Baltic J. Modern Computing, Vol. 7 (2019), No. 2, pp. 171–189
https://doi.org/10.22364/bjmc.2019.7.2.01

Parsing with Scannerless Earley Virtual Machines

Audrius Šaikūnas

Institute of Data Science and Digital Technologies, Vilnius University, Akademijos 4, LT-08663 Vilnius, Lithuania

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 show how to translate grammars into virtual machine instruction sequences that are then used by the parsing algorithm. Additionally, we introduce an optimization that merges shared rule prefixes to increase parsing performance. Finally, we present and evaluate an implementation of Scannerless Earley Virtual Machine called `north`.

Keywords: implementation, scannerless parsing, virtual machine

1 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 Scannerless Earley Virtual Machine, or SEVM for short. It is a continuation of Earley Virtual Machine (Šaikūnas, 2017). It is a virtual machine based parser, heavily based on the original Earley parser (Earley, 1970) and inspired by Regular Expression Virtual Machine (WEB, a). SEVM is capable of parsing context-free languages with data-dependant constraints. The core idea behind of SEVM 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 *medium-level intermediate representation (MIR) grammars* that are executed or interpreted by the SEVM to match various input sequences.

In chapter 2 we present the Scannerless Earley Virtual Machine. Then, in chapter 3 we present the primary optimization for SEVM, which significantly improves parsing performance. Finally, in chapter 4 we present and evaluate an implementation of SEVM.

2 Scannerless Earley Virtual Machine

2.1 Overview of the parsing process

The parsing process consists of the following primary steps:

1. Translation of input grammar to MIR (medium-level intermediate representation). In this step, the textual representation input grammar is parsed, analysed for semantic errors, and finally grammar MIR is generated.
2. SEVM initialization: the input data is loaded, necessary data structures for parsing are initialized.
3. Parser execution: the grammar MIR is either interpreted, or translated into machine code via just-in-time (JIT) compiler and then executed natively.
4. Optimization: during parser execution, upon invoking a grammar rule, it may be optimized using subset construction, potentially merging shared prefixes of multiple rules to increase parsing performance.

2.2 SEVM structure

A SEVM parser is a tuple $\langle chart, exec\_stack \rangle$:

- $chart$ is an index map from parse positions to *chart entries*. An index map is a map that also assigns unique indices to values, allowing to lookup values by both keys or indices.
- $exec\_stack$ is the primary execution stack. It stores chart entry indices, which contain at least one active task. The top element of the stack stores the index of *currently active task*.

A *chart entry* is a tuple $\langle reductions, running, susp\_task \rangle$:

- $reductions$ is a list of non-terminal reductions.
- $running$ is a stack that stores active tasks.
- $susp\_task$ is a list of suspended tasks paired with the conditions for waking them.

A *task* in SEVM is an instance of a (compiled) grammar rule: it represents the progress of parsing a specific grammar rule at a specified input position. More formally, a task is a state machine that can be represented with a tuple $\langle\ state\_id, origin, position, tree\_id, grammar\_id\ \rangle$:

- $state\_id$ is the state index of task's state machine.
- $origin$ is the origin position of the task (the starting input position).
- $position$ is the current input position. Immediately after creation $origin$ is equal to $position$.
- $tree\_id$ is the index of resulting the parse-tree node.
- $grammar\_id$ is the grammar index. SEVM supports parsing inputs with multiple grammars, some of which may be loaded/created dynamically during parsing. This index refers to the grammar used by the current task.

A *suspended task* task is a task that has been suspended as a result of another rule invocation: when a task invokes a rule for parsing another non-terminal in SEVM, it gets suspended until the task for parsing the callee completes, at which point the caller is resumed. A suspended task is a tuple $\langle\ task, pos\_spec\ \rangle$:

- $task$ is the task that has been suspended.
- pos_spec is the positive match specifier: a list of conditions for resuming this task. Each condition entry contains $match\_id$, min_prec and $state\_id$: $match\_id$ is a non-terminal match index that allows to match reduction indices. min_prec represents the minimum precedence value of that match. $state\_id$ is a state index, in which this task is to be resumed, should a matching reduction occur.

A *reduction* represents a segment of input where a rule (non-terminal) successfully matched. More formally, it is a tuple $\langle\ reduce\_id, length, tree\_id\ \rangle$:

- $reduce\_id$ is the reduction index. It represents a concrete non-terminal symbol which has been matched at the position of the current chart entry.
- $length$ is the length of the match in bytes.
- $tree\_id$ is the parse-tree index that represents the match.

2.3 MIR structure

Compiled/preprocessed grammars are stored in medium intermediate representation, or MIR for short. MIR is a abstract syntax tree-like structure that stores the grammars in a single static assignment (SSA) form.

More precisely, a rule in SEVM MIR is represented as a list of basic blocks. Each basic block contains 0 or more statement instructions and terminates with exactly one control instruction. Statement instructions do not alter control flow of execution, while control instructions do. Instructions are always executed in a context of a task.

There are several primary statement instructions:

- `StmtReduce` $reduce\_id$ creates a reduction with reduction index $reduce\_id$. The reduction length is computed by subtracting the current task $position$ from task $origin$. The reductions are stored in the origin chart entry (the chart entry whose position is $origin$). Duplicate reductions are ignored.
- `StmtRewind` n rewinds the current task by n characters/bytes. It does so by subtracting n from the $position$ of the current task.
- `StmtCallRuleDyn` $call\_spec$ creates a new task for parsing rule(s) denoted by call specifier $call\_spec$. The call specifier is a list of $match\_id$ and min_prec pairs, where $match\_id$ refers to the target non-terminal rule (either abstract or concrete) and min_prec is the minimum precedence. Duplicate calls or fully overlapping calls (whose call specifier is a subset of the union of previously performed calls at the current position) are ignored.

Additional statement instructions can be added to allow general purpose computation during parsing (such as arithmetic, logic, memory instructions).

There are several primary control instructions:

- `CtlStop` terminates the currently running task.
- `CtlBr` B unconditionally transfers execution to basic block B.
- `CtlFork` $B_1, B_2, ..., B_n$ forks the the execution of the current task to basic blocks $B_1, B_2, ..., B_n$. This is typically done by pushing the copies of the current task to $running$ with updated $state\_id$ values that correspond to basic blocks $B_n, B_{n-1}, ..., B_2$. Then the currently running task continues to B_1. This instruction is used to fork the current task into several tasks to traverse several different alternative parse paths. Multiple successful parse paths mean that the currently parsed fragment is ambiguous.
- `CtlMatchChar` $c \rightarrow B_{pos}, B_{neg}$ is used to match terminal symbol c: c is matched against the terminal symbol at $position$. If c matches $input_{position}$, then the $position$ of the current task is increased by 1 and execution is resumed in B_{pos}. Otherwise, the execution is resumed in B_{neg}. This instruction loosely corresponds to the *shift* action of LR parsers, or *scanner* step of Earley parsers.
- `CtlMatchClass` $a_1..b_1 \rightarrow B_1, a_2..b_2 \rightarrow B_2, ..., a_n..b_n \rightarrow B_N, else \rightarrow B_{fail}$ works similarly to `CtlMatchChar`, but can be used to match multiple characters at the same time. If the current input character is in interval $a_i..b_i$ then the control is transferred to basic block B_i. If none of the intervals match, then execution is transferred to B_{fail}. The input intervals may not overlap. This instruction is typically implemented by using a transition table to quickly match the input character against multiple intervals.
- `CtlMatchSym` pos_spec is used to match non-terminal symbols using match specifier pos_spec. This is done by suspending the current task (adding it to $susp\_tasks$ of chart entry with matching $position$). The match specifier is in form $\langle M_1, P_1 \rangle \rightarrow B_1, \langle M_2, P_2 \rangle \rightarrow B_2, ..., \langle M_n, P_n \rangle \rightarrow B_n$. Then, if a reduction occurs, which starts at $position$ with reduction index $reduce\_id$, this task is *resumed* in basic block B_i, if $reduce\_id$ matches $\langle M_i, P_i \rangle$, where M_i is a match index and P_i is the minimum precedence value of that match. It is important to note that this instruction, unlike

`CtlMatchChar` and `CtlMatchClass`, is non-deterministic and a single reduction may cause a single suspended task to be resumed multiple times in different positions.

Suspended tasks are resumed by making a copy of that task in appropriate $state\_id$ and adding it to $running$ of the origin chart entry.

To match reduction index R against match index M and minimum precedence value P, a match table is used. Each grammar contains one match-table MT that stores entries $\langle M_1, P_1, R_1\rangle, ..., \langle M_n, P_n, R_n\rangle$. If $\langle M, P, R\rangle \in MT_{grammar\_id}$, where $MT_{grammar\_id}$ is the match table of grammar with index $grammar\_id$, then match index M with minimum precedence value P **matches** reduction index R in grammar with index $grammar\_id$.

Such match/reduce index separation enables to dynamically create/modify grammars during parsing, enabling to parse adaptable (reflective) grammars (Stansifer and Wand, 2011). Duplicating an existing match table and adding new entries effectively extends the grammar. Conversely, duplicating an existing match table and removing entries from it shrinks the grammar. Both of these operations do not modify the original grammar and thus allow to parse input fragments, where grammar extensions are short-lived and may apply to only a specific block of input. Even more interestingly, the starts of such blocks may be ambiguous: if a start of a block with updated grammar is ambiguous, then the execution may be forked into two tasks: one with the original $grammar\_id$, where the grammar is unchanged and another with updated $grammar\_id$ and corresponding match table. This effectively causes the same input fragment to be parsed with two completely separate grammars. Just like with `CtlFork` instruction, if both parse paths complete successfully, then the parse input is ambiguous.

2.4 Grammar description language

```
rule main() {
  parse "q";
}
```

(a) concrete grammar rule

```
rule_dyn expr();

#[part_of(expr, 50)] rule add() {
  parse (expr!, "+", expr);
}

#[part_of(expr, 100)] rule zero() {
  parse "0";
}
```

(b) abstract grammar rule with 2 members

Fig. 1: Terminal symbol matching example

Even though SEVM is capable of parsing all context-free languages and thus could use BNF/eBNF/YACC as the input language for grammars, a new grammar description

language has been created for SEVM to expose additional features typically not present in YACC-like parsers. Existing YACC grammars can be trivially rewritten to SEVM grammars, as SEVM grammars are a superset of YACC grammars. Some of SEVM grammar language features are presented in this chapter, but many others are beyond the scope of this paper as they rely on additional parser features that are not elaborated in this paper.

The primary unit of grammar composition in SEVM is a grammar rule. There are two types of grammar rules:

- *Concrete grammar rules* (commonly referred to as just *rules*): each rule defines a single non-terminal and contains body, which is composed out of statements.
- *Abstract grammar rules* do not have a body, but other concrete grammar rules may be added with `#[part_of(...)]` attribute to the abstract grammar rule as members. Invoking an abstract grammar rule causes all its members to be invoked, which is alternative way of writing $M_1|M_2|...|M_n$, where M_i is a member of the abstract grammar rule. This construct provides an extension point for composing multiple grammars. See fig. 1 for an example of abstract grammar expression. Additionally, each member of an abstract grammar rule has an associated numeric precedence value (ranging from 0 to 255), which enables to implement operator precedence and associativity.

In the SEVM version described in this paper, only one statement exists: the **`parse`** statement, which contains a single *grammar expression*. Additional statements may be added to SEVM (such as loops, conditionals, variable declarations, etc) to enable imperative control of parsing process.

A *grammar expression* defines a pattern which may be matched/parsed against the input. There are several grammar expressions in SEVM:

- String literal grammar expressions: `"text"`. They allow to match a sequence of characters and are translated into a sequence of `CtlMatchChar` instructions.
- Character class grammar expressions: `r"a-zA-Z"`. They allow to match a single input character. They are analogous to character classes or character sets found in regular expressions. Each character class grammar expression is translated into a single `CtlMatchClass` instruction.
- Direct rule call grammar expression: `A`, where `A` is another concrete grammar rule. They enable to match non-terminal symbols.
- Non-associative rule call grammar expression: `A`, where `A` is another abstract grammar rule. If the grammar expression is a descendant of a member of `A` with precedence P, then `A` is invoked with precedence $P+1$. Otherwise, the precedence value is 0.
- Associative rule call grammar expression: `A!`, where `A` is another abstract grammar rule. If this expression is a descendant of a member of `A` with precedence value P, then the `A` is invoked with P.
- Sequence grammar expressions: $E_1, E_2, ..., E_n$, where E_i is another grammar expression. They allow to compose multiple grammar expressions into a sequence.
- Zero-or-one grammar expression: `E?` enables to optionally match grammar expression `E`.

- Zero-or-more grammar expression: `E*` enables grammar expression `E` to be matched zero or more times.
- One-or-more grammar expression: `E+` enables grammar expression `E` to be matched one or more times.
- Grouping grammar expression: `(E)` allows to group grammar expression `E`. The grouping grammar expression has no additional semantics other than overriding operator precedence of other grammar expressions.

2.5 Matching terminal symbols

As mentioned in section 2.3, terminal symbols at MIR level in SEVM are matched with `CtlMatchChar` and `CtlMatchClass` instructions. Sequences of terminal symbols are matched with sequences of corresponding `CtlMatchChar` and `CtlMatchClass` instruction sequences. Optional matching as well as repetition is expressed additionally with `CtlFork` instruction.

```
rule main() {
  parse "hello";
}
```

(a) input grammar

```
main: {
  #0: CtlMatchChar 'h' => #2, #1
  #1: CtlStop
  #2: CtlMatchChar 'e' => #3, #1
  #3: CtlMatchChar 'l' => #4, #1
  #4: CtlMatchChar 'l' => #5, #1
  #5: CtlMatchChar 'o' => #6, #1
  #6: StmtReduce R(:main)
      CtlStop
}
```

(b) MIR of input grammar

Fig. 2: Terminal symbol matching example

An example grammar for matching terminal symbol sequence `hello` and its MIR are shown in fig. 2, where `R(:main)` refers to the reduction index of rule `main`. Each character of matched sequence is transformed to a corresponding `CtlMatchChar` instruction, that on success proceeds to parse the next character and on failure transfers to basic block `#1`, which ultimately executes `CtlStop` that terminates the task.

2.6 Matching non-terminal symbols

Parsing non-terminals in SEVM is significantly more complicated and the process involves at least 3 different instructions. Typically, the process of matching a non-terminal (calling a grammar rule) can be summarized in the following steps:

1. A task for parsing a non-terminal is created and queued with `StmtCallRuleDyn` instruction.

2. The caller is suspended with `CtlMatchSym` instruction.
3. The callee (the newly created task) is executed.
4. Eventually, the callee (if matching was successful) executes `StmtReduce` instruction, which causes the caller to be resumed (by making a copy of it with updated $state\_id$ and pushing it to the top of $running$ in the appropriate chart entry).

Step 1 may be skipped if the callee, represented by call specifier, was invoked at this position before. Similarly, `CtlMatchSym` instruction in step 2 may immediately wake the caller if there already one or more matching reductions exist at the input position of the call.

```
rule other() {
  parse "a";
}

rule main() {
  parse (
    other,
    other,
    other
  );
}
```

(a) input grammar

```
other: {
  #0: CtlMatchChar 'a' => #2, #1
  #1: CtlStop
  #2: StmtReduce R(:other)
      CtlStop
}

main: {
  #3: StmtCallRuleDyn <M(:other), 0>
      CtlMatchSym <M(:other), 0..255> => #4
  #4: StmtCallRuleDyn <M(:other), 0>
      CtlMatchSym <M(:other), 0..255> => #5
  #5: StmtCallRuleDyn <M(:other), 0>
      CtlMatchSym <M(:other), 0..255> => #6
  #6: StmtReduce R(:other)
      CtlStop
}
```

(b) MIR of input grammar

Fig. 3: Non-terminal symbol matching example

An example grammar and corresponding MIR for matching a sequence of non-terminal symbols is shown in fig. 3. In the example MIR `M(:other)` refers to match index of rule `other`. Each call to `other` rule in `main` is translated into a pair of `StmtCallRuleDyn` and `CtlMatchSym` instructions. In each instance `StmtCallRuleDyn` creates a task for parsing `other` with minimum precedence of 0. Similarly, each `CtlMatchSym` suspends the current task, effectively causing execution control to yield to callee, which eventually performs `StmtReduce` and terminates with `CtlStop`.

2.7 Matching repetition

Repetition of terminal, non-terminal symbols and their combinations is performed identically. All repetitions in SEVM are expressed with `CtlFork` instruction:

- Optional grammar expression $A?$ (where A is any other grammar expression) forks the execution into two parse paths: one, where A matched 0 times, effectively skipping it, and where it is matched 1 time.
- Zero-or-more grammar expression $A*$ is translated similarly to $A?$, but after successfully parsing A, the execution is unconditionally transferred back to the beginning of $A*$, causing the parsing of A to loop.
- One-or-more grammar expression $A+$ first attempts to parse A, and upon successful match, the execution is forked into two paths: one back to the beginning of $A+$ and one to the successor of $A+$ (which may be a `StmtReduce` instruction).

```
rule zero_or_one() {
  parse "a"?;
}

rule zero_or_more() {
  parse "a"*;
}

rule one_or_more() {
  parse "a"+;
}
```

(a) input grammar

```
zero_or_one: {
   #0: CtlFork #1, #3
   #1: CtlMatchChar 'a' => #3, #2
   #2: CtlStop
   #3: StmtReduce R(:zero_or_one)
       CtlStop
}
zero_or_more: {
   #4: CtlFork #5, #8
   #5: CtlMatchChar 'a' => #7, #6
   #6: CtlStop
   #7: CtlBr #4
   #8: StmtReduce R(:zero_or_more)
       CtlStop
}
one_or_more: {
   #9: CtlMatchChar 'a' => #11, #10
  #10: CtlStop
  #11: CtlFork #9, #12
  #12: StmtReduce R(:one_or_more)
       CtlStop
}
```

(b) MIR of input grammar

Fig. 4: Repetition matching example

See fig. 4 for example MIR for parsing each of these repetition operators. For simplicity sake, the repeating element in each rule is terminal character `a`, but it may contain any grammar expression.

2.8 The parsing algorithm

Because the majority of parser work is performed within various parsing instructions, the overall parsing algorithm of SEVM is quite simple:

1. Pop the current task from the $running$ of the currently active chart entry:
   - If the current $running$ is empty, then remove top element from $call\_stack$.
   - If $call\_stack$ is empty, then terminate the parser.
2. Execute the current task either by interpreting the corresponding instructions, or by invoking the corresponding just-in-time compiled function that implements the rule. The execution of the current task terminates either with `CtlStop` instruction, which completely discards the current task, or with `CtlMatchSym` which stores the task in appropriate $susp\_tasks$ list.
3. Go to step 1.

After the parser terminates, chart entry of position 0 is inspected: if its reduction list contains a reduction with the starting non-terminal `main` with total length of the input, then the the parser successfully analyses the entire input. If no such reduction exists, then the parser completed unsuccessfully and the chart entry with the highest input position (more specifically, its $susp\_tasks$ list) may be analysed to determine which non-terminals failed to match and caused the parser to fail.

2.9 Obtaining parse forest

The result of SEVM parser is a shared packed parse forest (SPPF). In `north`, SPPF is internally represented by a structure similar to a binary tree.

Each task constructs a corresponding SPPF node during its execution. When a new task is initially constructed, its $tree\_id$ points to an empty node. `CtlMatchSym` instruction appends a new child node when a suspended task is resumed (when a non-terminal symbol is successfully matched) by creating a new *shift* node, which contains both the old $tree\_id$ value and newly matched child node. `CtlReduce` instruction creates a *reduction* node that represents a successful parse of a non-terminal.

The parse forest contains 4 types of nodes:

1. **Empty nodes**. Newly constructed tasks contain an empty $tree\_id$.
2. **Shift nodes**. A shift node is a binary node that contains previous node and a newly added node.
3. **Reduce nodes**. Reduce node contains previous $tree\_id$ and source range (the start and end offsets) of the reduction.
4. **Alternative nodes** represent ambiguous parses. These nodes are similar to GLR's packing nodes.

When an ambiguous reduction occurs (a reduction, whose position, $reduce\_id$ and $size$ match), the original reduce node is converted into alternative node. Alternative nodes form a linked list out of corresponding ambiguous reduction nodes.

The root of the parse forest can be obtained by inspecting $reductions$ list of chart entry at position 0.

2.10 Parsing with constraints

By introducing additional instructions to SEVM, it is possible to parse grammars with context-dependent constraints. These constraints are not meant to encapsulate high-level language semantics (such as disambiguating identifiers from typenames in C), but rather to allow to parse non-context-free tokens. For example, Ruby programming language has DOCHERE multiline string tokens than start and terminate both with the same user-provided string (similar to matching opening and closing tags in XML).

```
rule backref() {
  parse text@
    ("a" | "b")+;
  parse " ";
  parse =text;
}
```

(a) input grammar

```
backref: {
  #0: CtlMatchChar
        'a' => #2, 'b' => #2, #1
  #1: CtlStop
  #2: CtlFork #0, #3
  #3: %text_end = StmtCurrPos
      CtlMatchChar ' ' => #4, #1
  #4: %text_start = StmtOriginPos
      CtlMatchDyn %text_start,
        %text_end => #5, #1
  #5: StmtReduce R(:backref)
      CtlStop
}
```

(b) MIR of input grammar

Fig. 5: Matching with backreferences

Figure 5 shows a simple grammar rule that uses backreferences: it defines a sequence of `a` and `b` characters, followed by a space, which is then followed by exact same sequence of `a` and `b` characters. The `var@expr` grammar expression allows to capture the input that matches `expr` to variable `var`. The `@` operator only records initial and ending positions of the matched input. If the initial position of `expr` matches the start of the rule, then rule origin position is used instead.

In MIR this is achieved by introducing 3 additional instructions:

- `StmtCurrPos` allows to query the current position of the current task.
- `StmtOriginPos` allows to query the origin position of the current task.
- `CtlMatchDyn` allows to dynamically match a specified interval of input and to transfer execution on successful/failed match.

More complicated context-dependant constraints may be added to grammars by introducing additional general-purpose instructions to MIR, however that is beyond the scope if this paper.

3 MIR subset construction optimization

3.1 Overview

In this chapter we present one of the most important optimizations for SEVM: MIR subset construction. It is based/inspired by the Practical Earley Parser (Aycock and Horspool, 2002) and Yakker (Jim et al., 2010).

The key idea is rather simple: normally, when countering a grammar expression `A | B | C`, where `A`, `B`, `C` represent non-terminals, each of those non-terminals would be parsed in turn. However this is inefficient, because at very least 3 calls and matches (which would result in suspended tasks) would be need to parse such grammar expression. Furthermore, it is possible that `A`, `B`, `C` may share a common prefix that would be re-parsed on each of invocation of corresponding non-terminal.

To alleviate this problem, MIR subset construction is used. Instead of performing 3 separate (first `A`, then `B`, then `C`) parses, all of these 3 grammar rules (more specifically, their MIRs) are merged ("optimized") into a single rule, which is then invoked instead. In this scenario, parsing `A | B | C` results in a single `StmtCallRuleDyn`, which will invoke the combined rule, and one `CtlMatchSym`, which will match any of the 3 non-terminals.

3.2 MIR ϵ-closures

Much like ϵ-closures used for converting NFAs to DFAs (Rabin and Scott, 1959), MIR ϵ-closures are used to facilitate conversion of non-optimized MIR to optimized MIR.

A MIR ϵ-closure of a *closure-seed* is a set of *relevant* instruction indices reachable from the *closure-seed* with no side effects.

A *closure-seed* is a set of MIR node indices (which may contain rule, basic block or instruction indices).

A *relevant* instruction is an instruction that has side-effects (alters any value in the current task or chart).

Table 1: Rules for constructing MIR ϵ-closures

Instruction	**Actions**
`CtlBr` B	QUEUE(B)
`CtlFork` B_1, ..., B_n	QUEUE(B_1); ...; QUEUE(B_n)
`CtlMatchChar`	ADD
`CtlMatchClass`	ADD
`CtlMatchSym`	ADD
`CtlStop`	
`StmtCallRuleDyn` CS	ADD
`StmtCallRuleDyn` CS (LR)	$\forall E \in CS$, QUEUE(E)
`StmtReduce` A	ADD; QUEUE($SUCC$)
`StmtRewind` n	ADD

MIR ϵ-closure B can be constructed from closure-seed A with the following algorithm:

1. Add all elements of A to the construction queue Q.
2. For each unique element E in Q, perform the following:
   - If E is an abstract rule index, then all the members of that rule to Q (based on the current grammar).
   - If E is a concrete rule index, then add the index of the first basic block of that rule to Q.
   - If E is a basic block, then add the index of the first instruction of that basic block to Q.
   - If E is an index of an instruction, then execute corresponding actions for that instruction provided in table 1.

Table 1 lists actions to be executed when encountering different instructions in the construction queue:

- QUEUE(A) adds A to construction queue Q (only if it doesn't exist already).
- ADD adds the current instruction index to the resulting ϵ-closure B.

$SUCC$ in rule for `StmtReduce` refers to successor instruction index. It's also important to note that there are two rules for handling `StmtCallRuleDyn` instructions: the first one is used when the call is performed **not** at the beginning of a rule, the second one is used for the calls that appear at the start of the rule. This effectively causes all calls that appear at the start of a rule to be inlined when performing subset construction.

3.3 MIR subset construction

The subset construction is most commonly performed as a result of `StmtCallRuleDyn` instruction. When that happens, the members of provided call specifier are used as a closure-seed. The resulting ϵ-closure then represents instructions that need to be executed at the entry point of optimized rule.

Then member instructions of ϵ-closure are merged:

- `CtlMatchChar` are merged as equivalent `CtlMatchClass`.
- `CtlMatchClass` are merged into a single `CtlMatchClass`. If the resulting `CtlMatchClass` contains overlapping intervals, then target basic blocks of the overlap are merged (by recursively invoking MIR subset construction with the set of target basic blocks as the closure-seed).
- `CtlMatchSym` are merged into a single `CtlMatchSym`, where overlapping match conditions are merged by merging the target basic blocks.
- `StmtCallRuleDyn` are merged by merging their call specifiers.
- Other instructions remain unmerged.

If after merging there is more than one instruction, then they are placed into new basic blocks which then are executed with a `CtlFork` instruction. Otherwise the instruction is appended to the end of the current basic block.

```
rule A() {
  parse ("a", AA);
}

rule AA() { ... }

rule B() {
  parse ("a", BB);
}

rule BB() { ... }

rule main() {
  parse (
    A | B | "c"
  );
}
```

(a) input grammar

```
main: {
  #0: CtlFork #2, #6
  #1: CtlStop
  #2: CtlMatchClass 'a'..'a' => #3,
                    'c'..'c' => #7,
                    else => #1
  #3: StmtCallRuleDyn <M(:AA), 0>,
                      <M(:BB), 0>
      CtlMatchSym <M(:AA), 0..255> => #4,
                  <M(:BB), 0..255> => #5
  #4: StmtReduce R(:A)
      CtlStop
  #5: StmtReduce R(:B)
      CtlStop
  #6: CtlMatchSym <M(:A), 0..255> => #7,
                  <M(:B), 0..255> => #7
  #7: StmtReduce R(:main)
      CtlStop
}
```

(b) optimized MIR for rule `main`

Fig. 6: Subset construction example

MIR subset construction process then continues recursively when `CtlMatchChar`, `CtlMatchClass` or `CtlMatchSym` are encountered (also when `StmtCallRuleDyn` is encountered at the beginning of a rule). To prevent infinite recursion, ϵ-closures and the resulting entry points of those closures are cached.

An example input grammar and its optimized MIR are shown in fig. 6. `A` and `B` rules are inlined into the resulting rule for `main`. The terminal prefixes of `A`, `B` and `main` are merged into a single `CtlMatchClass` instruction (`#2`). Because `A` and `B` have a shared prefix (`a`), the calls and matches to `AA` and `BB` are merged as well (`#3`).

3.4 Parsing ambiguities

SEVM, just like the original Earley parser, traverses all available parse paths. Figure 7 shows two ambiguous grammar rules and their optimized MIRs.

In (G)LR family of parsers there are two main types of conflicts arising from grammar ambiguities: SHIFT/REDUCE and REDUCE/REDUCE conflicts.

Rule `shift_reduce` simulates the scenario of SHIFT/REDUCE conflict: rule `shift_reduce` defines a sequence of `a` characters. However it is not clear when such sequence should terminate. Because of this the rule after parsing every instance of character `a` will perform a reduction of non-terminal `shift_reduce` and then attempt to continue at basic block 0. As a result, a reduction for each possible length of the sequence will be performed.

```
rule shift_reduce() {
  parse "a"+;
}


rule a1() {
  parse "a";
}

rule a2() {
  parse "a";
}

rule reduce_reduce() {
  parse a1 | a2;
}
```

(a) input grammar

```
shift_reduce: {
   #0: CtlMatchChar 'a' => #2, #1
   #1: CtlStop
   #2: StmtReduce R(:shift_reduce)
       CtlBr #0
}
reduce_reduce: {
   #3: CtlFork #4, #5
   #4: CtlMatchSym
         <M(:a1), 0..255> => #8,
         <M(:a2), 0..255> => #8
   #5: CtlMatchChar 'a' => #7, #6
   #6: CtlStop
   #7: StmtReduce R(:a1)
       StmtReduce R(:a2)
       CtlStop
   #8: StmtReduce R(:reduce_reduce)
       CtlStop
}
```

(b) Optimized MIR of input grammar

Fig. 7: Ambiguous grammar example

Rule `reduce_reduce` simulates a REDUCE/REDUCE conflict. In this case, all 3 rules `a1`, `a2` and `reduce_reduce` are merged into a single optimized MIR rule. After successfully matching character `a` (basic block 5), two reductions are performed: one for `a1` and another for `a2` (basic block 7). After performing the reduction for `a1`, this task is awakened at basic block 8. After performing the reduction for `a2`, no new tasks are awakened, because it is detected, that the reduction for `a2` is ambiguous (matches a previous reduction for `a1`).

It is important to reiterate that no parse path under normal conditions is ever traversed multiple times: this is crucial to achieve acceptable performance for using SEVM in practise. Duplicate parse paths are rejected by inspecting $reductions$ while executing `CtlReduce`, thus preventing waking the same task twice and by inspecting $susp\_task$ while executing `StmtCallDyn` to ensure that duplicate tasks for the same non-terminal are not created in the first place.

In case of C programming language, statements like `a * b;` are parsed ambiguously both as declarations and expressions. As a result, a parse forest is produced with an ambiguous node indicating both possible parse paths. It is then left up to the user of SEVM to prune the SPPF manually based on semantic constraints and construct non-ambiguous AST for further processing.

4 Evaluation

4.1 Method

In order to prove that SEVM may be used to parse real-world programming languages, a SEVM implementation called `north` was created. In addition to what is described in this paper, `north` also contains additional features and optimizations:

- Garbage collection. Chart entries are believed to be no longer necessary are discarded, reducing memory usage and making the implementation more cache-friendly.
- Partial reduction incorporation, which is a more limited variation of (Scott and Johnstone, 2005). Some reductions are resolved statically, making it no longer necessary to traverse $susp\_tasks$ in order to resume suspended tasks.
- Token-level disambiguation. Keywords, identifiers and different operators are disambiguated at character-level without requiring reject reductions used in SGLR parsers (Brand et al., 2002).
- Just-in-time compilation. Grammar MIRs are translated into machine code during parser execution with the help of LLVM library (ORC JIT).

Then, ANSI C and Rust grammars for `north` were implemented:

- ANSI C is a widely used language both in practise and in parser implementation comparisons. The ANSI C grammar for `north` does not disambiguate identifiers and type names. As a result statements like `a * b;` are parsed ambiguously both as declarations and expressions.
- Rust was selected as second test language, because its grammar is significantly larger than ANSI C and it contains less ambiguities.

The following parser implementations were selected for comparison:

- `north`. It's the scannerless parsing method described in this paper.
- `flex` + `bison`. `bison` is one of the most commonly used LALR(1) parser generators. Because `bison` is a non-scannerless parser, a `flex` lexer generator was used in conjunction.
- `flex` + `yaep`. `yaep` is Yet Another Earley Parser, which is one of the very few Earley parser implementations available. It is also a token-based parser so `flex` lexer was used in conjunction.
- `dparser`. It's a scannerless implementation of GLR parser (Tomita, 1985).
- `syn`. It's a library/parser designed for parsing Rust code. It uses it's own internal lexer.

The following input files were used for comparison:

- `ansic_470k.c`. This file was taken from `yaep` test suite. It's a 14.8 MB file that contains ≈475000 lines of preprocessed C code. The file was created by combining the source code of entire `gcc` 4.0 compiler into one file, preprocessing it, and removing any non-ANSI C constructs from it (such as `gcc` extensions).
- `rust_650k.rs`. This file was obtained by concatenating all files from `rustc` compiler repository (excluding the test suite) and performing minor adjustments to it, so the resulting file is a syntactically valid Rust program. The file is 22.3 MB in size and contains ≈650000 lines of code, including whitespace and comments.

4.2 Test environment

The test results described in this chapter were obtained on machine with the following specifications:

- **Processor**: Intel i7-3930k.
- **Memory**: 16 GB of DDR3 RAM, 1333 MHz.
- **Operating system**: Ubuntu 18.04.1 LTS.
- **Linux kernel**: 4.15.0-36.
- **GCC**: version 7.3.0.
- **rustc**: version 1.30.0-nightly (90d36fb59 2018-09-13).
- **flex**: version 2.6.4.
- **bison**: version 3.0.4.
- **dparser**: version 1.30.
- **yaep**: obtained from GitHub with revision 1f19d4f5 (WEB, b).

4.3 Test results

Table 2: Table showing the median times it takes to parse sample inputs

Parser	Language	N	IQR	% Outliers	Median time (s)
`bison`	ANSI C	10	0.0008	20.0	0.4974
`dparser`	ANSI C	10	0.0104	20.0	16.1007
north	ANSI C	10	0.0162	0.0	4.6132
`yaep`	ANSI C	10	0.0737	0.0	1.7231
north	Rust	10	0.0197	0.0	6.3258
`syn`	Rust	10	0.0346	0.0	5.5434

The performance comparison results are shown in table 2.

The fastest ANSI C parser (from the ones tested) is `bison`, but that's not surprising, as this parsing method is quite deterministic and restricted to only LALR(1) grammars (`bison` does support GLR grammars as well, but LALR(1) grammar was used to parse ANSI C). `yaep` is 2nd and is quite a bit slower than `bison`, but it is also more general as it's based on Earley parser. However, it is not scannerless. `north` is 3rd and is almost 9 times slower than `bison`, but it's the fastest both scannerless and generalized parsing method. Finally, GLR-based scannerless `dparser` comes last.

Only two parsers were used to compare Rust parsing performance, but that's because Rust is a fairly new programming language and not many parsers/grammars for parsing Rust code exist. `syn` is a hand-written recursive descent parser that was only marginally faster than `north`.

4.4 Validity

The reduce threats to internal result validity, the following precautions were taken:

- All benchmarks/tests were run in the same environment with same configuration.
- Each test was executed multiple times, to increase consistency of the results.
- Before running each set of tests, the specific test scenario was warmed-up for at least 3 seconds to reduce the influence of hardware/software caching and/or dynamic CPU frequency policy.
- IQR method was used to identify outliers to detect other unwanted and unforeseen performance influences that may have happened during execution of the tests.

As for the external validity, the question can be divided into two parts:

- Will the performance of `north` generalize to other ANSI C and Rust workloads?
- Will the performance of `north` generalize to other programming languages?

The first question is simpler: the obtained test results should reflect the performance of parsing other C programs, because the sample inputs for both ANSI C and Rust should cover the entire grammars of ANSI C and Rust and as such any performance pitfalls would have been detected already.

To answer the second question, an important observation needs to made: the performance of `north` is primarily influence by two factors:

1. The average recursive depth of the grammar.
2. The amount of ambiguities present in the parse input/grammar.

All parsing methods will be less performant with higher grammar rule depths: LR parsers, just like SEVM, will need more reductions to parse more deeply nested grammar rules, recursive descent parsers will require more calls/returns. Furthermore, SEVM allows grammar designers to slightly reduce the depth of grammars with the use of abstract grammar rules.

The more important factor for overall `north` and SEVM performance is the amount of ambiguities present in the input file/grammar. The grammars of programming languages are typically designed to contain no ambiguities. If they do exist, it's because of special circumstances, like in ANSI C: where the input is highly ambiguous if no type information is available during parsing. As such, the ANSI C test for `north` may be considered a practical worst-case scenario in regards to ambiguities. Therefore, the observed performance of `north` should generalize to other programming languages as well that exhibit similar level of ambiguousness to ANSI C and/or Rust.

5 Conclusions

We have presented a new, scannerless, virtual machine based approach called SEVM for parsing context-free grammars, which was heavily inspired by the classic Earley parser. We have described the parsing method and how the input grammars are translated into medium-level intermediate representation (MIR) that is then used for parsing. We have also shown an important optimization for this parsing method that merges shared prefixes of grammar rules, which significantly increases the parsing performance. Finally, we demonstrated a SEVM implementation called `north` and have shown that it may be used to parse ANSI C and Rust programs with reasonable performance.

Acknowledgements

Thanks to Vilnius University, Institute of Data Science and Digital Technologies for financing this research.

References

Aycock, J., Horspool, R. (2002). *Practical Earley Parsing*. Compututer Journal, 620–630.

Brand, M., Scheerder, J., Vinju, J., Visser, E. (2002). *Disambiguation Filters for Scannerless Generalized LR Parsers*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Earley, J. (1970). *An Efficient Context-free Parsing Algorithm*. Commun. ACM, 94–102.

Jim, T., Mandelbaum, Y., Walker, D. (2010). *Semantics and Algorithms for Data-dependent Grammars*. Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2010, Madrid, Spain), POPL '10, ACM, New York, United States.

Rabin, M., Scott, D. (1959). *Finite Automata and Their Decision Problems*. IBM Journal of Research and Development, 114–125.

Scott, E., Johnstone, A. (2005). *Generalized Bottom Up Parsers With Reduced Stack Activity*. Comput. J., 565–587.

Stansifer, P., Wand, M. (2011). *Parsing Reflective Grammars*. Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications (2011, Saarbrucken, Germany), LDTA '11, ACM, New York, United States.

Tomita, M. (1985). *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA.

Šaikūnas, A. (2017). *Parsing with Earley Virtual Machines*. Communication Papers of the 2017 Federated Conference on Computer Science and Information Systems (2017, Prague, Czech Republic).

WEB (a). *Regular Expression Matching: the Virtual Machine Approach* (2009). `https://swtch.com/~rsc/regexp/regexp2.html`

WEB (b). *Yet Another Earley Parser* (2018). `https://github.com/vnmakarov/yaep`

Received October 21, 2018 , revised March 21, 2019, accepted April 9, 2019