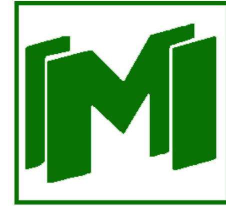
Vilnius University
Institute of Mathematics and Informatics
L I T H U A N I A



INFORMATICS ENGINEERING (07 T)

TOOLS AND TECHNIQUES FOR SYNTATIC AND SEMANTIC EXTENSION

Audrius Šaikūnas

October 2017

Technical Report MII-DS-07T-17-16

VU Institute of Mathematics and Informatics, Akademijos str. 4, Vilnius LT-08663, Lithuania
www.mii.lt

Abstract

Extensible programming languages are languages whose features (syntax or semantics) can be added by users without modifying the compiler of the language. Reflectively extensible programming (REP) languages are a subset of such languages, where extensions for the language are defined using the same language and which can be mixed with regular code. This report serves as the second part of the final dissertation: the report contains the theoretical details of a novel parsing method called Earley Virtual Machine. This parsing method provides sufficient features to parse a REP language.

Keywords: Extensible languages, parsing methods, adaptable parsing

Contents

1 Extensible parsing with Earley Virtual Machines 4
1.1 Earley Virtual Machines 4
1.1.1 Introduction to Earley Virtual Machines 4
1.1.2 EVM grammars 5
1.1.3 EVM states 5
1.1.4 EVM fibers 6
1.1.5 EVM interpreter 6
1.1.5.1 Parsing terminal symbols 6
1.1.5.2 Parsing non-terminal symbols 7
1.1.5.3 Resuming suspended fibers 8
1.1.6 Compiling basic EVM grammars 9
1.2 General purpose computation in EVM 9
1.3 Improving source grammar flexibility 10
1.3.1 Regular right hand sides in production rules 10
1.3.2 Rule and operator precedence 11
1.3.3 Specifying operator associativity 12
1.4 Parsing with regular lookahead 13
1.4.1 Fixed length lookahead 13
1.4.2 Variable length lookahead 14
1.5 Parsing with data dependant constraints 15
1.5.1 EVM grammar language 15
1.5.2 Matching input against dynamic content 16
1.6 Abstract syntax tree construction 18
1.6.1 Automatic AST construction 18
1.6.2 Manual AST construction 20
1.6.3 Delayed semantic actions 22
1.6.3.1 The arguments for delayed semantic actions 22
1.6.3.2 Constructing execution history labels 23
1.6.3.3 Compilation of grammars that use delayed semantic actions 24
1.6.3.4 Replaying labels 24
1.7 Parsing reflective grammars 27
1.7.1 Dynamic grammar composition 27
1.7.2 Extensions to EVM grammar language 28
1.7.3 Compiling EVM grammars with domains 29
1.7.4 Loading multiple grammar modules in EVM 29
1.7.5 Parsing reflective grammars in EVM 30
1.8 EVM performance improvements 31
1.8.1 Garbage collection of suspended fibers 31
1.8.2 Eliminating dynamic non-terminal call indirection 32
1.8.3 On-demand instruction subset construction 32
1.8.3.1 Importance of subset construction 32
1.8.3.2 Instruction ε-closures 33
1.8.3.3 Merging instruction ε-closures 34
1.9 Conclusions 36
References 37

1 Extensible parsing with Earley Virtual Machines

1.1 Earley Virtual Machines

1.1.1 Introduction to Earley Virtual Machines

Earley Virtual Machines (or EVM for short) is a new approach to parsing that is based on virtual machines and is heavily inspired by Earley parser.

The core idea behind EVM is to separate the two grammar representations used by the parser: the user writers *source grammars* in a plain-text format which are then parsed an compiled into *compiled grammars* that are then executed by the parser.

EVM consists of the following elements, each of which will be described in more detail in future chapters:

- **Source grammars** are parser grammars in plain text format. These grammars are written by the user of the parser and describe the parsed language in terms of grammar rules. Additionally, source grammars may contain the abstract syntax tree construction instructions, which allow to control the process of AST construction in fine detail.
- **Compiled grammars** or **grammar modules** are internal representation of source grammars. As the name implies, compiled grammars are compiled from source grammars. Compiled grammars contain sequence of low-level instructions that drive the parsing process.
- The **interpreter** is one of the primary elements of EVM. It *interprets* or executes the instructions contained in one or more grammar modules. As a result, an abstract syntax tree is constructed based on the parse input. The process of interpreting compiled grammars is synonymous to parsing the input data in the context of EVM.
- **States**. EVM state is an internal structure utilized by the interpreter that tracks execution of the interpreter. These EVM states have a close resemblance to the Earley parser states. There may exist one EVM state per each terminal symbol.
- **Fibers**. EVM fibers have a close relationship to Earley parser items. Each fiber represents a task of grammar rule execution. A fiber may be thought of as a thread of a general purpose programming language in which one grammar rule is executed.
- The **fiber queue** is a queue of fibers that are ready for execution. The interpreter works by removing the first fiber from the queue and keeps executing it until it yields. At which point the next fiber is removed from the queue and execution of it commences. Empty fiber queue indicates a parse error.

1.1.2 EVM grammars

Much like formal grammars, *basic EVM grammars* consist of production rules, where each production rule defines how to parse a single non-terminal symbol.

More formally, a *basic EVM grammar* is a set of productions in form $sym \to body$, where sym is a non-terminal symbol and $body$ is a *grammar expression*.

A *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 (grouping) grammar expression, where e is a grammar expression.
- $e_1 e_2$ is a sequence grammar expression, where e_1 and e_2 are grammar expressions.

EVM *compiled grammar* is a tuple $\langle instrs, rule_map \rangle$. $instrs$ is the sequence of instructions that represents the source 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.

1.1.3 EVM states

An EVM state is a structure that tracks the progress of interpreter at a particular point in the terminal symbol input sequence. Each EVM state has an index that corresponds to appropriate position of the input sequence.

Each EVM state S_i is a tuple $\langle susp, trace, reductions \rangle$, where:

- i is the position of the input sequence.
- $susp$ is a list of suspended tasks at position i. When one rule calls another, the caller is suspended until one or more of the callees complete. Each entry of the suspended task list is a pair $\langle fiber, symbol_map \rangle$, where $fiber$ is the suspended fiber. $symbol_map$ represents the reason of the suspension: it contains the set of non-terminal symbols that the callee expects to parse. Upon parsing any of these symbols, the caller fiber is resumed by adding it's copy to the fiber queue (thus signalling that the target non-terminal symbol has been parsed successfully and the parsing of caller rule may resume).
- $trace$ is the *execution trace set* (or ETS for short). It is a set of pairs $\langle ip, stack \rangle$. Whenever a new fiber is created (either by calling a new non-terminal symbol or by resuming a suspended fiber), the instruction pointer ip and the $stack$ of the *candidate* fiber is checked against the ETS. If the pair is not present in the ETS, then the

creation of the fiber commences and this pair is added to the ETS. Otherwise, the creation of the fiber is aborted. This mechanism ensures that the input position is parsed with the same grammar rule and the same context only once, thus avoiding exponential parsing complexity found in certain variations of recursive descent parser. The ETS also blocks infinite left recursion.

- *reductions* is a multimap that stores successful reductions that originate from state/offset i. The key of the multimap is a non-terminal symbol that indicates the target symbol, where the value of the map is a tuple $\langle offset_1, priority, value\rangle$. $offset_1$ indicates the end position of the reduction. *priority* indicates the priority of the reduction. This value is used in conjunction with negative reductions and is described in more detail later. *value* is the user-specified value of the reduction. It usually contains the AST node of the reduction, or when delayed semantic actions are used, the label of the reduction. The primary purpose of the *reductions* is to store the intermediate parsing results. Additionally it is used to merge reductions whose starting positions, ending positions and non-terminal symbols match. This avoids the exponential complexity explosion in case of ambiguous grammars/inputs.

1.1.4 EVM fibers

A fiber represents the task of parsing a single non-terminal symbol. Whenever a non-terminal symbol needs to be parsed, one or more fibers are created to parse the symbol. More specifically, a fiber is a tuple $\langle origin, offset, ip\rangle$:

- *origin* is the origin input position of the fiber. It indicates the starting position of the target non-terminal symbol in the terminal symbol input sequence. This value is used when completing reductions: a successful non-terminal symbol reduction is recorded in *reductions* variable of state S_{origin}. Additionally, appropriate suspended threads of state S_{origin} are resumed in state S_{offset}.

- *offset* indicates the input position of current fiber. When a single terminal symbol is parsed successfully, the current fiber is *advanced* by increasing this offset by 1.

- *ip* indicates instruction pointer of the current fiber.

1.1.5 EVM interpreter

1.1.5.1 Parsing terminal symbols

Terminal symbols in EVM are parsed with instruction `i_match_char`. This instruction has a single operand that contains a jumptable. This jumptable consists of pairs $\langle symbol, target_ip\rangle$, where *symbol* is a terminal symbol to be matched. *target_ip* is the target instruction pointer to jump to if the *symbol* is matched successfully.

Table 1: Terminal symbol sequence parsing example

Grammar rule	Instruction sequence

`S -> a b c`	`...`
	`20: i_match_char a -> 21`
	`21: i_match_char b -> 22`
	`22: i_match_char c -> 23`
	`23: i_reduce S, 0`
	`24: i_stop`
	`...`

In most basic cases, this instruction can be used only with a single entry in it's jumptable. However, when using subset construction optimization, multiple `i_match_char` instructions can be merged into one by combining their jumptables.

In case of a successful terminal symbol match, the ip of current fiber is set to the appropriate instruction pointer provided in the jumptable. Additionally, the current fiber is advanced by increasing it's $offset$ by 1.

In case of matching failure (when no terminal symbol in the jumptable matches the one in the $offset$ position of the input), the current fiber is immediately discarded: the execution of the fiber is halted and the fiber yields.

An example of a simple source grammar and it's instruction sequence is provided in table 1.

1.1.5.2 Parsing non-terminal symbols

Parsing of non-terminal symbols in EVM is significantly more involved. Multiple instructions are used to facilitate matching of non-terminal symbols:

- `i_call_dyn` S is used to *initiate* parsing of non-terminal symbol S. This instruction creates one or more fibers. The instruction pointers of newly created fibers are set to entry points of the compiled rules that define non-terminal symbol S. *origin* of the new fibers is set to $offset$ of the caller. Finally, newly created fibers are added to the fiber queue. It is important to note, that fiber creation process is still subject to the ETS rules: multiple `i_call_dyn` invocations to the same non-terminal symbol S will not result in additional fiber creation. After executing `i_call_dyn` instruction, the caller fiber continues it's execution normally.

- `i_match_sym` $S_1 \rightarrow ip_1, ..., S_n \rightarrow ip_n$ is used to match successful non-terminal symbol parses, that have been previously initiated by `i_call` family of instructions. Whenever a `i_match_sym` is executed, the current fiber is suspended by adding it to the list of suspended fibers *susp* in state S_{offset}. Additionally, the interpreter attempts to pre-emptively resume the suspended fiber in case any of the target non-terminal symbols have been successfully parsed prior to executing the current `i_match_sym` instruction.

- `i_reduce` $S, prio$ is used to perform reduction of the non-terminal symbol S. Firstly,

Table 2: Non-terminal symbol sequence parsing example

Grammar rule	Instruction sequence

`S -> A B C`	`...`
	`30: i_call_dyn A`
	`31: i_match_sym A -> 32`
	`32: i_call_dyn B`
	`33: i_match_sym B -> 34`
	`34: i_call_dyn C`
	`35: i_match_sym C -> 36`
	`36: i_reduce S, 0`
	`37: i_stop`
	`...`

this instruction records the presence of new reduction with priority *prio* in state S_{origin}. In case that there have been other reductions with same length and non-terminal symbol in state S_{origin}, but with greater priority, the current reduction is abandoned. This mechanism is used to implement negative reductions that can be used to exclude certain undesirable parses (for example, certain keywords can be excluded from identifiers). If the reduction is not abandoned, then this instructions finds all the suspended fibers in state S_{origin} that have been waiting for S and attempts to resume them. After completing `i_reduce` instruction, the current fiber continues executing normally.

- `i_stop` instruction discards the current fiber.

A simple example of matching several non-terminal symbols is provided in table 2.

1.1.5.3 Resuming suspended fibers

EVM fibers can be resumed in two circumstances: during `i_match_sym` or `i_reduce` instruction execution. In both cases, the suspended fibers can be resumed with the following steps:

1. The suspended thread is duplicated.
2. ip of the copy is set to target instruction pointer, which is retrieved from $symbol_map$.
3. $offset$ of the copy is set to $offset$ of the fiber that executes `i_reduce`. In case of pre-emptive resumption in `i_match_sym`, the new $offset$ value is retrieved from $reductions$ entry in state S_{offset}.
4. The new fiber is traced, by recording it's presence in state's S_{offset} execution trace set. If a matching entry already exists, the resumption of the fiber is aborted.
5. The new fiber is added to the fiber queue to be executed later by the interpreter.

Table 3: Basic source grammar compilation rules

Grammar element	**Instruction sequence**

Grammar:
G = $\{P_1, ..., P_n\}$
	`i_call_dyn` *main*
	`i_match_sym` *main* $\rightarrow l_{accept}$
	l_{accept} :
	`i_accept`
	`i_stop`
	$code(P_1)$
	...
	$code(P_n)$

Production rule: $P \rightarrow e$	$code(e)$ `i_reduce` $P, 0$ `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	$code(e_1)$ $code(e_2)$

1.1.6 Compiling basic EVM grammars

The rules for compiling basic source grammars to corresponding instruction sequences are proved in table 3. The notation $code(e)$ refers to corresponding sequence of instructions when compiling grammar element e. Instruction `i_accept` signals the interpreter that a matching input has been parsed. *main* is the name of starting non-terminal symbol of the grammar that is being compiled.

1.2 General purpose computation in EVM

The current model of EVM is quite flexible and can be extended to support general purpose computation during parsing. This general purpose computation may be used to imperatively control the parsing process and thus implement some of the required functionality to support data-dependant constraints.

EVM fibers already support stacks that can be used to store intermediate general purpose computation results. The following instructions are required, to enable general purpose execution during parsing:

- `i_br` *ip*. Unconditional branch to instruction pointer *ip*.
- `i_bz` *ip*. Conditional branch to instruction pointer *ip*. The branch condition value

is popped from the top of current fiber stack.

- `i_pop`. Remove and discard top stack element of the current fiber.
- `i_peek` *n*. Duplicate stack element *n* and push it to the top of the stack.
- `i_int_add`. Integer addition. Pop two values from top of the stack, add them as integers and push the result to top of the stack.
- `i_int_sub`. Integer subtraction.
- `i_int_neg`. Integer negation.
- `i_int_push`. Push immediate integer constant to top of the stack.
- `i_int_more`. Integer comparison.
- `i_str_push`. Push reference of string constant to the stack.
- `i_call_foreign` *id*, *n*. Call foreign method identified by index *id* with *n* arguments. Push the result of the call to the stack. Foreign methods are methods implemented in host environment of EVM and can be used to extend the functionality of EVM without having to directly modify the way EVM is implemented.
- ...

The list of instructions is non-exhaustive and additional instructions may be added based on requirements.

1.3 Improving source grammar flexibility

1.3.1 Regular right hand sides in production rules

Regular right hand sides is a feature commonly found in recursive descent and Packrat family of parsers [For02]. It allows the usage of regular operators in right hand sides of production rules. This simplifies the definition of new grammars, as repeated and optional grammar elements no longer need to be expressed solely via alternation and recursion.

To support such operators in EVM grammars, the definition of EVM grammar expression needs to be expanded. In addition to exiting grammar expressions, the following elements are too considered to be grammar expressions:

- $e?$ is optional grammar expression, where e is a grammar expression.
- $e+$ is one-or-more grammar expression, where e is a grammar expression.
- $e*$ is zero-or-more grammar expression, where e is a grammar expression.
- $e_1|e_2$ is alternative grammar expression, where e_1 and e_2 are grammar expressions.

Table 4: Regular operator compilation rules

Grammar element	**Instruction sequence**
Optional grammar expression: $e?$	`i_fork` l_{end} $code(e)$ l_{end}:
One-or-more grammar expression: $e+$	l_{start}: $code(e)$ `i_fork` l_{start}
Zero-or-more grammar expression: $e*$	l_{start}: `i_fork` l_{end} $code(e)$ `i_br` l_{start} 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}:

Figure 1: A grammar that defines simple expressions with binary operators

```
S -> E
E -> E "+" F | E "-" F | F
F -> F "*" T | F "/" T | T
T -> "0" | "1"
```

All of these new grammar elements can be implemented in EVM by adding one additional instruction:

- `i_fork` ip_{new} instruction clones (*forks*) the current fiber and sets the instruction pointer of the new fiber to ip_{new}. The newly created fiber is scheduled to be executed by adding it to the fiber queue, while the existing one continues executing normally.

The rules for compiling the new operators into instruction sequences are provided in the table 4.

1.3.2 Rule and operator precedence

Almost every existing programming language supports the notion of binary operators with differing precedences. In grammars such operators with different precedences are commonly implemented via operator expression hierarchies, as shown in fig. 1. Each different operator precedence level gets a separate non-terminal symbol, under which operators with that precedence level are defined. While such operator with precedence definition method is simple and easy to understand, it quickly becomes cumbersome when dealing with real-world programming languages, such as C++, Ruby and similar, which often have over 15 different levels of operator precedences.

Furthermore, extending such language grammars to include additional operators becomes difficult, especially when the new operator has a precedence level that is in be-

tween of two existing neighbour precedence levels. In that case, a new non-terminal symbol for the new operator precedence level has to be defined and the existing rule that defines lower precedence operators has to be updated to use the newly defined operator.

Because definition of operators (either unary, or binary) is such a fundamental task when defining new grammars for programming languages, newer parser generators and language translation frameworks often allow specifying the precedences of operators directly either by assigning each operator a numeric precedence value, or using operator definition order to infer the precedence of each operator [EKV09]. As such, it would be beneficial for EVM to support specification of operator precedence levels natively, especially because one of the goals of EVM is to support adaptable grammars that can be extended dynamically during runtime.

In EVM the term operator precedence is generalized to *rule precedence*, as any grammar rule can have an explicit precedence value. All rules that have no explicit precedence definition have default precedence value of 0.

When compiling source grammars, the precedences are stored in *rule_map* entry of the compiled grammar. As a result, *rule_map* contains a multimap from non-terminal symbols to rule instruction entry point and rule precedence pairs.

Furthermore, instruction for invoking non-terminal symbols `i_call_dyn` needs to be extended to include *minimum rule precedence* operand, which is then used to filter out rules with lower precedence than requested. Source grammar compiler can make use of this operand when detecting that a grammar rule is recursively invoking itself: in that case only rules with greater precedence in comparison to the precedence of current rule should be invoked. Such mechanism emulates the behaviour of operator hierarchy without having to explicitly define it.

Changing just `i_call_dyn` to support rule precedences is insufficient however, because `i_match_sym` instruction has no notion of rule precedence and as such will interpret any successful match of target non-terminal symbol as a valid one, even when the the callee expects only a non-terminal symbol with a specific minimum precedence. Therefore, a new instruction is needed to match non-terminal symbols with a specified precedence:

- `i_match_dyn` $S, prec_{min}$ instruction matches successful parses only of non-terminal symbol with minimum precedence $prec_{min}$. Just like the original `i_call_dyn`, it suspends the current fiber and attempts to pre-emptively resume it by checking the existing reductions in state S_{offset}. When resuming the fiber, it's instruction pointer is set to $ip+1$.

1.3.3 Specifying operator associativity

Operator associativity can be considered as a separate edge case of rule precedence. Left associative operator $E+E$ means that the left non-terminal E can be expanded recursively into itself, while the right E has to be expanded into expression only with higher prece-

Figure 2: Rewritten grammar that defines simple expressions with binary operators

```
S -> E
E[10] -> *E "+" E
E[10] -> *E "-" E
E[20] -> *E "*" E
E[20] -> *E "/" E
E[30] -> "0" | "1"
```

Table 5: Updated non-terminal symbol compilation rules

Grammar element	**Instruction sequence**
Non-terminal grammar expression (non-recursive): A	`i_call_dyn` $A, 0$ `i_match_sym` $A \rightarrow ip_{next}$
Non-terminal grammar expression (recursive): A	`i_call_dyn` $A, prec+1$ `i_match_dyn` $A, prec+1$
Non-terminal associative grammar expression (recursive): $*A$	`i_call_dyn` $A, prec$ `i_match_dyn` $A, prec$

dence. As such, operator associativity specification can be implemented using operator precedence mechanism.

A new grammar element needs to be added to grammar expression to indicate when a non-terminal symbol is allowed to recursively expand into itself:

- $*A$ is non-terminal associative grammar expression, where A is a non-terminal symbol. When used in a production rule whose head is A, this grammar expression indicates, that $*A$ can be expanded recursively with current production rule.

As indicated above, by default all recursive non-terminal invocations are non-associative. This is because if user were to forget to explicitly specify associativity of $E+E$ expression, it would become ambiguous, as it could be interpreted both as left and right associative at the same time.

The example grammar in fig. 1 can now be rewritten using explicit rule precedences and non-terminal associative symbols into the one shown in fig. 2. New operators can be added as needed by specifying new production rules with explicit precedences. When adding new operators, no existing rules need to be changed or altered in any way.

The updated rules for generating instruction sequences for non-terminal symbols are provided in table 5. *prec* value refers to the precedence of the current rule that is being compiled. By default this value is 0, if not specified explicitly with square bracket notation.

1.4 Parsing with regular lookahead

1.4.1 Fixed length lookahead

Parsing lookahead is a useful feature that can simplify specifying grammars. When using a parser in scannerless mode, lookahead becomes mandatory, as it is needed to

Table 6: Fixed length lookahead example

Grammar rule	Instruction sequence

`A -> a+ &b`	`40: i_match_char a -> 41`
	`41: i_fork 40`
	`42: i_match_char b -> 43`
	`43: i_advance -1`
	`44: i_reduce A`
	`45: i_stop`

Figure 3: Grammar rule that defines identifier using fixed length lookahead

```
id -> [a-zA-Z_] [a-zA-Z_0-9]* &[^a-zA-Z_0-9]
```

Table 7: Fixed length lookahead compilation rules

Grammar element	Instruction sequence
Fixed length lookahead: &*e*	$code(e)$ `i_advance` $-length(e)$

implement greedy-matching when defining language tokens. For example, identifier can be defined as a sequence of alphanumerical characters that terminates on first non-alphanumerical symbol. As such, in order to correctly specify the termination point of an identifier, single-character lookahead is required.

In EVM fixed length lookahead could be mostly implemented already using the existing `i_match_char` instruction that is used to match terminal symbols. All what is needed is to backtrack to correct correct input offset after performing lookahead. This could be implemented using a new instruction:

- `i_advance` n instruction advances current fiber by n symbols. This operand may be negative to perform fixed length backtracking.

To make use of this instruction, the definition of grammar expression needs to be extended to include:

- &*e* is positive lookahead grammar expression, where e is a grammar expression.

An example usage positive lookahead operator is provided in table 6. Figure 3 shows an example where positive lookahead can be used in a real-world scenario when defining identifiers.

The rule for compiling fixed-length lookahead grammar expressions is provided in table 7. $length(e)$ refers to the character (terminal symbol) length of grammar expression e.

1.4.2 Variable length lookahead

Variable length lookahead in EVM can be implemented with a similar fashion. However, the difficulty in this case is not knowing how many terminal symbols to backtrack after

Table 8: Variable length lookahead compilation rules

Grammar element	Instruction sequence

Variable length lookahead: &e	`i_push_offset`
	$code(e)$
	`i_pop_offset`

performing the lookahead operation. As such, this information can be recorded and used dynamically by leveraging general purpose computation capability of EVM.

To support variable length lookahead, two additional instructions are required:

- `i_push_offset` pushes the $offset$ value of the current fiber to it's stack.
- `i_pop_offset` pops the $offset$ value of the current fiber from it's stack.

The rules for compiling variable length lookahead grammar expressions is provided in table 8. It is important to note that both fixed and variable length lookahead share the same notation. As such, it is up to source grammar compiler to determine when the lookahead operation is fixed length and to use the appropriate compilation rule. It is also possible to use variable length lookahead even in situations where fixed length lookahead would be more suitable, but with additional performance cost, as variable length lookahead makes use of fiber's stack.

1.5 Parsing with data dependant constraints

1.5.1 EVM grammar language

We have already shown that EVM is capable of performing general purpose computation and hinted that conditional control transfer can be used to drive the parsing process. However, the current grammar language that is only capable of specifying simple production rules that are composed from grammar expressions. Therefore, in order to be able to make use of conditional control transfer, the source grammar language needs to be extended to include control flow statements.

Table 9 presents the updated grammar elements and their instruction sequence compilation rules. The list of new grammar elements is non-exhaustive and doesn't include additional variations of existing elements (for example, various integer operations can be implemented in similar fashion to integer addition just by changing the final instruction).

Every variable defined within rule body is assigned a *stack slot*. A stack slot is a position in fiber's stack where the value for the variable is stored. $stack_slot_v$ refers to the stack slot number for variable v.

In the new grammar language, all grammar elements are divided into several categories:

- **Top level declarations** are used to define new grammar rules.

- **Statements** are used to control execution flow. In the extended grammar language, the bodies of rules are composed of statements.
- **Expressions** are used to perform general purpose computations, much like in traditional programming languages.
- **Grammar expressions** are used to perform parsing. Grammar expressions can be executed by using **parse** statement.

Grammar rule definitions are now extended to support parameters that can be used to control execution flow. To implement this, additional instruction changes are required:

- `i_call_dyn` instruction needs to be extended to include the argument number to copy to the callee. The copied arguments are discarded from the caller's stack frame after the call is complete.
- `i_reduce_r` (*reduce and return*) instruction needs to be created to allow returning values from the callee. It behaves exactly the same as `i_reduce`, but also pops a value from the current fiber's frame and stores it in *reductions* entry of state S_{origin}. This value can be accessed later by `i_match_dyn_r` instruction.
- `i_match_dyn_r` instruction behaves exactly the same as `i_match_dyn`, but also pushes the return value of the callee to the current fiber's stack.

parse and other control statements can be mixed and matched to parse complex data dependant grammars that cannot be parsed with traditional context-free parsers. For example, table 10 show how to parse fixed length fields, commonly found in binary formats.

1.5.2 Matching input against dynamic content

While the mechanism for dependant parsing described in previous chapter is powerful, but it is not sufficient to parse languages like XML: in order to be able to parse XML it is necessaries to be able to extract a fragment of parsed input and then use that extracted fragment for further matching.

As a result, two additional additions to grammar expression are required:

- $v@e$ is a *capturing grammar expression*, where e is a grammar expression and v is a name (identifier) for a new variable. After successfully matching e, this operator will store the range (the start end end offsets) of the matched input.
- $= v$ is a *dynamic match grammar expression*, where v is a variable that stores input range. This operator is used to match input against the one that is referenced by the range.

To implement these new constructs, only one new instruction is needed:

Table 9: Extended grammar language elements and their compilation rules

Element name	Syntax	Instruction sequence
Grammar rule	**rule** $sym(arg_1, ..., arg_n)$ $stmt_1$ ... $stmt_n$ **end**	$code(stmt_1)$ ... $code(stmt_n)$ `i_reduce` sym, 0 `i_stop`
Block statement	$stmt_1$ ... $stmt_n$	$code(stmt_1)$ ... $code(stmt_n)$
If statement	**if** *cond* *body* **end**	$code(cond)$ `i_bz` l_{end} $code(body)$ l_{end}:
Parse statement	**parse** *grammar_expr*	$code(grammar_expr)$
Return statement	**return** *expr*	$code(expr)$ `i_reduce_r` sym, 0 `i_stop`
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 addition expression	$e_1 + e_2$	$code(e_1)$ $code(e_2)$ `i_int_add`
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` $stack_slot_v$
Parameterized non-terminal grammar expression	$A(arg_1, arg_2, ..., arg_n)$	$code(arg_1)$ $code(arg_2)$ ... $code(arg_n)$ `i_call_dyn` A, $prec_{min}$, n `i_match_dyn` A, $prec_{min}$

- `i_match_range` pops two integer values from the fiber's stack that represents input range and attempts to match the input at current position against the characters referenced by the range. In case of a successful match, the current fiber is advanced by the length of the range. In case of a failure, the current fiber is discarded. This instruction is fairly unique in EVM, as it is the only one that can match more that one terminal symbol at the same time.

A grammar rule example that can match simplified XML tags is provided in fig. 4. This rule combines multiple key elements of EVM to successfully parse XML tags: fixed

Table 10: Parsing fixed length fields

Grammar rule	Instruction sequence

`rule field(n)`
`  while n > 0`
`    parse "a"`
`    n = n - 1`
`  end`
`end`
	`10: i_peek 0`
	`11: i_push_int 0`
	`12: i_int_more`
	`13: i_bz 20`
	`14: i_match_char a -> 15`
	`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", 0`
	`21: i_stop`

Figure 4: Simplified XML tag grammar rule

```
rule xml_element()
  parse "<" start@([a-zA-Z_] [a-zA-Z_0-9]* &[^a-zA-Z_0-9]) xml_attrs ">"
  parse (*xml_element)*
  parse "</" =start ">"
end
```

Table 11: Rules for compiling capturing and dynamic match grammar expressions

Grammar element	Instruction sequence
Capturing grammar expression: $v@e$	`i_push_offset` $code(e)$ `i_push_offset`
Dynamic match grammar expression: $=v$	`i_peek` $stack_slot_{v0}$ `i_peek` $stack_slot_{v1}$ `i_match_range`

length lookahead, associative non-terminals, dynamic matching.

Rules for compiling newly added grammar expressions into instruction sequences are provided in table 11. $stack_slot_{v0}$ and $stack_slot_{v1}$ refer to the stack slot indices of values produced by `i_push_offset` instructions.

1.6 Abstract syntax tree construction

1.6.1 Automatic AST construction

EVM in it's current iteration cannot be called a parser, as it only currently performs input recognition. As such, for EVM to be truly useful and applicable, there needs to be a way to construct the abstract syntax tree of the matched input.

There are multiple ways of how AST can be constructed in EVM, and in this section we describe *automatic* AST construction that requires no grammar modifications or any additional input from the user to be able to construct the AST.

Such method of AST construction can be implemented by augmenting the definition of EVM fiber: an additional stack can be added to each fiber that can store children nodes

of the current non-terminal symbol that is being parsed. To make use of such stack, the following instructions would need to be updated:

- `i_reduce` $A, prio$ in addition to performing reduction, additionally constructs the AST node for the non-terminal symbol that is being reduced. The newly constructed node is composed from nodes found in children node stack. Additionally, the node is tagged with non-terminal symbol A. Furthermore, the source range for the non-terminal can be added by including a copy of the pair $\langle origin, offset \rangle$, as $origin$ refers to the starting position and $offset$ refers to the current (and thus ending) position of the non-terminal. Finally, `i_reduce` *registers* the newly constructed node.

- `i_match_dyn` additionally adds the corresponding node index to the child AST stack, thus making these indices available during AST node construction in `i_reduce` instruction.

It is important to note, that EVM is capable of parsing ambiguous grammars, in which case the AST size may grow exponentially. To avoid this, shared packed parse forests (or SPPFs for short) can be used [Sco08]. In SPPFs subtrees that refer to alternative parse paths are packed into a single ambiguous node.

The key difficulty in constructing such SPPFs within EVM is that corresponding reductions may not happen sequentially: it is entirely possible that two reductions that refer to alternative parses may be separated by several, completely unrelated reductions. As such, the SPPF cannot be constructed in a single pass, as any node that was previously constructed may become ambiguous as more reductions complete.

Therefore, a layer of indirection is necessary to ensure that nodes can be changed from non-ambiguous to ambiguous after they have been constructed. In EVM's case, each node is assigned a unique index. Nodes in EVM internally are referred by storing and passing these indices around: the child node stack of each fiber stores node indices and `i_reduce` instruction uses node indices to compose new nodes. The actual node data (such as child node vectors) are stored separately.

To allow the changing of node type, the node registration process within `i_reduce` is used:

- If a reduction is *unique* (i.e. there are no other reductions that share the same source interval and the same non-terminal symbol), then a normal child node is constructed. Then it is assigned a unique index and this index is stored within *reductions* entry in appropriate state.

- If a reduction is *non-unique* (or ambiguous), then a normal child node is created and it is assigned a unique index. However, this time the existing node is converted to ambiguous packed node, and the newly created node is added as it's child.

The conversion of non-ambiguous node to ambiguous node works by duplicating the target node, assigning it a new unique index and changing the target node's type to

ambiguous. The duplicate of the original is then added as the only child of the converted node.

These node registration and conversion processes ensure that the node references are not broken when node conversion occurs. This enables incremental construction of SPPFs when there is no prior knowledge of which nodes will become ambiguous.

While this approach of AST construction is simple, it has two primary flaws:

- Inclusion of undesirable AST child nodes. EVM is primarily a scannerless parser, and as such will be used to parse whitespace. It is not uncommon to define a non-terminal symbol for recognising whitespace and then using that within other grammar rules. As such, during automatic AST construction, nodes that represent whitespace will be added to the resulting AST, possibly unnecessarily increasing the overall size of AST and littering it with nodes that carry no semantic information.

- Rigid and inflexible AST node type. Every normal node of the AST currently shares the same type and thus the same structure. Such behaviour however may not be desirable, as different non-terminal symbols represent different language elements with unique behaviours. Furthermore, it is common to use the AST to store semantic information when performing semantic analysis of the AST during later stages of compilation. Current node model has no space reserved for such semantic information and changing the node type would require changing the internals of EVM itself. The most flexible way to use the parsed result would be to convert the EVM AST to possibly polymorphic user-defined AST type that includes all the necessary fields and behaviours to perform semantic analysis.

Because of the flaws of this AST construction method, other alternatives should be investigated.

1.6.2 Manual AST construction

Manual AST construction is the polar opposite of the automatic AST construction: instead of requiring the EVM to define and construct the AST automatically, the responsibility of the AST definition and construction is moved completely to the user.

As EVM supports general purpose computation, it would be logical to assume that this method could be extended to enable manual and imperative construction of the AST.

Firstly, the EVM grammar language needs to be extended with the following constructs:

- $v : E$ is a capturing non-terminal grammar expression, where v is the variable name for storing the captured result and E is one of available non-terminal grammar expressions (plain or associative).

Table 12: Grammar rule for parsing and AST node construction of binary addition

Grammar rule	Instruction sequence

rule expr[10]
  parse l:*expr "+" r:expr
  return <add l r>
end
	60: i_call_dyn "expr", 10
	61: i_match_dyn_r "expr", 10
	62: i_match_char '+' -> 63
	63: i_call_dyn "expr", 11
	64: i_match_dyn_r "expr", 11
	65: i_str_push "add"
	66: i_peek 0
	67: i_peek 1
	68: i_new_node 2
	69: i_reduce_r "expr", 0
	70: i_stop

- <*name* arg_1 ... arg_n> is a node construction expression. Node is constructed with head *name* and arguments arg_1 ... arg_n. Arguments can be other nodes, integer values or string values.

Additional instruction *i_new_node n* is needed that constructs new AST node with n arguments/children. The head (type) of the node must be provided in the stack before pushing arguments. As a result, *i_new_node* will always pop $n+1$ elements from the stack. This instruction is needed to implement node construction expression. However it can be implemented as a foreign call as well.

Example usage of manual AST construction is provided in table 12.

To avoid exponential AST growth in ambiguous cases, similar mechanism for constructing SPPFs as described in previous section should be used. `i_new_node` should return a node index and `i_reduce_r` should include the node registration logic that would enable the merger of ambiguous subtrees into packed nodes.

However, it is known that the grammar is unambiguous or that the ambiguity would be minimal, then direct node references could be used and `i_reduce_r` would no longer need to include the node registration logic. Furthermore, nodes could be constructed in the host environment via foreign calls, thus allowing user to manually define and use different node types where desirable. That way, both weaknesses of automatic AST node construction could be avoided at a cost of having to manually specify (both within the grammars and possibly within the host environment) of how to construct the AST.

Even though this approach has numerous advantages of the automatic AST construction, one key flaw still persists:

- Wasted resources during speculative parsing. As EVM performs parsing breadth first, quite a few parse paths get discarded. Consider parsing expression $2+3*4$. Upon parsing the $2+3$ portion of the input, a complete addition node would be constructed and stored within *reductions* entry of S_1. However, this node would be never used, as eventually the remainder of input would be parsed and two additional nodes would be constructed (one for $3*4$ and one for the whole expression). The problem here is twofold: highly speculative nature of EVM and too eager construction of the resulting nodes. The problem becomes even more significant when

Figure 5: Grammar rule for parsing argument list separated by commas

```
rule arg_list
  parse (a0:arg ("," a1:arg)*)?
  return <arg_list a0 *a1>
end
```

using more "heavy" nodes that contain fields that are meant to be used during later stages of compilation, source ranges for error reporting and other information. In that case both the memory usage of unused nodes and the time it takes to construct them may become a significant performance drain of overall parsing process.

As such, it would be useful, if node construction could be delayed only until the parser is sure that the node won't be discarded.

1.6.3 Delayed semantic actions

1.6.3.1 The arguments for delayed semantic actions

Delayed semantic actions [JM11] is an attempt to avoid too eager computation within non-terminal rules that may not contribute to the parsing result in the Yakker parser [JMW10]. In this section we present an adaptation of delayed semantic actions for EVM.

The core idea behind delayed semantic actions is to separate parsing into two distinct phases:

- **Early** and non-deterministic phase, that performs parsing and constructs an execution history.
- **Late** and deterministic phase, that consumes the execution history and uses it to execute necessary semantic actions (possibly for AST construction).

Consider the example in table 12. It contains three semantic actions, whose execution can be delayed: the assignment of l variable, the assignment of r variable and finally the construction of the AST node. In case of EVM, delaying these 3 actions would mean that fiber's stack in many situations would become optional thus making fiber suspension process more efficient, as it's no longer necessary to both allocate and store the stacks of suspended fibers.

The advantage of delaying AST construction becomes even more apparent in the example provided in figure 5. Both operators in EVM that provide repetition ($+$ and $*$) are implemented in EVM by using `i_fork` instruction, which makes a copy of the current fiber with altered instruction pointer. In case that the actual argument list consists of n elements, EVM will perform n forks and reductions in *arg_list* rule alone. As a result, $n+1$ *arg_list* nodes will be constructed, out of which n will be never used again (assuming that the grammar is non-ambiguous). As such, delaying AST construction is of vital importance in EVM.

1.6.3.2 Constructing execution history labels

As mentioned previously, the core idea behind delayed semantic actions is to construct execution history composed of *labels* that somewhat mirrors the structure of AST, but with one key difference: whereas the AST nodes are heavyweight and contain significant amount of information, the individual labels are small and lightweight. Then these labels can be *replayed* (either in separate late phase, or in parallel during parsing), thus executing the semantic actions that have been previously delayed.

Several different label types are required:

- *Tag label* is a unary label. It stores a reference to the previous label and a general purpose numeric value. The semantic meaning of the numeric value depends on other nearby labels.
- *Call label* is a binary label that indicates a call branch. It stores a reference to the previous label and a reference to the reduction label of the callee.
- *Normal reduction label* is a unary label that indicates a successful non-ambiguous reduction. Stores a reference to the previous label and the reduction tag. Reduction tag is a value that uniquely identifies a reduction. Normal reduction label may be mutated to ambiguous reduction label.
- *Ambiguous reduction label* is a binary label that indicates an ambiguous reduction. Stores two references to reduction labels, which may too be ambiguous.
- *Resolved reduction label* is a 0-ary label that stores the result of the reduction, which is computed by executing corresponding delayed actions. Normal and ambiguous reduction labels can be mutated into resolved labels after they have been replayed. The use of resolved labels avoids replaying the same reduction labels several times.
- *Nil label* is a 0-ary label that terminates tag or call label chain.
- *Range label* is a unary label that that holds a source range. Used when parsing language tokens to hold starting and ending position of a token, thus avoiding the need for two separate tag labels.

To facilitate the construction of labels, a definition of a Fiber is extended to include a *current label label*. General purpose stack is not used for holding labels, as the fiber's stack is a variably sized structure, thus requiring separate allocation.

Furthermore, additional instructions and existing instruction changes are required:

- `i_trace` *tag* constructs a new tag label $\langle label, tag \rangle$ and sets the label of the current fiber to the newly constructed one. This instruction is used to delay execution of statements and expressions within rule definition.
- `i_trace_offset` sets *label* to $\langle label, offset \rangle$. It it used to capture the current parsing location so it may be used when replaying labels.

- `i_trace_range` sets *label* to $\langle label, origin, offset\rangle$. It it used to capture the input range of the current non-terminal so it may be used when replaying labels.
- `i_reduce` A and `i_reduce_r` A now construct a normal reduction label $l_1 = \langle label, A\rangle$. Then this label is registered by checking if the new reduction is ambiguous. In case that this is true, then existing reduction label l_0 is duplicated and a new ambiguous label $\langle l_0, l_1\rangle$ is constructed **in place** of the old one.
- `i_match_sym`, `i_match_dyn`, `i_reduce` and `i_reduce_r` now construct a call label when resuming suspended fibers.

All newly constructed fibers (usually with `i_call*` family of instructions) are initialized with nil label.

1.6.3.3 Compilation of grammars that use delayed semantic actions

The rules for compiling grammars with delayed semantic actions are provided in table 13.

A *fully capturing parse statement* is a parse statement that contains a single capturing grammar expression that captures the entire input of a non-terminal symbol. It is meant to be used in language token definitions. A fully capturing parse statement is an optimized variation of the original parse statement. If a rule contains a single parse statement and the grammar expression of that statement is a capturing one, then the original parse statement may be substituted with a fully capturing one. This is an important optimization for parsing tokens, as it avoids the need for processing. In other words, the `i_trace_range` instruction when compiling the statement is only added as a suffix. This becomes especially important when using `i_trace_range` in conjunction with subset construction optimization.

$label_{next}$ in table 13 refers to the next label index. Labels in capturing non-terminal grammar expressions are indexed from 100 to differentiate them from the ones generated with `i_trace_offset` instruction. These labels are referred to as *action labels* as they refer to a delayed action (in this case, assignment of a variable). Action labels are specifically defined to be locally, but not globally unique. That means that in every non-terminal rule action labels are numbered from 100. This further aids when performing instruction subset constructions, as `i_trace` instructions with the same tag may be merged together.

1.6.3.4 Replaying labels

Execution history labels are created within EVM, often by using specialized label creation instructions. However, they can be replayed outside of EVM, possibly in the host environment. This reduces AST construction difficulty, as native data structures and method/function calls may be used to construct the AST.

When the EVM completes parsing, the reduction label of starting symbol may be found in *reductions* entry of state S_1, whose length matches the total length of the input.

Table 13: Rules for compiling grammars with delayed semantic actions

Element name	Grammar element	Instruction sequence
Fully capturing parse statement	**parse** $r@grammar_expr$	$code(grammar_expr)$ `i_trace_range`
Delayed return statement	**return** *expr*	`i_reduce_r` *sym*, 0 `i_stop`
Capturing grammar expression	$v@e$	`i_trace_offset` $code(e)$ `i_trace_offset`
Capturing non-terminal grammar expression	$v : E$	$code(E)$ `i_trace` $label_{next}$

This label is the result of parsing and can be used independently of EVM to perform semantic action playback.

The label playback process consists of several steps:

1. **Collection**. During the collection step, labels for a single non-terminal symbol are collected into an array (essentially flattening a linked list of labels into array). The first label in the resulting array is always the normal (non-ambiguous) reduction label that contains the unique reduction tag. The rest of the labels are added to the array in the order they were constructed. Call labels are added to the resulting array without traversing the callee labels.

2. **Replay function selection**. Once the label sequence is collected, the replay function based on the non-terminal symbol tag is selected. Every non-terminal rule has a corresponding replay function that can be used to replay labels for that non-terminal rule.

3. **Execution**. The appropriate replay function is invoked. Within it's body, the necessary local variables are initialized and label array is iterated over and the corresponding semantic action for each label is executed. This step may invoke label playback recursively when resolving call labels.

4. **Disambiguation**. If the original reduction label was ambiguous, then disambiguation function is invoked, which has to produce a single value from all possible alternatives. When constructing SPPFs, the result of disambiguation step is a SPPF node that combines all possible alternatives.

5. **Resolution**. The original reduction label is replaced with a resolved label that stores the result of the playback.

Depending on the current label, a different action is performed during the resolution step:

- For call labels, the label playback process is invoked recursively. The resulting resolved label is recorded as the *previous label*.

Table 14: Grammar rule and the corresponding instruction sequence for binary addition when delayed semantic actions are used

Grammar rule	Instruction sequence

`rule expr[10]`
`  parse l:*expr "+" r:expr`
`  return <add l r>`
`end`
	`60: i_call_dyn "expr", 10`
	`61: i_match_dyn_r "expr", 10`
	`62: i_trace 100`
	`63: i_match_char '+' -> 64`
	`64: i_call_dyn "expr", 11`
	`65: i_match_dyn_r "expr", 11`
	`66: i_trace 101`
	`67: i_reduce_r "expr", 0`
	`68: i_stop`

Figure 6: The replay function for binary addition in Ruby programming language

```
def action_expr(replay)
  l = nil
  r = nil
  each_action(replay) do |action_id|
    case action_id
    when 100
      l = prev_result
    when 101
      r = prev_result
    end
  end
  return create_add_node(l, r)
end
```

- For range labels, the label is only recorded as the previous label.
- For tag labels, the appropriate semantic action is executed based on the numeric value of the tag.
- Other labels may not be encountered in a properly constructed execution history during the execution step.

The grammar rule example provided in table 12 can now be compiled into a different instruction sequence, shown in table 14, when delayed semantic actions are used.

The replay function for the rule, implemented in Ruby programming language is shown in fig. 6. The method `each_action` iterates over the collected labels (starting from the 2nd label). `prev_result` accesses the resolved value of the previous resolved label. `create_add_node` is a user defined method that constructs the binary addition AST node. It is important to note, that the replay function can be implemented in any language and it is not in any way bound just to Ruby programming language. For example, the same replay function can be implemented in C programming language, as shown in fig. 7.

Figure 7: The replay function for binary addition in C programming language

```
void action_expr(replay_t* replay) {
  node_t* l = NULL;
  node_t* r = NULL;
  REPLAY_ITERATE(label, replay) {
    switch (label_action_id(label)) {
      case 100:
        l = (node_t*) replay_prev_result(replay);
        break;
      case 101:
        r = (node_t*) replay_prev_result(replay);
        break;
    }
  }
  return create_add_node(l, r);
}
```

1.7 Parsing reflective grammars

One of the key reasons of choosing Earley parser as basis for constructing the parsing method for a REP language is it's flexibility and limited need for grammar preprocessing. In this chapter we describe how EVM can be extended to support adaptable grammars. The approach for implementing adaptable grammar support in EVM is inspired by **??**.

1.7.1 Dynamic grammar composition

Because EVM is primarily a scannerless parser, dynamic syntactic extension can be achieved by dynamically loading additional grammars during the parsing process. EVM grammars are composed out of grammar rules, so dynamic syntactic extension would consist of extending the active set of grammar rules.

The current version of EVM is fairly dynamic: non-terminal symbols are invoked via `i_call_dyn` instruction, which spawns possibly several fibers to parse the target non-terminal. The successful completion of a non-terminal is detected by a corresponding `i_match_dyn` instruction. There is no reason why the list of active grammar rules used by these instructions has to be static. By adding additional instructions that manipulate this list it would be possible to dynamically extend or constrain the active language that is begin parsed.

Unfortunately, a single global list of active grammar rules is insufficient to correctly parse any context-free grammar, as the statement for grammar rule activation may be ambiguous. Which means that in such situation a parser must be able to parse the same input with two separate sets of grammar rules: one in case the the recognised statement meant activation of new grammar rules and another if that was just an ordinary statement. Therefore, the active list of grammar rules has to be bound to a specific fiber.

To avoid having to make multiple copies of the active grammar rules, the target language can be divided into *domains*. A domain is a part of a grammar. Each grammar rule is assigned a set of domains. Each fiber has a set of active domains. If the set of rule

Table 15: Additional grammar language elements to support reflective grammars

Element name	**Element syntax**
Domain definition	`domain dom1`

Grammar rule with domain annotation	`@domains dom1 dom2 dom3`
	`rule name`
	`  stmt1`
	`  ...`
	`  stmtN`
	`end`

Domain activation statement	`with_domains dom1 dom2 dom3`
	`  stmt1`
	`  ...`
	`  stmtN`
	`end`

domains is a subset of fiber's active domains, then that grammar rule is considered to be active within the context of the domain. By manipulating the set of active domains it is possible to dynamically extend and constrain the current language.

Additionally, this method of grammar division and domain activation can be used to eliminate certain flaws present in traditional parsers: for example, there is no reason why **break** should be a reserved keyword in C programming language. Because **break** keyword is meaningless outside of loop and switch constructs, it should only be recognised as a keyword inside of bodies of such constructs. However, due to lexer and parser limitations that is not the case. However, by using EVM it would become possible to dynamically activate the rule for **break** keyword only inside a looping construct body. Similarly, the **return** keyword (and the grammar rule for it) could be activated only within a function body and so on.

1.7.2 Extensions to EVM grammar language

To enable domain manipulation within EVM, additional grammar elements are required. They are listed in table 15:

- Domain definitions are used to create new domains within a grammar.
- Domain annotations for grammar rules allow specifying the domain set under which the grammar rule should be considered active. If the domain annotation is not provided, then the rule is considered to be always active.
- Domain activation statements are used to temporarily activate new domains. If there there are parse statements within domain activation body, then the active domain set is inherited by the callees.

Example of a simplified grammar that uses domains to enable **break** statement only within the body of a loop statement is provided in fig. 8.

By adding every rule of a language extension to a specific domain, it is possible to enable or disable the entire language extension with a single statement.

Figure 8: Example domain usage

```
domain loop

@domains loop
rule statement
  parse "break"
end

rule while_loop
  parse "while" expr
  with_domains loop
    parse statement+
  end
  parse "end"
end
```

1.7.3 Compiling EVM grammars with domains

The most complex operation in EVM regarding domains is new domain activation. It is not enough just to add a simple instruction pair to enable and disable new domains: **with_domains** statements may be nested recursively, as such repeated domain activations should not affect the active domain set. Similarly, upon leaving the **with_domains** block, the only those domains should be disabled, which have been previously enabled within the same block.

Therefore, the following new instructions are required to enable domain support in EVM:

- `i_dom_push_active` pushes the active domain set to the stack of the current fiber.
- `i_dom_enable` *dom* enables the domain *dom* by adding it to the active domain set.
- `i_dom_enable_dyn` pops the target domain from the stack and enables it by adding the domain to the active domain set.
- `i_dom_disable` *dom* disables the domain *dom* by removing it from the active domain set.
- `i_dom_restore` *n* restores the active domain set by retrieving it from stack slot *n* of the current fiber.

These instructions can be used to compile the **with_domains** statement, as shown in table 16. $stack_slot_{dom}$ refers to the stack slot that contains the previous active domain set pushed by `i_dom_push`.

1.7.4 Loading multiple grammar modules in EVM

Whenever using EVM to parse a language, the base variant of that language most likely will be contained in a single compiled grammar module that will be loaded into EVM

Table 16: Rule for compiling domain activation statement

Element name	Grammar element	Instruction sequence
Domain activation statement	**with_domains** dom_1 ... dom_n *body* **end**	`i_dom_push_active` `i_dom_enable` dom_1 ... `i_dom_enable` dom_n $code(body)$ `i_dom_restore` $stack_slot_{dom}$

during EVM initialization. Language extensions then could be contained in separate grammar modules that can be both generated and loaded dynamically during parsing.

Loading multiple grammar modules in EVM is not trivial, as each grammar module has it's own address space. To support multiple address spaces within EVM the instruction pointer can be extended to include the module index. That way each instruction pointer in EVM that is store internally (for example, the *ip* of a fiber) is a pair $\langle id_{mod}, ip \rangle$, where id_{mod} is the module index and *ip* is relative instruction pointer to the start of the module. All existing instructions would use relative instruction pointers (such as `i_fork`, `i_match_char`, etc).

In practise, for performance reasons several bits of instruction pointer can be reserved for storing the module index. That way the instruction pointer could remain word-sized.

Additionally, all the grammar rules of any language extension should belong to a corresponding extension domain. That way language extensions could be enabled dynamically only for desired scopes with `i_dom_enable_dyn` instruction.

Additional instructions that work with absolute instruction pointers may be added in future if necessary for performance reasons.

1.7.5 Parsing reflective grammars in EVM

The mechanisms described in this chapter can be used to implement adaptable/reflective grammars by applying the following steps:

1. Define the base language. During this step grammar for the base programming language should be defined. This could an existing programming language (such as C) or entirely new one.

2. Define the extension metalanguage within the base language. EVM does not provide a specific extension metalanguage, as the extension metalanguage should be defined to match the syntax of the base language (however, the extension metalanguage could be designed to be similar to the EVM grammar language). The extension metalanguage should include extension activation construct for activating defined language extensions.

3. Implement compilation of metalanguage language extension node into a grammar module as described in this chapter. If the extension language matches EVM gram-

mar language, then the rules for compiling EVM grammar language elements can be used directly to implement this compilation step.

4. Implement the extension activation construct by adding a foreign call, which would lookup the target extension grammar module in host environment. After finding the target grammar module, it should be loaded into EVM. The foreign call should return the domain index for the extension. The domain of the extension then can be activated with `i_dom_enable_dyn` instruction within the extension activation construct. At this point EVM becomes capable of parsing constructs defined in the previously specified extension.

1.8 EVM performance improvements

In this chapter we describe several EVM optimizations that significantly increase the overall parsing performance (both in term of CPU time and memory usage).

1.8.1 Garbage collection of suspended fibers

EVM currently creates a state for every input position where other non-terminal rules are invoked with `i_call` instruction family. This state information is then used to record execution trace, to store reduction information and the most importantly to park suspended fibers so they may be resumed later. All this information over time adds to a significant amount. However, not all of it is needed for further parsing. 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 unnecessary, together with the suspended fibers they contain, may be discarded before the parsing process completes.
- The only the reduction instructions access variables from previous states.
- 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 to avoid creating duplicate fibers. 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.

Such 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. To reduce the garbage collector's performance impact to the parsing process, the garbage collector could be run every *n* parsed terminal symbols.

1.8.2 Eliminating dynamic non-terminal call indirection

Rules for parsing non-terminals in EVM are invoked with `i_call_dyn` and then are matched with `i_match_dyn` instruction. However, both of these instructions perform significant amount of redundant work:

- The list of candidate rules is fetched from *rule_map* map.
- The candidate rules are filtered based on current active domain set.
- The candidate rules are filtered based on minimum rule precedence.

If the the activate domain set for a specific call is known during compile time, then the instruction pointers for target rule entry points and reduction tags can be computed during compile time. As such it becomes no longer necessary to perform dynamic rule lookup and filtering during parse time. Therefore dynamic instructions `i_call_dyn` and `i_match_dyn` can be replaced into corresponding static ones: `i_call` and `i_match_sym`.

`i_call` ip_{target}, n is a new instruction that invokes non-terminal rule with entry point ip_{target} and n arguments.

1.8.3 On-demand instruction subset construction

1.8.3.1 Importance of subset construction

EVM is based on Earley parser and therefore inherits some of it's flaws. One of the main reasons why Earley parser in it's original form is not used for parsing programming languages is it's inefficiency.

One of the more common tasks of parsing programming languages is parsing expressions. Even older programming languages (such as C++) have huge operator hierarchies with many precedence levels. For example, C++ language has:

- 12 arithmetic operators.
- 6 comparison operators.
- 3 logical operators.

- 6 bitwise operators.
- 10 compound operators.
- 7 member and pointer operators.

That's a total of 44 distinct operators. This list does not include around 20 more operators that are more difficult to classify. This means, that if EVM was used to implement a C++ parser and if every operator was defined in a separate rule, every time an expression could be encountered, EVM would create around 50 fibers to parse **a single** expression. Roughly a quarter of these expressions are prefix operators, so corresponding fibers would be discarded as soon as the first character was parsed. The remaining fibers would be suspended to parse the first operands of unary (postfix) and binary operators. After completing that operand and parsing the character(s) that represent the binary operator (such as +, -, *, etc), all but one of the remaining fibers would be discarded.

This is a huge issue that prevents usage of EVM for any practical application. 50 fiber creations, 35 suspensions, additional 35 fiber creations after resuming the suspended fibers just to parse a single binary expression. This problem also affects the original Earley parser. To combat this inefficiency, an efficient variation of Earley parser has been produced.

The way EVM currently operates can be similar to a non-deterministic finite automaton: just like a NFA can be in multiple states at the same time, so does EVM can execute multiple fibers at the same time. But it is well known that any NFA can be converted into DFA by applying the process known as subset construction. The Faster Earley Parser [MH96] or Efficient Earley Parser with Regular Right-hand Sides [JM10] are both based on this algorithm. By applying such parsing algorithms to parse C++, it would no longer take 50 distinct fibers (or items in Earley parser case) to parse a single expression: all 50 grammar rules could me merged into 1 optimized rule.

Because EVM is unique that it uses instruction sequences to represent grammars, the traditional subset construction or their modifications for Earley parser cannot be applied directly to EVM. Furthermore, EVM is capable of loading and enabling additional grammars during parsing, therefore subset construction needs to applied on-demand for only those grammar rules that are about to be used for parsing. As such, specialized subset construction algorithm for EVM grammar modules that supports all existing features of EVM needs to be created.

1.8.3.2 Instruction ε-closures

The first step of subset construction is computation of an ε-closure. ε-closure in automata theory is a set of states in NFA reachable from initial state by ε transitions. The ε-closure always includes the initial state as well.

Similarly, in EVM we can define instruction ε-closure as a set of instruction pointer and active domain set pairs, which are reachable from initial instruction pointer with initial activate domain set by executing only *unordered* instructions.

unordered instructions are instructions whose order of execution doesn't affect the outcome of computation (or parsing). For example, `i_call` and `i_fork` are unordered instructions, because a block of such instructions can be executed in any order without affecting the result.

For efficiency reasons, `i_dom_enable` is considered to be partially unordered. By including this instruction into the set of unordered instructions, it can be optimized away completely by tracking the changes of current active domain set. This way the overhead of being able to parse adaptable grammars can be mostly eliminated (adaptable grammars still need to compiled into grammar modules and then loaded into EVM).

Instruction closure computation begins with a set of initial *domain addresses*. A *domain address* is an instruction pointer and active domain set pair. All of the initial domain addresses are placed into a queue. Then appropriate actions are executed for each element of the queue based on the instruction which is referenced by instruction pointer of the current element.

There are two possible actions:

- **continue** *da* action adds the domain address *da* to the queue if it's not already present.
- **relevant** *da* action adds the domain address *da* to the resulting instruction closure set.

The actions to be executed for each instruction are provided in table 17. *ip* and *ads* refer to instruction pointer and active domain set of the current entry correspondingly. $entries(A, ads)$ refers to the set of rule entry points for non-terminal A with current active domain set ads.

1.8.3.3 Merging instruction ε-closures

The goal of merging instruction ε-closures is twofold: merger of similar instructions to avoid duplicate computation and elimination of dynamic elements that can reduce parsing performance.

Because of the second goal, dynamic instructions like `i_call_dyn` and `i_match_dyn` are replaced with their static counterparts. In general, all instructions are merged based on *instruction merger key*. If two instructions share the same instruction merger key then they can be merged into a single instruction. The instruction merger keys can be derived from rules provided in table 18.

Once the merger keys have been computed for all instructions in the ε-closure, similar instructions can be merged. Each type of instructions is merged differently:

- `i_match_chars` *table* instructions are merged by merging their jumptables: transitions that share the same character are merged by computing their ε-closure and optimizing it. The resulting instruction is a `i_match_chars`.

Table 17: Rules for computing instruction closures

Instruction	Action
`i_br` *target*	continue $\langle target, ads \rangle$
`i_call` *target, n*	If call visitation is disabled: relevant $\langle ip, ads \rangle$ continue $\langle ip+1, ads \rangle$ If call visitation is enabled: continue $\langle target, ads \rangle$ continue $\langle ip+1, ads \rangle$
`i_call_dyn` *A, n*	If call visitation is disabled: relevant $\langle ip, ads \rangle$ continue $\langle ip+1, ads \rangle$ If call visitation is enabled: continue $\langle target, ads \rangle, \forall target \in entries(A, ads)$ continue $\langle ip+1, ads \rangle$
`i_dom_disable` *dom*	continue $\langle ip+1, ads \setminus dom \rangle$
`i_dom_enable` *dom*	continue $\langle ip+1, ads \cup dom \rangle$
`i_fork` *target*	continue $\langle target, ads \rangle$ continue $\langle ip+1, ads \rangle$
`i_reduce` *A, n*	relevant $\langle ip, ads \rangle$ continue $\langle ip+1, ads \rangle$
`i_stop`	
All others	relevant $\langle ip, ads \rangle$

Table 18: Rules for computing instruction merger keys

Instruction	Merger key
`i_call` *target, n*	$\langle$"*call*"$, n\rangle$
`i_call_dyn` *A, n*	$\langle$"*call*"$, n\rangle$
`i_match_chars` *table*	$\langle$"*match_chars*"$\rangle$
`i_match_dyn` *A*, $prec_{min}$	$\langle$"*match_syms*"$\rangle$
`i_match_syms` *table*	$\langle$"*match_syms*"$\rangle$
`i_reduce` *A, prio*	$\langle$"*reduce*"$, A\rangle$
`i_reduce_r` *A, prio*	$\langle$"*reduce_r*"$, A\rangle$
All others: $instr\ arg_1, \ldots, arg_n$	$\langle instr, arg_1, \ldots, arg_n \rangle$

- `i_match_dyn` and `i_match_syms` instructions are merged into a single `i_match_syms`. The merger process works similarly to the merger of `i_match_chars`: instructions are merged by merging their jumptables. In case of `i_match_dyn` (which has no jumptable argument), jumptables are computed based on `i_match_dyn` operands and active domain set. Then transitions that share the same non-terminal symbol are merged by computing their ε-closure and optimizing it.

- `i_call` and `i_call_dyn` are merged into a single `i_call_opt` or a `i_call` instruction. This is done by computing ε-closure of entry points of the target non-

terminal. If optimized instruction sequence for resulting ε-closure already exists, then a direct call with `i_call` to that instruction sequence is generated. Otherwise `i_call_opt` *closure* is generated. *closure* refers to the target ε-closure. This instruction is used to avoid subset construction of the entire grammar module. Only upon executing `i_call_opt` the optimized (subset constructed) version for the closure is generated, thus making instruction subset construction process only run on-demand.

- `i_reduce` *A* instructions are merged simply based on reduction non-terminal into a single `i_reduce` instruction. This way duplicate reductions with the same non-terminal get eliminated.

Other instructions are merged by adding them to *instruction blocks* and merging matching prefixes of these blocks. Instruction block is a sequence of instructions that terminates with a terminator instruction. All control transfer instructions are block terminator instructions. That includes instructions like `i_br`, `i_match_chars`, `i_match_syms`, etc. This is necessary, because many EVM instructions are executed sequentially and have no way to transfer control to arbitrary position.

Once instructions are merged, they can be outputted to a target grammar module. Resulting instructions are outputted in a specific order:

1. Unordered instructions: `i_call` and `i_reduce`.
2. $n-1$ `i_fork` instructions for the following n ordered instructions.
3. n ordered instructions.
4. `i_stop` instruction if $n = 0$.

An example of optimized (subset constructed) instruction sequence is provided in table 19. The resulting instruction sequence is longer, however it is more deterministic. For example, it can be seen at offset 16 of optimized instruction sequence, that prefixes for addition and multiplication have been merged successfully and that it will take a single instruction at offset 16 to match the binary operator, at which point parsing diverges based on matched operator.

1.9 Conclusions

In this chapter we have presented Earley Virtual Machines: a virtual machine-based and Earley parser inspired parsing method that:

- Can parse arbitrary context-free grammars.
- Supports regular right hand sides in production rules.
- Supports regular lookahead.

Table 19: Subset construction example

Source grammar	Compiled grammar	Optimized grammar

rule A[0]
  parse A "+" *A
end
 	
rule A[5]
  parse A "*" *A
end
 	
rule A[10]
  parse "b"
end
	10: i_call_dyn "A", 1
	11: i_match_dyn "A", 1
	12: i_match_char '+' -> 13
	13: i_call_dyn "A", 0
	14: i_match_dyn "A", 0
	15: i_reduce "A0", 0
	16: i_stop
	
	20: i_call_dyn "A", 6
	21: i_match_dyn "A", 6
	22: i_match_char '+' -> 23
	23: i_call_dyn "A", 5
	24: i_match_dyn "A", 5
	25: i_reduce "A1", 0
	26: i_stop
	
	30: i_match_char '+' -> 31
	31: i_reduce "A2", 0
	32: i_stop
		01: i_call 30
		03: i_fork 25
		05: i_match_syms "A1" -> 7,
		  "A2" -> 16
		07: i_match_chars '+' -> 9
		09: i_call 38
		11: i_match_syms "A0" -> 13,
		  "A1" -> 13, "A2" -> 13
		13: i_reduce "A0"
		15: i_stop
		16: i_match_chars '*' -> 18,
		  '+' -> 9
		18: i_call 30
		20: i_match_syms "A1" -> 22,
		  "A2" -> 22
		22: i_reduce "A1"
		24: i_stop
		25: i_match_chars 'b' -> 27
		27: i_reduce "A2"
		29: i_stop
		30: i_fork 36
		32: i_match_syms "A2" -> 34
		34: i_match_chars '*' -> 18
		36: i_match_chars 'b' -> 27
		38: i_fork 42
		40: i_match_syms "A1" -> 7,
		  "A2" -> 16
		42: i_match_chars 'b' -> 27

- Supports adaptive grammars by dynamically loading end enabling new grammars during parsing.
- Provides multiple means for abstract syntax tree construction.
- Supports data-dependant constraints.
- Supports subset construction optimization that can be used to increase determinism and reduce number of dynamic elements executed during parsing.

All of these EVM features provide sufficient means for implementing a parser for a REP language.

References

[AH02] John Aycock and R. Nigel Horspool. Practical earley parsing. *Compututer Journal*, 45:620–630, 2002.

[BS07] Claus Brabrand and Michael I. Schwartzbach. The metafront system: Safe and extensible parsing and transformation. *Science of Computer Programming*, 68(1):2–20, August 2007.

[Bur75] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, October 1975.

[CV14] Walter Cazzola and Edoardo Vacchi. On the incremental growth and shrinkage of lr goto-graphs. *Acta Informatica*, 51(7):419–447, 2014.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[EKV09] Giorgios Economopoulos, Paul Klint, and Jurgen Vinju. *Faster Scannerless GLR Parsing*, pages 126–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[For02] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.

[For04] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, 2004.

[Gri04] Robert Grimm. Practical packrat parsing. Technical report, New York University, Computer Science, 2004.

[JM10] Trevor Jim and Yitzhak Mandelbaum. Efficient earley parsing with regular right-hand sides. *Electronic Notes in Theoretical Computer Science*, 253(7):135 – 148, 2010.

[JM11] Trevor Jim and Yitzhak Mandelbaum. Delayed semantic actions in yakker. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, LDTA '11, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[JMW10] Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 417–430, New York, United States, 2010. ACM.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607 – 639, 1965.

[KVW10] Lennart C.L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. *SIGPLAN Not.*, 45(10):918–932, October 2010.

[MH96] Philippe McLean and R. Nigel Horspool. A faster earley parser. In *Proceedings of the 6th International Conference on Compiler Construction*, CC '96, pages 281–293, London, UK, UK, 1996. Springer-Verlag.

[RBDIA14] Leonardo V.S. Reis, Roberto S. Bigonha, Vladimir O. Di Iorio, and Luis Eduardo S. Amorim. The formalization and implementation of adaptable parsing expression grammars. *Sci. Comput. Program.*, 96(P2):191–210, December 2014.

[Sco08] Elizabeth Scott. Sppf-style parsing from earley recognisers. *Electronic Notes in Theoretical Computer Science*, 203(2):53 – 67, 2008.

[SJ05] Elizabeth Scott and Adrian Johnstone. Generalized bottom up parsers with reduced stack activity. *Comput. J.*, 48(5):565–587, September 2005.

[SJ06] Elizabeth Scott and Adrian Johnstone. Right nulled glr parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, July 2006.

[SW11] Paul Stansifer and Mitchell Wand. Parsing reflective grammars. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, LDTA '11, pages 10:1–10:7, New York, United States, 2011. ACM.

[Tom85] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.