

Parsing with Earley Virtual Machines

Audrius Šaikūnas

Institute of Mathematics and Informatics
Akademijos 4, LT-08663 Vilnius, Lithuania
Email: tuxmarkv@gmail.com

Abstract—Earley parser is a well-known parsing method used to analyse context-free grammars. While being less efficient in practical contexts than other generalized context-free parsing algorithms, such as GLR, it is also more general. As such it can be used as a foundation to build more complex parsing algorithms.

We present a new, virtual machine based approach to parsing, heavily based on the original Earley parser. We present several variations of the Earley Virtual Machine, each with increasing feature set. The final version of the Earley Virtual Machine is capable of parsing context-free grammars with data-dependant constraints and support grammars with regular right hand sides and regular lookahead.

We show how to translate grammars into virtual machine instruction sequences that are then used by the parsing algorithm. Additionally, we present two methods for constructing shared packed parse forests based on the parse input.

I. INTRODUCTION

Parsing is one of the oldest problems in computer science. Pretty much every compiler ever written has a parser within it. Even in applications, not directly related to computer science or software development, parsers are a common occurrence. Date formats, URL addresses, e-mail addresses, file paths are just a few examples of everyday character strings that have to be parsed before any meaningful computation can be done with them. It is probably harder to come up with everyday application example that doesn't make use of parsing in some way rather than list the ones that do.

Because of the widespread usage of parsers, it no surprise that there are numerous parsing algorithms available. Many consider the parsing problem to be solved, but the reality couldn't be farther from the truth. Most of the existing parsing algorithms have severe limitations, that restrict the use cases of these algorithms. One of the newest C++ programming language compiler implementations, the CLang, doesn't make use of any formal parsing or syntax definition methods and instead use a hand-crafted recursive descent parser. The HTML5 is arguably one of the most important modern standards, as it defines the shape of the internet. Yet the syntax of HTML5 documents is defined by using custom abstract state machines, as none of the more traditional parsing methods are capable of matching closing and opening XML/HTML tags.

It is clear that more flexible and general parsing methods are needed that are capable of parsing more than only context-free grammars.

As such, we present a new approach to parsing: the Earley Virtual Machine, or EVM for short. It is a virtual machine based parser, heavily based on the original Earley parser.

EVM is capable of parsing context-free languages with data-dependant constraints. The core idea behind of EVM is to have two different grammar representations: one is user-friendly and used to define grammars, while the other is used internally during parsing. Therefore, we end up with *grammars* that specify the languages in a user-friendly way, which are then compiled into *grammar programs* that are executed or interpreted by the EVM to match various input sequences.

We present several iterations of EVM:

- EVM₀ that is equivalent to the original Earley parser in it's capabilities.
- EVM₁ is an extension of EVM₀ that enables the use of regular operators within grammar rule definitions, thus easing the development of new grammars.
- EVM₂ further extends EVM₁ by allowing the use of regular-lookahead operator.
- Finally, EVM₃ is an extension of EVM₂ that enables general purpose computation during parsing, and as such allows to conditionally control the parsing process based on the results of previously parsed data. Therefore, EVM₃ can be used to recognize data-dependant language constructs that cannot be parsed by more traditional parsing methods.

In section III we present two separate methods for constructing the abstract syntax trees (or more precisely, shared packed parse forests) based on the input data. Automatic AST construction enables automatic construction of shared packed parse forests, without any changes to the grammar. On the other hand, manual AST construction requires explicit instructions from the user to control the AST construction.

II. EARLEY VIRTUAL MACHINE

A. EVM structure

For every terminal input symbol $input_i$ EVM creates a state S_i . A state S_i is a tuple $\langle input_i, F, FS \rangle$.

Each state S_i contains a set of fibers F . Fibers in EVM loosely correspond to items in Earley parser. Each fiber is a tuple $\langle ip, origin \rangle$. ip is the instruction pointer of the current fiber and $origin$ is the origin state number. The $origin$ value indicates the input offset where the currently parsed rule has begun. In a way, the $origin$ value may be considered as the return "address" for the rule.

Additionally, each state has a set of *suspended* fibers FS . Fibers are suspended when they invoke different rules/non-terminal symbols. A fiber remains suspended until the appropriate non-terminal symbol is matched, at which point the fiber

is resumed by copying it to the set of running fibers F in the current state.

An EVM₀ *grammar* is a set of productions in form $sym \rightarrow body$, where sym is a non-terminal symbol and $body$ is a grammar expression.

An EVM₀ *grammar expression* is defined recursively as:

- a is a terminal grammar expression, where a is a terminal symbol.
- A is a non-terminal grammar expression, where A is a non-terminal symbol.
- ϵ is an epsilon grammar expression.
- (e) is a brace grammar expression, where e is a grammar expression.
- e_1e_2 is a sequence grammar expression, where e_1 and e_2 are grammar expressions.

Every EVM *grammar program* is a tuple $(instrs, rule_map)$. $instrs$ is the sequence of instructions that represents the compiled grammar. $rule_map$ is mapping from non-terminal symbols to locations in the instruction sequence, which represents entry points for the grammar program. It is used to determine the start locations of compiled rules for specific non-terminal symbols.

Every instruction in EVM completes with one of the following results:

- $r_continue$. This result is used to indicate that the current fiber should continue executing instructions in order. That means that after executing an instruction ip of current fiber should be increased by 1.
- $r_continue_to\ ip_{new}$. This behaves exactly like $r_continue$, however ip of current fiber is set to ip_{new} after completing execution of an instruction.
- $r_discard$. This result is used to indicate that the current fiber needs to be terminated.
- $r_suspend$. This result is used to indicate that the current fiber needs to be suspended.

To correctly represent all Earley parser grammars, the following instructions are required:

- $i_call_dyn\ sym$. This instruction is used to begin parsing of non-terminal symbol sym . It dynamically invokes all rules with product sym . This constitutes creating new fibers in state S_{curr} with ip pointing to beginning of every rule with product sym and origin $curr$. The instruction completes with result $r_continue$. Because the fibers of every state are stored in a set, multiple or recursive invocations of the same product have no effect.
- $i_match_sym\ sym_1 \rightarrow ip_1, \dots, sym_n \rightarrow ip_n$. This instruction is used to detect a successful reduction of one or more non-terminal symbols sym_1, \dots, sym_n . When sym_i is reduced, a new fiber is created with ip pointing to corresponding ip_i . The instruction operands essentially form a jump table. Whenever instruction i_match_sym is executed, the current fiber is suspended and moved to the the set of suspended fibers in the current state (it completes with result $r_suspend$).

TABLE I
EVM₀ GRAMMAR COMPILATION RULES

Grammar element	Instruction sequence
Grammar: $G = \{P_1, \dots, P_n\}$	$i_call_dyn\ main$ $i_match_sym\ main \rightarrow l_{accept}$ $l_{accept} :$ i_accept i_stop $code(P_1)$ \dots $code(P_n)$
Production rule: $P \rightarrow e$	$code(e)$ $i_reduce\ P$ i_stop
Terminal grammar expression: a	$i_match_char\ a \rightarrow ip_{next}$
Non-terminal grammar expression (dynamic): A	$i_call_dyn\ A$ $i_match_sym\ A \rightarrow ip_{next}$
Epsilon grammar expression: ϵ	
Brace grammar expression: (e)	$code(e)$
Sequence grammar expression: e_1e_2	$code(e_1)$ $code(e_2)$

- $i_match_char\ char_1 \rightarrow ip_1, \dots, char_n \rightarrow ip_n$. This instruction is used to match terminal symbols. If the current input symbol in_{curr} matches one of the instruction operands $char_i$ then a new fiber $\langle ip_i, origin \rangle$ in state S_{curr+1} is created. After executing this instruction, the current fiber is discarded (it completes with result $r_discard$).
- $i_reduce\ sym$. This instruction performs the reduction of non-terminal symbol sym . It is used to indicate that a grammar rule with product sym has matched successfully. The instruction finds all suspended fibers in state S_{origin} that have been previously suspended with instruction i_match_sym and resumes them in state S_{curr} . Only those fibers are resumed, which have sym among their operands. The fibers are resumed by creating a copy of suspended fiber in the current state with updated instruction pointer. The instruction completes with result $r_continue$.
- i_stop . This instruction stops and discards the current fiber. It is used to destroy the current fiber when a parse rule is matched successfully, usually immediately after executing a i_reduce instruction. The instruction completes with $r_discard$.
- i_accept . This instruction is used to indicate the parse input is valid and can be accepted.

EVM₀ grammar compilation rules are shown in table I. Productions of a grammar are compiled in sequence. $code(E)$ represents instruction sequence for grammar element E , where E may be a grammar, a production rule or a grammar expression. ip_{next} is instruction pointer of the next instruction.

An example EVM₀ grammar and the corresponding grammar program are shown in table II.

```

function EXECUTE_INSTRUCTION( $f$ )
   $\langle ip, origin \rangle \leftarrow f$ 
   $instr \leftarrow instrs_{ip}$ 
  if  $instr = i\_call\_dyn$  then
    return EXEC_CALL_DYN( $f$ )
  else if  $instr = i\_match\_sym$  then
    return EXEC_MATCH_SYM( $f$ )
  else if ... then
    ...
  else
    invalid instruction
  end if
end function

procedure MAIN
   $F_{main} \leftarrow \langle 1, 1 \rangle$ 
  add fiber  $F_{main}$  to  $S_1$ 
  for all  $i \leftarrow 1, input\_length$  do
    if  $F_i = \emptyset$  then
      parse error
    end if
    for all  $f \in F_i$  do
      while EXECUTE_INSTRUCTION( $f$ ) do
        end while
    end for
  end for
end procedure

```

Fig. 1. EVM₀ parser algorithmTABLE II
EVM₀ GRAMMAR COMPILATION EXAMPLE

Grammar	Instruction sequence
	0: i_call_dyn E
	1: i_match_sym E \rightarrow 2
	2: i_accept
	3: i_stop
	4: i_match_char a \rightarrow 5
	5: i_reduce I
	6: i_stop
I \rightarrow a	7: i_call_dyn I
E \rightarrow I	8: i_match_sym I \rightarrow 9
E \rightarrow E + I	9: i_reduce E
	10: i_stop
	11: i_call_dyn E
	12: i_match_sym E \rightarrow 13
	13: i_match_char + \rightarrow 14
	14: i_call_dyn I
	15: i_match_sym I \rightarrow 16
	16: i_reduce E
	17: i_stop

B. Extending EVM to support regular right-hand sides

To support regular right-hand sides in production rules, our definition of a grammar needs to be extended.

An EVM₁ grammar is a set of productions in form $sym \rightarrow body$, where sym is a non-terminal symbol and $body$ is an

TABLE III
EVM₁ GRAMMAR COMPILATION RULES

Grammar element	Instruction sequence
One-or-more grammar expression: e^+	l_{start} : $code(e)$ i_fork l_{start}
Kleene star grammar expression: e^*	l_{start} : i_fork l_{end} $code(e)$ i_br l_{start} l_{end} :
Optional grammar expression: $e?$	i_fork l_{end} $code(e)$ l_{end} :
Alternative grammar expression: $e_1 e_2$	i_fork l_{other} $code(e_1)$ i_br l_{end} l_{other} : $code(e_2)$ l_{end} :

EVM₁ grammar expression.

An EVM₁ grammar expression is defined as follows:

- If e is an EVM₀ grammar expression, it is also an EVM₁ grammar expression.
- e^+ is a one-more-more grammar expression, where e is a grammar expression.
- e^* is a klenne star grammar expression, where e is a grammar expression.
- $e?$ is an optional grammar expression, where e is a grammar expression.
- $e_1|e_2$ is an alternative grammar expression, where e_1 and e_2 are grammar expressions.

To implement these new grammar constructs additional virtual machine instructions are required:

- i_br ip_{new} . This instruction is used to perform an unconditional branch to the given instruction pointer ip_{new} . This instruction completes with result $r_continue_to$ ip_{new} .
- i_fork ip_{new} . This instruction is used to fork the current fiber. After forking, the instruction pointer of the new fiber is set to ip_{new} . This instruction completes with result $r_continue$.

EVM₁ grammar compilation rules are shown in table III. l_{start} , l_{end} and l_{other} are code labels. During grammar compilation these labels are replaced with concrete instruction pointer values.

An example grammar rule that uses regular operator $+$ and its corresponding instruction sequence are shown in table IV.

C. Regular lookahead in EVM

In addition to having regular right-hand sides, sometimes it is helpful to perform regular lookahead during parsing. The EVM₁ may be further augmented to support this feature.

An EVM₂ grammar expression is defined as:

- e , where e is EVM₁ grammar expression.
- $e_1 > e_2$ is a positive lookahead grammar expression, where e_1 and e_2 are grammar expressions.

TABLE IV
EVM₁ GRAMMAR COMPILATION EXAMPLE

Grammar rule	Instruction sequence
B → (A a)+	...
	20: i_call_dyn A
	21: i_match_sym A → 22
	22: i_match_char a → 23
	23: i_fork 20
	24: i_reduce B
	25: i_stop
	...

Grammar expression $A > B$ means that A should only match if it is immediately followed by B .

There are two ways of implementing regular lookahead in the current model of EVM:

- When parsing $A(B > C)D$, EVM can first parse A , then B , then both C and D in parallel. Should it appear during parsing that C fails to match, then the corresponding parse should be rejected.
- When parsing $A(B > C)D$, EVM can first parse A , then B , after which parsing of D is delayed until C matches successfully. Once C matches successfully, parsing of D is resumed.

Both of these approaches have advantages and disadvantages. The first approach is simpler and doesn't require any fundamental changes to the way EVM processes input: one terminal symbol at a time without an ability to backtrack and reparse certain parts of the input. However it also may require storing additional information about fiber relationships and complex logic for discarding fibers resulting of invalid parses. More importantly, this approach starts parsing D preemptively even when it is not known whether or not the lookahead will succeed. As a result, such lookahead implementation may be less efficient.

The second approach requires changes to EVM to allow suspending fibers that depend on lookahead expressions and later resume then in backtracked position. This approach should be more efficient when the lookahead grammar expression fails to match often. We select the latter approach for implementing lookahead in EVM.

Right now, EVM₁ is built under assumption that `i_match_sym` is always executed before corresponding `i_reduce`. However with grammars that contain lookahead this may no longer be true in all cases. Consider grammar expression $(A > B)B$. Initially, non-terminal symbol A is matched, after which the instruction sequence for the lookahead sub-expression B will be executed. It will cause matching of non-terminal symbol B . If B parses successfully, then the whole sub-expression $(A > B)$ will be matched and the original fiber for parsing $(A > B)B$ will be resumed. It will try to parse B again by invoking symbol B with instruction `i_call_dyn`, which will cause no new fibers to be created, as B was already parsed in the lookahead sub-expression. That means that the original thread will be permanently suspended with instruction `i_match_sym` and

the expression $(A > B)B$ will always fail to parse.

To avoid the previously described issue and to support parsing lookahead expressions, EVM can no longer rely on the strict ordering of `i_match_sym` and `i_reduce` instructions. As a result, the following changes to EVM are required:

- EVM₂ in every state has to keep track of reductions that happened and their respective lengths.
- Instruction `i_reduce` has to store in origin state S_{origin} reduction symbol and it's final state index $curr$.
- Instruction `i_match_sym` has to check all of it's operands for the reductions that may have already happened and to resume fibers in final states of the reductions. This may only occur if the matching symbol was already parsed in a lookahead sub-expression.
- Every fiber has to have a priority value, as lookahead sub-expressions have to be executed first.

An EVM₂ state S_i is a tuple $\langle i, R, T, FS \rangle$, where i is the state index, R is the reduction map, T is execution trace set, FS is the suspended fiber set. Because the running fiber set is longer stored stored within the state, it is necessary to ensure that there are no duplicate fibers. This is achieved by using execution trace set T , which stores instruction pointer and origin state index pairs. Whenever a new fiber is to be created, the EVM first checks whether or not such fiber already exists in target state. If it does, then no new fiber is created. This eliminates infinite left-recursion and having to reparse the same input with the same grammar rule multiple times.

An EVM₂ fiber F is a tuple $\langle sid, prio, ip, origin \rangle$, where sid is state index, $prio$ is priority value, ip is instruction pointer and $origin$ is state index of origin (caller/return) state.

An EVM₂ parser is a tuple $\langle input, P, Q, S \rangle$, where $input$ is the input terminal symbol sequence, P is the grammar program, Q is the fiber queue and S is the state sequence. During parsing, fibers are executed according to their priority. Fibers with lower priority are removed from the fiber queue first. If there multiple fibers with the same priority, then the fiber with lowest state index is removed first. Otherwise, the order of execution is unspecified.

To support regular lookahead, additional two instructions are required:

- `i_lookahead ipahead`. This instruction is used to begin parsing of a lookahead sub-expression at instruction pointer $ipahead$. It creates a new fiber $\langle curr, prio - 1, ipahead, curr \rangle$, where $curr$ is the index of the current state, $prio$ is the priority of the current fiber. The instruction completes with result `r_continue`.
- `i_lookahead_ret sym`. This instruction is used to finish parsing lookahead sub-expression. It is identical to `i_reduce` with one key difference: fibers are resumed not in current state, but in origin state. The instruction completes with `r_discard`.

EVM₂ grammar compilation rules are provided in table V. $usym$ is an unique non-terminal symbol created for each lookahead expression.

TABLE V
EVM₂ GRAMMAR COMPILATION RULES

Grammar element	Instruction sequence
Lookahead grammar expression: $e_1 > e_2$	$code(e_1)$ $i_lookahead\ l_{ahead}$ $i_match_sym\ usym \rightarrow l_{end}$ $l_{ahead}:$ $code(e_2)$ $i_lookahead_ret\ usym$ $l_{end}:$

D. Eliminating dynamic rule resolution

In the current model of EVM, rule resolution is performed dynamically during runtime by the `i_call_dyn` instruction. It is possible to eliminate this indirection by replacing every `i_call_dyn prod` instruction with a sequence of `i_call ip` instructions, where every `i_call` instruction invokes a different production rule for the same product *prod*. This also means that the compiled grammar programs are no longer required to keep track of the *rule_map* variable, that was previously used by the `i_call_dyn` instruction.

E. Parsing with data-dependant constraints

It is well known that many languages used in practise cannot be represented purely by using context-free grammars. For example, in order to parse XML sources, an additional automata is needed to match opening and closing XML tags. Other languages have fixed-width fields of width *n*, where *n* is an integer value preceded before the field. Specifying such languages with context-free grammars is impossible as well, as the parser has to semantically recognize the meaning behind the length field and to use that value to continue parsing.

In order for parser like EVM₂ to parse XML, it needs somehow to "remember" the opening XML tags and later match the closing tag only against the remembered string. This can be achieved by further augmenting EVM₂ into EVM₃ by applying the following changes:

- Fibers have to be extended to contain stacks. A fiber's stack may be used to perform general-purpose computation during parsing.
- Execution trace set has to be extended to include fiber's stack. This enables EVM₃ to simultaneously execute multiple fibers with the same instruction pointer and the same origin, but with different data-constraints, which will be stored in the stack.
- `i_call` instruction has to be extended with an operand to contain the number of parameters to copy from the stack of the current fiber to the stack of target fiber.
- The following new categories of instructions are needed: stack instructions, conditional control transfer instructions, data processing instructions.

To support general-purpose computation, EVM₃ grammars need to be extended to include statements and expressions (not to be confused with grammar expressions). In EVM₃ grammar rule definitions look akin to function definitions or subroutines that are found in general-purpose programming

languages: grammar rule definitions are composed out of 0 or more procedural statements. These statements enable to adjust the control flow of the grammar rule.

Expressions, just like in more traditional programming languages, enable manipulation of variable values. Both variable values and intermediate expression values are stored in the stack of the current fiber.

To separate "traditional" value-based expressions from the grammar expressions that are used to specify matching rules, a special **parse** statement is introduced to the grammar language. The **parse** statement may be mixed in with the other grammar statements to more accurately control and constrain the parse process. This enables to direct the parsing process based on the variable values that may have been derived from the parse input.

To support these new features, new EVM instructions are required. Non-exhaustive list of new instructions in EVM₃:

- `i_push_int v`. Pushes an integer value *v* to the stack of the current fiber.
- `i_pop n`. Removes the top *n* values from the stack.
- `i_peek i`. Pushes a copy of stack value with index *i* to the top of the stack.
- `i_poke i`. Removes the top value of the stack and sets the stack value with index *i* to that value.
- `i_bz ipnew`. Conditional control transfer to *ip_{new}*. Removes the top stack value and performs jump to *ip_{new}* if the value is 0.
- `i_match_char_dyn`. Used for data-dependent matching of terminal symbols. Matches the top value of the stack with *input_{curr}*. If the symbols match, it creates a new fiber $\langle curr + 1, prio, ip + 1, origin \rangle$. The instruction completes with result `r_discard`. An optimized version of the instruction may move the fiber from the *S_{curr}* state to *S_{curr+1}* with updated *ip* value.
- `i_foreign_call n idx`. Calls a foreign function with index *idx* and *n* arguments. The call is performed by popping *n* arguments from the stack and passing them to the foreign callee. The result of the callee is pushed back to the stack. The instruction may be used to implement various data processing operations without adding additional data-processing instructions.
- `i_add_int`. Pops two top elements from the stack and pushes their sum to the top of the stack.
- ...

Most of the newly added instructions complete with result `r_continue`, unless stated otherwise. New data processing (integer, string, list handling) instructions may be added as needed. The same operations may be implemented by using `i_foreign_call` instruction to call external functions in the environment that implements EVM.

EVM₃ grammar to grammar program compilation rules are provided in table VI. The table lists only the core elements of the grammar language to illustrate the overall grammar compilation process. New statements and expressions may be added as needed. Additionally, the syntax of the grammar

TABLE VI
EVM₃ GRAMMAR COMPILATION RULES

Grammar element	Instruction sequence
EVM ₃ grammar rule: rule $sym(arg_1, \dots, arg_n)$ $stmt_1$... $stmt_n$ end	$code(stmt_1)$... $code(stmt_n)$ $i_reduce\ sym$ i_stop
If statement: if $cond$ $body$ end	$code(cond)$ $i_bz\ l_{end}$ $code(body)$ $l_{end}:$
Parse statement: parse $grammar_expr$	$code(grammar_expr)$
While statement: while $cond$ $body$ end	$l_{start}:$ $code(cond)$ $i_bz\ l_{end}$ $code(body)$ $i_br\ l_{start}$ $l_{end}:$
Variable declaration statement: var $v = expr$	$code(expr)$
Integer constant expression: $value$	$i_push_int\ value$
Variable read expression: v	$i_peek\ stack_slot_v$
Variable write expression: $v = e$	$code(e)$ $i_poke\ stack_slot_v$
Dynamic terminal match grammar expression: $@expr$	$code(expr)$ $i_match_char_dyn$
Parameterized non-terminal grammar expression: $A(arg_1, arg_2, \dots, arg_n)$	$code(arg_1)$ $code(arg_2)$... $code(arg_n)$ $i_call\ n, ip_1$... $i_call\ n, ip_m$ $i_pop\ n$ $i_match_sym\ A \rightarrow l_{end}$ $l_{end}:$

language may be changed to more closely suit the environment in which EVM is being implemented.

An example EVM₃ grammar rule that imperatively matches n of 'a' characters and the compiled instruction sequence are shown in table VII. This compiled instruction sequence may be later invoked with instruction $i_call\ 1, 10$, because this grammar rule has 1 parameter and the instruction sequence implementing the grammar rule starts at offset 10.

F. Garbage collection of suspended fibers

The current version of EVM creates a state for every input terminal symbol. In case of a successful parse, every state needs to contain at least one fiber. If EVM is used in a scannerless setting, this means that the total amount of memory required for EVM will be significantly higher than that of the input string. As such, to support parsing longer strings, memory footprint of the EVM needs to be lowered.

There are several important observations to make:

- Most states and fibers after suspension will be never needed during parse again. As such, some states that are

TABLE VII
EVM₃ GRAMMAR COMPILATION EXAMPLE

Grammar rule	Instruction sequence
rule $field(n)$	10: $i_peek\ 0$
while $n > 0$	11: $i_push_int\ 0$
parse a	12: i_int_more
$n = n - 1$	13: $i_bz\ 20$
end	14: $i_match_char\ a \rightarrow 15$
end	15: $i_peek\ 0$
	16: $i_push_int\ 1$
	17: i_int_sub
	18: $i_poke\ 0$
	19: $i_br\ 10$
	20: $i_reduce\ "field"$
	21: i_stop

unnecessary, together with the fibers they contain, may be discarded before the parsing process completes.

- The only instructions that access variables from previous states are i_reduce and $i_lookahead_ret$.
- State index sid of a fiber is always equal or higher to the lowest value sid in the fiber queue. In other words, new fibers are always created with monotonically increasing state indices.

Based on these observations, the following optimizations can be made:

- Execution trace sets may be discarded from states with indices from interval $[1, sid_{min})$, where sid_{min} is the lowest state index in fiber queue Q . These sets are only needed in states, where new fibers may be created. Because new fibers are created with monotonically increasing state indices, the sets are no longer needed.
- *Unreachable* states with indices $[2, sid_{min})$ may be discarded completely.

A state with index sid is *reachable* if there exists a fiber (either running or suspended) with origin state index $origin$ equal to sid . As such, mark-and-sweep garbage collector may be employed to identify reachable and unreachable states.

The described garbage collector will discard all states with the fibers they contain that are not part of any parse rule/active reduction that can be traced back to the starting non-terminal symbol. As a result, it will have a significant impact on overall memory usage, especially when EVM is used without a dedicated scanner. To reduce the garbage collector's performance impact to the parsing process, the garbage collector may be run every n parsed terminal symbols.

III. CONSTRUCTING THE ABSTRACT SYNTAX TREE

So far, the last EVM version is only capable of *recognizing* the input. However recognizers have only limited practical applicability. As such, for EVM to truly be useful in practice, there needs to be a way to construct the abstract syntax tree from the terminal input symbol sequence.

Normally, extension of a recognizer into a parser is a fairly trivial task. However in case of EVM, constructing the AST is not as simple, because EVM supports parsing ambiguous inputs that may result in a parse forest, which represents

multiple valid parse trees for the same input. In case of highly ambiguous parses, naive approach of storing a complete copy of a parse tree for each valid parse path is not viable, as it may lead to an exponential parse forest growth. As a result, parsers that support parsing ambiguous inputs use special data structures to represent the parse trees, called *shared packed parse forests*, or SPPFs for short.

SPPFs look and behave similarly to regular parse trees. However, to represent ambiguity within a parse tree, special packed nodes are used. Children of a packed node represent different parses for the same input fragment. Furthermore, matching subtrees of packed nodes may be shared to reduce space requirements for storing the SPPF. In case of ϵ -grammars, SPPFs may contain cycles.

There are two methods available for constructing SPPFs within EVM₃: automatic and manual AST construction.

A. Manual AST construction

As the name implies, ASTs in EVM₃ may be constructed manually by relying on EVM₃'s imperative capabilities. The virtual machine may be extended with additional instructions that allow the creation and management of tagged AST nodes.

Specifically, the following new instructions are required:

- `i_new_node num, sym`. This instruction pops `num` AST node indices from the stack and creates an AST node with `num` popped children and label `sym`. Additionally, this instruction stores the interval `[origin, curr]` within the newly created node to represent the source interval of the node. Upon node creation, `i_new_node` pushes the reference of the resulting node to the stack. The newly created node is considered to be *detached*.
- `i_reduce_r sym`. Used to reduce a non-terminal symbol with a node-value. Works similarly to `i_reduce`, however upon execution it additionally pops a reference to an AST node from the stack and *attaches* it to the AST. The node attachment process works by assigning the node an unique index and storing that index and the node reference pair within the AST node list. Additionally, the node's unique index is stored within the reduction map of the origin state. If a reduction with the same non-terminal symbol and length already exists in the origin state, this indicates existence of ambiguity, as now there are two candidate non-terminal reductions within the same source interval. As such, a packed node is created in place of the previously created node. The old node gets assigned a new index and the two newly assigned indices are added as the children of the newly created packed node. If the previous node index already points to a packed node, then the node that is being attached is added to the packed node's children list.
- `i_match_sym_r sym1 → ip1, ..., symn → ipn`. Just like the original `i_match_sym`, this instruction is used to match a non-terminal symbol. Upon successful resumption of a fiber previously suspended by `i_match_sym_r`, the instruction also pushes an index

of AST node that represents the recently matched non-terminal symbol. The node referenced by the returned node index may be mutated later on by `i_reduce_r` instruction, as new ambiguous reductions are performed.

By applying this strategy of AST construction, AST nodes are created with the `i_new_node` instruction. Leaf nodes have no children and only contain the source range of input they represent. Nodes are then "returned" from grammar rules by using `i_reduce_r` instruction, which associates every AST node with a tuple $\langle sym, start, end \rangle$, where `start` and `end` represent the source range of the node. In the event that more than one reduction is being associated with the same tuple, then a packed node is created with children that represent the alternative parses of non-terminal symbol `sym` in the source range `[start, end]`. The returned node index is retrieved with the help of `i_match_sym_r` instruction, which pushes the index of the node into the stack of the callee. This enables the grammar program to use that index in future `i_new_node` instruction calls to construct non-leaf AST nodes. The node index that is returned from the start rule represents the root node.

It is important to note the importance of using node indices to represent the nodes rather than node references (or pointers): at the time of any given reduction (the call to `i_match_sym_r`) there is no way to know if there will be a matching future reduction that will cause the original reduction to become ambiguous. As such, node indices exist as a form of indirection, which allows swapping of a regular non-ambiguous node into an ambiguous packed node, when it is determined that there is more than one way to parse a given source range with the same non-terminal symbol.

To make use of these instructions, additional changes are required to the grammar language:

- A new AST node construction expression is needed to allow construction of AST nodes.
- A new **return** statement is needed that allows returning a previously constructed AST node.
- A new assignment grammar expression is needed to allow assignment of non-terminal return values (which store AST node indices) to previously declared variables.

An example grammar rule that manually constructs AST and the corresponding instruction sequence are provided in table VIII.

Even though manual AST construction requires new instructions and additional changes to the way grammars are specified and compiled, it also enables us to fine-tune and precisely control the AST construction process. Nodes, which are not meant to be included in the final AST may not be included as arguments for the `i_new_node` instruction, which effectively excludes such nodes from the final AST. Some grammar rules, such as the ones for parsing whitespace may not include any calls to `i_new_node` at all and may use the old `i_reduce` instruction, which should decrease the total amount of unnecessary AST nodes constructed and increase the overall performance of EVM₃ parser.

TABLE VIII
MANUAL AST CONSTRUCTION EXAMPLE

Grammar rule	Instruction sequence
rule E	...
parse l:E + r:I	30: i_call_dyn E
return node("plus",	31: i_match_sym_r E -> 32
l, r)	32: i_match_char + -> 33
end	33: i_call_dyn I
	34: i_match_sym_r I -> 35
	35: i_peek 0
	36: i_peek 1
	37: i_new_node 2, "plus"
	38: i_reduce_r e
	39: i_stop
	...

B. Automatic AST construction

In some cases it may be desirable to be able to construct ASTs without any additional changes to the parser grammars. In such event, automatic EVM AST construction may be used instead.

To support automatic AST node construction, the following changes to EVM₃ are required:

- Every fiber has to be extended to contain a list (or a stack) of child node indices that will be used during reduction to construct the AST node.
- `i_reduce sym` instruction has to be extended: 1) it has to construct an AST node with label `sym` and children from the child stack of the current fiber; 2) it has to include the node packing logic of the `i_reduce_r` instruction.
- `i_match_sym` instruction has to be extended: upon resuming a previously suspended fiber by `i_match_sym`, the instruction has to push the corresponding node index of the matched non-terminal symbol to the children stack.

Essentially, automatic AST construction works by merging instructions `i_reduce_r` and `i_new_node` into `i_reduce`; `i_match_sym_r` into `i_match_sym` and using a separate array/stack in each fiber instead of the general purpose stack to store the children node indices for future reductions.

While this approach is easier to implement, it is also requires more memory during parsing, as there is no way to exclude unnecessary nodes from the final abstract syntax tree. Both of these AST construction approaches create SPPFs.

IV. RELATED WORKS

The original Earley parser was first described in [1]. Originally it was created for parsing natural languages and saw limited use for parsing computer languages. Back then, the Earley parser was too inefficient to parse computer languages, as the computer resources were limited and computer languages used in practise were designed to be parsable by simpler and more efficient parsing algorithms, such as LR.

In recent years, as the computer performance rose, newer variations of the Earley parser appeared specifically designed for parsing non-natural languages. An Efficient Earley Parser

[2] modifies the original Earley parser by moving away from raw productions for internal grammar representation. Instead, it uses Earley graphs to represent the grammars internally. The move also enables the use of regular operators within the grammars. By applying a variation subset construction to the generated Earley graphs, the authors of [2] achieve additional performance gains. Finally, a further variation of Efficient Earley parser called Yakker [3] enables the use of data-dependant constraints within the grammars.

An Earley parser variation described in [4] moves in a different direction: the authors of [4] propose a version of Earley parser that is suitable for parsing reflective (more often referred as adaptable) grammars, that can be modified during parsing to augment the input language. A separate paper [5], describes how to translate Earley grammars into C programming language, thus eliminating many of dynamic elements of the original parser and improving the overall parsing performance.

It is also important to note that the original Earley parser is not a *true* parser, as it does not provide the means to construct the ASTs for the parse input. As such, paper by E. Scott proposes a method that enables construction of SPPFs during parsing in [6]. Because the way the garbage collection works in EVM, this method is not directly applicable to EVM.

Earley parser is not the only parser suitable for analysing context-free languages. The primary contender for that purpose is the GLR family of parsers. One of the more modern GLR variations is RNLGR parser [7]. However, much like like the original GLR, RNLGR is a table based parser and as such, while being very efficient, it is also fairly rigid and difficult to extend. Despite that, there have been attempts to augment various existing parsing methods, including Earley and GLR, to enable parsing of context-dependant constraints in [8]. A variation of RNLGR parser suitable for scannerless parsing is described in [9].

A recent alternative to context-free grammars for specifying languages is Parsing Expression Grammars, or PEGs for short [10] [11]. PEGs are often implemented by a Packrat parser, which is a memoizing recursive descent parser and as a result, the overall structure of the parser ends up being very simple. PEGs, just like EVM, support regular operators in rule definitions. Unfortunately, PEGs also inherit all the restrictions of recursive decent parsers, such as no support for parsing ambiguous grammars and no left-recursion. A virtual-machine based implementation for PEGs exists [12].

The use of virtual machines for parsing is not a new concept. One of the first descriptions of such parser was described by Donald E. Knuth in [13]. However, the inspiration for EVM came from [14], where a virtual machine for parsing regular expressions is proposed.

V. FUTURE WORK

There are several potential future research directions for EVM:

- Better parser error handling. To ease the use of EVM, there needs to be a way to automatically generate descriptive error messages in the event of a parse error.
- The performance of EVM needs to be evaluated. EVM is currently implemented as a prototype in Ruby programming language, which is used to parse a Ruby-like language, whose grammar consists of 500 lines of code. However, the Ruby implementation makes any performance comparisons to real-world parsers and parser generators, such as bison, invalid. Comparing performance to traditional parsers is further hampered by the fact that significant performance gains may be achieved by the use of regular operators to specify the repeated patterns in the parse input. As such, separate test grammars are required for EVM to maximize its performance.
- Translation of EVM grammar programs to LLVM IR would enable compiling EVM grammars into native machine code, which should increase the overall performance of the parser even further.
- Additional optimizations may be applied to compiled grammars programs to further increase the parser performance. Specifically, a variation of subset construction may be applied to reduce non-determinism and the number of fibers required for parsing.
- Negative lookahead, boolean operators and rule precedence specifiers would additionally simplify the development of new grammars.

VI. CONCLUSION

We have presented a new, virtual machine based approach to parsing context-free grammars, which was heavily inspired by the classic Earley parser. We have shown several versions of the Earley Virtual Machine with increasing complexity and expanding feature sets. The final version of EVM, the EVM3 is capable of recognizing context-free grammars with data-dependant constraints. Furthermore, EVM3 grammars support regular expression operators and regular lookahead in right hand sides of production rules,

which should simplify development of new grammars. Finally, we have shown two modifications of EVM3, which enable construction of shared packed parse forests during parsing.

ACKNOWLEDGMENT

Thanks to Institute of Mathematics and Informatics for financing this research.

REFERENCES

- [1] J. Earley, "An efficient context-free parsing algorithm," *Commun. ACM*, vol. 13, no. 2, pp. 94–102, 1970.
- [2] T. Jim and Y. Mandelbaum, "Efficient earley parsing with regular right-hand sides," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, pp. 135 – 148, 2010.
- [3] T. Jim, Y. Mandelbaum, and D. Walker, "Semantics and algorithms for data-dependent grammars," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, United States: ACM, 2010, pp. 417–430.
- [4] P. Stansifer and M. Wand, "Parsing reflective grammars," in *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, ser. LDIA '11. New York, United States: ACM, 2011, pp. 10:1–10:7.
- [5] J. Aycok and N. Horspool, *Directly-Executable Earley Parsing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 229–243.
- [6] E. Scott, "Sppf-style parsing from earley recognisers," *Electron. Notes Theor. Comput. Sci.*, vol. 203, no. 2, pp. 53–67, Apr. 2008.
- [7] E. Scott and A. Johnstone, "Right nulled glr parsers," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 4, pp. 577–618, Jul. 2006.
- [8] T. Jim and Y. Mandelbaum, "A new method for dependent parsing," in *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ser. ESOP'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 378–397.
- [9] G. Economopoulos, P. Klint, and J. Vinju, "Faster scannerless glr parsing," in *In Proceedings of the 18th International Conference on Compiler Construction (CC)*. Springer-Verlag, 2009.
- [10] B. Ford, "Packrat parsing: a practical linear-time algorithm with backtracking," *Master's thesis*, Massachusetts Institute of Technology, 2002.
- [11] —, "Parsing expression grammars: A recognition-based syntactic foundation," *SIGPLAN Not.*, vol. 39, no. 1, pp. 111–122, 2004.
- [12] S. Medeiros and R. Ierusalimschy, "A parsing machine for pegs," in *Proceedings of the 2008 Symposium on Dynamic Languages*, ser. DLS'08. New York, NY, USA: ACM, 2008, pp. 2:1–2:12.
- [13] D. E. Knuth, "Top-down syntax analysis," *Acta Inf.*, vol. 1, no. 2, pp. 79–110, Jun. 1971.
- [14] R. Cox. (2009) Regular expression matching: the virtual machine approach. [Online]. Available: <https://swtch.com/~rsc/regexp/regexp2.html>