priority task like a phone-call has to be addressed and hence, suspending the execution process at some point say 'X'. When the control is reverted back to the script engine application, it has to resume from the point X where we suspended.

Case 2: Not only in the case of priority tasks, but also in case of long loops or say infinite loops in the script, we need to suspend the application because execution cannot go infinitely as it will exhaust battery power and other resources. So identification of such infinite loops is critical to the smooth interpretation of the script.

As we discussed in the previous section that script interpretation involves traversing an AST in a non-recursive manner using a set of logical stacks like Execution Context Stack, Instruction Stack. The stack information form a logical context for a given script execution.

For case 1, the logical context has to be saved when the execution undergoes suspension and it has to be retrieved on resumption. Saving the stack information will give the advantage in terms of knowing the current AST node that was under execution just before suspension. That's clearly the top of the stack. So resumption of the interpretation will follow naturally since the next node information that is to be processed is already in the stack.

For case 2, every time, after execution of one instruction form the stack, will be compared with the allocated time period. If the duration of time is less, it will continue for the next instruction, else will suspend the execution and wait for the next time interval.

Table 2. Instruction push type

Push Type	Syntax
E_PUSH_AS_SWITCH_CONDITION	Switch
E_PUSH_AS_SWITCH_CASE_BODY	Switch-
CaseE_PUSH_AS_CASE_CONDITION	Case with Condition
E_PUSH_AS_SORT	Sort
E_PUSH_AS_DO_STMT	Do-
StatementE_PUSH_AS_TRY	Try
E_PUSH_AS_CATCH	Catch
E_PUSH_AS_THROW_EXPR	Throw-Expression
E_PUSH_AS_RETHROW_EXPR	Rethrow-Expression
E_PUSH_AS_REPLACE	Replace
E_PUSH_AS_UNARY_EXPR	Unary-Expression
E_PUSH_AS_FUNC_NAME_EXPR	Function-Name-Exp
E_PUSH_AS_FUNC_EXPR_RESOLVED	Function-Expr
E_PUSH_AS_ARG_LIST_EVALUTED	Function-Argument
E_PUSH_AS_RETURN	Return Statement
E_PUSH_AS_NEW_CALL	New Call
E_PUSH_AS_WITH	With Statement
E_PUSH_AS_FORIN_LEFT	FOR-IN Statement
E_PUSH_AS_CALLBACK	Callback Funtions
E_PUSH_AS_FORIN_RIGHT	FOR-IN Statement
E_PUSH_AS_MAP	MAP Statement
E_PUSH_AS_LABEL_EXECUTED	Lable Statement
E_PUSH_AS_WITH_ARGUMENT	With Arguments
E_PUSH_AS_EVAL	Eval Statement
E_PUSH_AS_TRY_CATCH	TRY with Catch
E_PUSH_AS_THROW	Throw Statement
E_PUSH_AS_CATCH_PREV	CATCH Statement
E_PUSH_AS_FORIN_NEXT	FOR-INNext
E_PUSH_AS_SET_TIMEOUT_STMT	Set-Time-Out
E_PUSH_AS_SUSPENSION	Suspension
E_PUSH_AS_NEW_AFTER_CALL	New after Call
E_PUSH_AS_TRY_END	Try-End Statement
E_PUSH_AS_LEFT_PRIMITIVE	Left Primitive
E_PUSH_AS_RIGHT_PRIMITIVE	Right Primitive
E_PUSH_AS_DYN_COMP_INLINE	Inline Script
E_PUSH_AS_FOREACH	Foreach
E_PUSH_AS_DEFAULT_RETURN	Default Return
E_PUSH_AS_NEW_RETURN	New Return
E_PUSH_AS_FINALLY	Finally Statement
E_PUSH_AS_EVAL_NODE	eval Statement
E_PUSH_AS_DELETE	Delete Statement
E_PUSH_AS_TYPEOF	Type-of Statement
E_PUSH_AS_CALL	Call Statement
E_PUSH_AS_APPLY	Apply Statement
E_PUSH_AS_LEFT_DOT	DOT-Operator

Table 3. Execution time

Objects in MS	Execution time
Array	610
Date	62
Error	735
Math	562
Reg Exp	1985
String	1265
Total (in MS)	5219

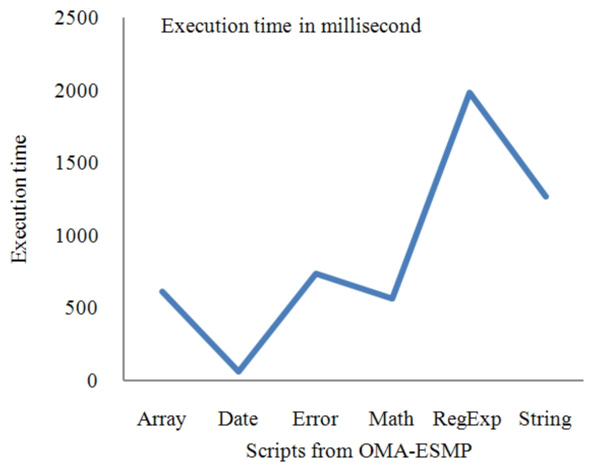

Fig. 4. Scripts Vs execution time

3.13. Evaluation

Expect LEFT, RIGHT, ROOT, as per the ECMA specification, we have added different instruction type as follows (**Table 2**).

We have downloaded the test scripts of ECMA objects from OMA-ESMP test cases (Open Mobile Alliance-ECMA Script Mobile Profile). The evaluation time has been calculated (**Table 3**) considering the interval time of 10 milliseconds and other constraints. **Figure 4** represents the scripts with the respective execution time.

4. CONCLUSION

This study presents a non recursive algorithm for the JavaScript. We have tested and verified this algorithm with top10 Alexa web-sites in different mobile devices. It executes all the scripts of the web-sites without blocking any mobile operation. We have ported, tested and verified our script engine with low end devices such are Moto RAZR v3 (brew 3.15), Qtopia (Linux OS), Samsung (Windows) and Nokia Series (Symbian OS). In future, this can be optimized further and execution time can be reduced further.

5. REFERENCES

Buss, A., 2011. TyCL: An interpreter/compiler of a typed language implementation of Tcl/Tk. Proceedings of the 18th Annual Tcl/Tk Conference, (ATTC' 11).

Cousot, P., 1999. The Calculational Design of a Generic Abstract Interpreter. In: Calculational System Design, Broy, M. and R. Steinbruggen (Eds.), NATO ASI Series F. Amsterdam, IOS Press.

Diessel, O. and U. Malik, 2002. An FPGA interpreter with virtual hardware management. Proceedings of the Abstracts and CD-ROM Parallel and Distributed Processing Symposium, Apr. 15-19, IEEE Xplore Press, pp: 155-162. DOI: 10.1109/IPDPS.2002.1016553

Fisl, I., Z. Konjovic and D. Surla, 1998. Design and implementation of the query language interpreter for bibliographic data retrieval. Novi Sad J. Math., 28: 11-19.

Hills, M., P. Klint, T.V.D. Storm and J. Vinju, 2011. A case of visitor versus interpreter pattern. Proceedings of the 49th International Conference on Objects, Models, Components, Patterns, (ICOCP'11), ACM Press, Springer-Verlag Berlin, Heidelberg, pp: 228-243.

Miller, P., 2008. Recursive make considered harmful. AUUGN J. AUG Inc., 19: 14-25.

Ortiz, A., 2008. language design and implementation using ruby and the interpreter pattern. ACM SIGCSE Bull., 40: 48-52. DOI: 10.1145/1352322.1352155

Strotz, R.H. and H.O.A. Wold, 1960. Recursive vs. nonrecursive systems: An attempt at synthesis. Econometrica, 28: 417-427.

Vrany, J. and A. Bergel, 2009. A debugger for the interpreter design pattern. Software Data Technol., 22: 73-85. DOI: 10.1007/978-3-540-88655-6_6

Wien, T.U., 2003. The structure of efficient interpreters. J. Instruct. Level Parallelism, 5: 1-25.