Vilnius University
Institute of Data Science and
Digital Technologies
L I T H U A N I A

INFORMATICS ENGINEERING (07 T)

TOOLS AND TECHNIQUES FOR SYNTATIC AND SEMANTIC EXTENSION

Audrius Šaikūnas

October 2018

Technical Report DMSTI-DS-07T-18-15

VU Institute of Data Science and Digital Technologies, 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 Implementation of Scannerless EVM ... 5
1.1 Scannerless EVM ... 5
1.1.1 Flaws of the original EVM ... 5
1.1.2 Overview of the internal structure of SEVM ... 7
1.2 Improving grammar expressiveness ... 9
1.2.1 Abstract grammar rules ... 9
1.2.2 Named precedence groups ... 10
1.2.3 Dominating terminals ... 12
1.3 Ambiguity elimination ... 14
1.3.1 Negative reductions ... 14
1.3.2 Strict execution ordering in SEVM runtime ... 16
1.3.3 Negative matches ... 18
1.3.4 Greedy non-terminal repetition ... 19
1.3.5 Strict execution ordering in SEVM optimizer ... 19
1.3.6 Token level ambiguity elimination ... 21
1.4 Parser optimizations ... 23
1.4.1 Profiling EVM ... 23
1.4.2 Just-in-time grammar compilation ... 24
1.4.3 DFA extraction ... 25
1.4.3.1 Terminal symbol matching shortcomings ... 25
1.4.3.2 Simple DFA extraction ... 27
1.4.3.3 Dominating terminals in extracted DFAs ... 29
1.4.3.4 Greedy tokens ... 29
1.4.4 Partially incorporated reductions ... 31
1.4.4.1 Reduction incorporated parsers ... 31
1.4.4.2 The cost of a reduction in SEVM ... 32
1.4.4.3 Reduction incorporation in SEVM ... 32
1.4.5 Garbage collection ... 35
1.5 Avoiding exponential complexity ... 36
1.6 Parse-tree construction ... 37
2 Evaluation ... 38
2.1 Overview of evaluation process ... 38
2.2 Language selection ... 38
2.3 Implementation selection ... 39
2.4 Comparison method ... 39
2.5 Test environment ... 40
2.6 Test data ... 40
2.7 Test results ... 41
2.7.1 Relative performance comparison ... 41
2.7.2 Performance influence of garbage collector ... 42
2.7.3 Performance influence of incorporated reductions ... 43
2.7.4 Performance influence of recursion type ... 44
2.8 Validity ... 46
2.8.1 Internal validity ... 46
2.8.2 External validity ... 47
2.9 Conclusions ... 48

References 49
Appendix Nr. 1. 51
Appendix Nr. 2. 51

1 Implementation of Scannerless EVM

1.1 Scannerless EVM

1.1.1 Flaws of the original EVM

Each parser implementation has several major characteristics by which these parsing methods can be compared:

- **Recognized grammar class**. Different parsing methods can recognize different classes of input languages. For example, LR(0) parsers can only recognize LR(0) grammars. More generalized methods, such as GLR [Tom85] can recognize wider class of input languages (all context-free languages in the case of GLR). However, even then it is possible that such parsing method may not be able to recognize all programming languages used in practise, as not all programming languages are context-free languages. The size of recognized grammar class determines how many real-world computer languages can be recognized by this parser.

- **Expressiveness of grammar language**. Parser development typically starts with creation of target language grammar. This grammar is written in a specific grammar description language, which is then read by parser or parser generator, which then is responsible for generating and/or configuring the parser so it then can recognize the target language. These grammar description languages often provide additional features beyond just production rules to express the target grammar in a more clear and concise fashion. For example, the `bison` parser generator supports operator precedence declarations, which provide a more clear and compact way to describe operator precedence. The existence of such operator precedence declarations does not allow to parse additional languages, but merely allows to express the already recognizable languages in a more intentional fashion. As a result, the greater expressiveness of the grammar language makes development of new grammars easier.

- **Performance**. The performance of the parsing method and it's implementation is one of the primary factors determining whether or not such parser is suitable for parsing real-world computer languages. Generalized parsing methods have existed for decades, however even today they are not widely used due to their lacklustre performance. The same is even more true for scannerless parsing methods, as not a single scannerless parser is used to parse any high-profile programming language (both `gcc`, `clang` and Lua use hand-written recursive descent parsers [Bur75], MRI Ruby implementation uses LALR(1) `bison`, CPython uses a custom bottom-up tokenizer and parser combination, etc).

- **Support for scannerless parsing** determines whether on not two grammars can be effortlessly combined. If two grammars can be combined during parser's runtime, then such parser can be used to parse extensible languages. Additionally, scannerless parsers must provide additional features to eliminate character-level ambiguity.

- **Error correction**. Any parser used in practise should be able to provide informative feedback when a parsing error occurs, so the user of such parser may be able to correct the errors in the parser's input. The more descriptive and informative error messages are, the less time the user needs to spend figuring out why the error occurred and how to fix it.

The original EVM and it's prototype implementation have several flaws that need to be rectified before a proper comparison of EVM with other parsing methods can be made:

- The original research prototype for EVM was implemented in Ruby programming language. Because Ruby is interpreted, any parser with written in this language will be orders of magnitude slower due to the overhead of the interpreter. As such a new EVM implementation is needed, if the performance of EVM is to be compared to other parsing methods.

- While EVM was created with scannerless parsing in mind, there is still one key issue that will severely limit the performance of EVM, even if EVM was implemented in a non-interpreted language: during parsing EVM creates a state for each terminal input symbol. This means that to parse input of length n, $n * size\_of(State)$ bytes of memory is needed just to represent parser states. These states will then contain additional dynamically allocating structures, such as the list of suspended tasks, reductions and the trace.

- Because EVM is a scannerless parser, there needs to be a way to disambiguate identifiers from keywords in languages that have such grammar elements. Furthermore, there needs to be a way to disambiguate operators that consist of more than one character (for example, logical operator `&&` in C may be incorrectly interpreted as a pair of `&` operators). While this disambiguation can be performed post-parse by eliminating invalid parse paths in resulting parse forest [vdBSVV02], both ambiguous parsing and invalid parse elimination would incur additional performance costs. As such, simple character-level ambiguities should be resolved as early as possible during parsing to avoid "useless" work that yields invalid parse trees.

- Trace simplification. In current EVM version, EVM records previous parse positions in a set called trace. This trace contains a complete snapshots of fiber states at various positions during parsing. It is important to notice that because of this there is significant overlap of information that is stored *trace* and the list of suspended tasks and the list of reductions in each state.

1.1.2 Overview of the internal structure of SEVM

In this section we provide a description of the internal structure of Scannerless EVM (SEVM). SEVM is a further modification of EVM that attempts to improve the performance of EVM and extend the parser enough for it to be able to recognize real-world computer languages.

SEVM consists of the following primary components:

- **Grammar compiler** translates textual representation of input grammar to medium-level intermediate language (or MIR for short). It also detects any syntax or semantic errors of the input grammar.
- **Optimizer** is responsible for merging grammar rules in MIR form. Optimizer takes a list of grammar rules to be merged in MIR form and produces combined MIR which implements all of the merged grammar rules, but with their prefixes merged.
- **Resolver** is responsible for invoking optimizer and translating the resulting MIR into machine-code.
- **Runtime** is responsible for coordinating the execution of parser.

SEVM consists of the following data structures:

- **MIR tree** is an abstract syntax tree of intermediate language representation. This is intermediate representation of SEVM grammars.
- **Chart** is the primary data structure of parser runtime. It closely corresponds to EVM state list. Chart is a sparse index map from input positions to **chart entries**.
- **Chart entry** stores all the information about parsing progress at a specific input position. Each chart entry contains the following: reduction list *reductions*, list of suspended tasks *suspended*, list of currently active tasks *running*, activity indicator *queued*.
- **Reduction** contains information about a single reduction: *kind*, *reduce_id*, *length*, *tree_id*. Reduction *kind* determines if the current reduction is an *accept* or *reject* reduction. This information is used to implement negative reductions. Reduction index *reduce_id* determines the non-terminal symbol associated with the reduction. Reduction *length* indicates the reduction length in bytes. Finally, *tree_id* stores the index of the resulting parse node.
- **Task** directly corresponds to fiber in the original EVM. Each task is responsible for parsing one or more non-terminal symbols. A task contains at least the following: *state_id*, *origin*, *position*, *tree_id*, *grammar_id*. Semantically a task can be viewed as a function closure in other programming languages. *state_id* determines the current state of the task: this value is used to implement task suspension and resumption. *origin* is the index of the chart entry in which this tasks was initially created. In

other words, it represents the starting position of the non-terminal that this task will parse. *position* indicates the current parsing position. It is an offset from beginning of the parse input. *tree_id* is the node index of the partially constructed parse-tree so far. *grammar_id* stores the active grammar index.

- **Suspended task** represents a task that was suspended and is awaiting for successful completion of child task. Tasks get suspended when calling other tasks/non-terminal symbols and resumed when these children tasks complete successfully with reductions. Each suspended task contains: *task*, *resumes*, *pos_match*, *neg_match*. A *task* is a copy of suspended task data. *resumes* records the occurrences each time this specific tasks is resumed. *pos_match* represents positive match conditions for resuming this task. *neg_match* represents negative match conditions for resuming this task. Both *pos_match* and *neg_match* are referred as match specifiers.

- **Resume** stores information about a single occurrence of task resumption: the index of reduction that woke the task (*reduce_id*), the length of that reduction, and parse-tree node index that was appended to the newly awakened task. This information is used to eliminate some duplicate parse paths that may lead to exponential complexity. See chapter 1.5 for more about eliminating exponential complexity.

- **Match specifier** is a map from *match_id* and precedence interval to *state_id*. When a reduction occurs at a position *pos* with reduction index *reduce_id* and precedence *prec*, then all suspended tasks in chart entry with position *pos*, whose *match_id matches reduce_id* are resumed in state *state_id*. In other words, match specifier stores the conditions when to resume a suspended task (when awaited reduction happens) and what to do when the resumption occurs (move the tasks into provided *state_id*).

- **Reduce index** represents a non-terminal symbol. Each non-abstract rule has a unique reduction index. Reduction indices are used only when performing reductions.

- **Match index** also represents a non-terminal symbol, but these indices are used on caller side. This separation of reduction and match indices allows to dynamically add new grammar rules, as multiple reduction indices can be matched against a single match index.

- **Call specifier** represents a set or grammar rules that are meant to be invoked during parsing. Optimizer uses call specifier and the grammar MIR as inputs to produce optimized MIR in which multiple rules are merged. Internally, call specifier is a sequence of *match_id* and minimum precedence *min_prec* value pairs.

- **Grammar** stores a mapping between reduce and match indices. Each grammar has a unique index.

- **Parse-tree** stores the automatically constructed parse tree during parsing. Chapter 1.6 details how parse trees are encoded.
- **Call stack** is a stack of chart indices that represents call stack of the parser.
- **DFA** is a data structure that encodes a deterministic finite automata, which is used to parse non-ambiguous intervals of input languages.

1.2 Improving grammar expressiveness

In this section we present and justify several extensions to the grammar language of SEVM.

1.2.1 Abstract grammar rules

Abstract grammar rules are new type of grammar rules that have several purposes:

- They provide an alternative way to declare production rules like $Z = A|B|C$.
- They provide an extension point for extending grammars. Original EVM grammar language provided no grammar construct to specify extensions points: EVM only provided low-level infrastructure needed to implement such extension points, but provided no metalanguage at grammar level to specify such extension points.

Abstract rules may be viewed as non-terminals in form $Z = A_1|A_2|...|A_n$, where Z is the name of the abstract rule and A_i are it's members. Abstract rules in `north` language can be declared with keyword `rule_dyn`. Upon declaration, the newly created abstract rule is empty and new members to it can be added by annotating member rules with `part_of` attribute. Additionally, `part_of` attribute may specify the precedence of this rule member. The precedence value is used when the rule member directly and recursively calls itself via abstract rule to determine if this rule should be part of the call.

It is also important to note that a single non-abstract rule may be a member in multiple abstract rules. In other words, a single rule item may have multiple `part_of` attributes.

Fig. 1 shows an example grammar that uses an abstract grammar rule to implement expression hierarchy, which contains $+$ and $*$ operators with appropriate precedence.

Each abstract rule (same as a normal rule) has a unique *match_id* that may be used to construct calls or perform non-terminal matches. However, unlike normal rules, abstract rules have no reduction indices *reduce_id*. Because of this, the resulting parse forest contains no nodes that represent abstract rules.

When a rule is annotated with `part_of` attribute, a new entry is added to the grammar match map that associates the *match_id* of abstract rule with *reduce_id* and precedence value of target rule.

Compared to traditional notation $A_1|A_2|...|A_n$, usage of abstract syntax rules has a number of advantages:

Figure 1: Abstract grammar rule example

```
rule_dyn expr();

#[part_of(expr, 10)]
rule expr_add() { parse (expr!, "+", expr); }

#[part_of(expr, 20)]
rule expr_mult() { parse (expr!, "*", expr); }

#[part_of(expr, 30)]
rule expr_zero() { parse "0"; }
```

- Increased performance. Abstract grammar rules do not perform reductions and are matched directly against callee *match_ids*.
- Each abstract rule member may have rule precedence. As such, abstract rules provide a simpler way to specify operator hierarchies.

1.2.2 Named precedence groups

Named precedence groups is a grammar feature closely related to abstract rules. Named precedence groups provide a way to call an abstract rule with custom precedence value. Consider the ANSI C grammar fragment provided in fig. 2.

The expressions that have the highest precedence in ANSI C language are *primary expressions*. Below them are *postfix expressions* with slightly reduced precedence. Even lower precedence have *unary expressions* and then *cast expressions*, etc. In expression hierarchies with precedence, rules that represent expressions with lower precedence only refer to expressions with higher precedence. This, however, is not always true: in ANSI C case, `unary_expression` refers to `cast_expression` which has lower precedence. Similar situation can be observed in grouping expression of `primary_expression`, which refers to `expression`, which is the top of the expression hierarchy.

In order to be able to represent such expression hierarchies with abstract syntax rules, there needs to be a way to *name* and invoke a specific level of rule hierarchy. This is what named precedence groups are for. In essence, named precedence groups are callable names attached to specific precedence level (value) of abstract grammar rule.

Named precedence groups may be declared with keyword **`group`**, which is then followed by the group name, the abstract rule name and the precedence level of that abstract rule. If abstract rule represents a set of concrete/normal rules, then named precedence group is a subset of that set.

The grammar fragment fig. 2 may be rewritten in `north` as 3. In `north`, ANSI C `expression` is an abstract grammar rule. Different precedence levels are just named precedence groups (`primary_expression`, `postfix_expression`, `unary_expression`, `cast_expression`).

There are several types of calls in `north`:

Figure 2: A simplified fragment of C99 grammar

```
primary_expression
  : IDENTIFIER
  | CONSTANT
  | STRING_LITERAL
  | '(' expression ')'
  ;

postfix_expression
  : primary_expression
  | postfix_expression '(' ')'
  | postfix_expression INC_OP
  | postfix_expression DEC_OP
  ;

unary_expression
  : postfix_expression
  | INC_OP unary_expression
  | DEC_OP unary_expression
  | unary_operator cast_expression
  ;

cast_expression
  : unary_expression
  | '(' type_name ')' cast_expression
  ;
```

Figure 3: A fragment of C99 grammar rewritten in north

```
rule_dyn expression();

group primary_expression: expression(100) {
  rule identifier_expression() { parse IDENTIFIER; }
  rule constant_expression() { parse CONSTANT; }
  rule string_literal_expression() { parse STRING_LITERAL; }
  rule grouping_expression() { parse ("(", expression!0, ")"); }
}

group postfix_expression: expression(90) {
  rule call_expression() { parse (expression!, '(', ')'); }
  rule inc_expression() { parse (expression!, INC_OP); }
  rule dec_expression() { parse (expression!, DEC_OP); }
}

group unary_expression: expression(80) {
  rule unary_inc_expression() { parse (INC_OP, expression!); }
  rule unary_dec_expression() { parse (DEC_OP, expression!,); }
  rule unary_op_expression() { parse (unary_operator, cast_expression); }
}

group cast_expression: expression(70) {
  rule cast_expression_() { parse ("(", type_name, ")", expression!); }
}
```

Figure 4: A `north` grammar rule for parsing ANSI C multi-line comments (attempt 1)

```
rule comment() {
  parse ("/*", ANY*, "*/");
}
```

- Concrete rule calls. These are in form `unary_operator`, where `unary_operator` refers to a concrete rule.
- Abstract rule calls (non-associative). These are in form `expression`, where `expression` refers to abstract rule. If the call is directly recursive from callee with precedence *prec*, then the same abstract rule with precedence $prec + 1$ is invoked. If the call is non-recursive or transitively recursive, then the abstract rule is invoked with the minimum precedence value of 0.
- Abstract rule calls (associative). These are in form `expression!`. They are statically ensured to be directly recursive. If callee has precedence value *prec*, then abstract rule with same precedence value *prec* is invoked.
- Abstract rule calls with explicit precedence value. These are in form `expression! prec`, where *prec* is an integer value. In such calls callee precedence level (if it exists) is ignored, and an abstract rule with precedence *prec* is invoked.
- Named precedence group calls. These are in form `cast_expression`, where `cast_expression` refers to a named precedence group. In this case, the abstract rule is invoked that is provided in the definition on the referenced named group with the appropriate precedence level.

Because now it is possible to express expression hierarchies using only abstract grammar rules (without having to manually declare concrete grammar rules representing different precedence levels), such hierarchies can be extended by either:

- Adding new rules to existing precedence levels with **`part_of`** attribute.
- Or, by adding entirely new levels (that may exist in-between other precedence levels) with their respective grammar rules.

Because of this, any non-trivial alternative grammar expression $A_1|A_2|...|A_n$ in `north` should be implemented using abstract grammar rules to maximize extensibility of the implemented grammar.

1.2.3 Dominating terminals

Multi-line comments in ANSI C programming language start with characters `/*` and terminate with `*/`. In `north` such comments may be parsed with a rule shown in fig. 4.

Figure 5: A north grammar rule for parsing ANSI C multi-line comments (attempt 2)

```
rule comment() {
  parse ("/*", (r"^*" | ("*", r"^/"))*, "*/");
}
```

Figure 6: A north grammar rule for parsing ANSI C multi-line comments (attempt 3)

```
rule comment() {
  parse ("/*", ANY*, dom_g "*/");
}
```

However such simple rule is not entirely correct: the comment terminator `*/` will be ambiguously matched both as comment terminator and as comment body, forking the rest of the input into two distinct path: one where comment never terminated, and another where `*/` was interpreted as comment terminator.

To avoid this ambiguity, the rule may be redefined as shown in fig. 5. In this case the character sequence `*/` is excluded from comment body by firstly allowing the comment body only to contain non-`*` characters (`r"^*"`), and then requiring that character `*` must not be followed by a slash (`("*", r"^/")`). Such rule correctly and unambiguously parses C comments, however it's nowhere as clear as the initial rule shown in fig. 4.

It would be ideal if there was a way to specify that the slash in the comment terminator `*/` would take precedence over the one possibly found in comment body. This would enable to retain the correct and non-ambiguous semantics of grammar rule of fig. 5 while keeping the simpler definition of fig. 4.

Such precedence or priority in SEVM can be specified using *dominating terminal symbols*. In north grammars user may annotate grammar expressions with **dom_g** specifier, which would cause the last characters or all descendant string grammar sub-expressions to parsed with higher precedence. That way the grammar rule for parsing C comments may be rewritten to the one shown in fig. 6.

The two primary reasons for implementing dominating terminals is that they can be simplify the definition of various grammar rules without any reduction of parsing performance: such behaviour may be implemented statically in SEVM optimizer. The rule shown in fig. 6 may be translated to unoptimized MIR graph shown in fig. 7.

Figure 7: Unoptimized MIR for the rule shown in fig. 6

```
#0 CtlMatchChar '/' => #1
#1 CtlMatchChar '*' => #2
#2 CtlFork #3, #5
#3 CtlMatchClass 0..255 => #4
#4 CtlBr #2
#5 CtlMatchChar '*' => #6
#6 CtlMatchChar '/' => #7, DOM
#7 StmtReduce REDUCE_ID(:comment), NORMAL
   CtlStop
```

Then this MIR would be optimized via subset construction, during which the following ε-closures would be constructed: `[#0]`, `[#1]`, `[#3, #5]`, `[#3, #5, #6]`, `[#3, #5, #7]`.

The most important closure of the set is the `[#3, #5, #7]`, because that's where ambiguity occurs. It is important to note this closure is constructed as the successor of `[#3, #5]` which is reachable via character `/`.

By modifying SEVM optimizer's implementation and annotating instruction `#6` with domination flag `DOM` (that was added as a result of **`dom_g`** specifier), we may request that all the outgoing edges from instruction `#6` should take precedence over all other edges. As a result of this change, the closure `[#3, #5, #7]` now becomes `[#3, #5, DOM #7]`, thus making it possible to simply filter the closure and retain instruction nodes only with highest priority, which after filtering becomes `[DOM #7]`.

Such implementation of dominating terminals is not only simple and effective, but also enables using dominating symbols to disambiguate tokens at character-level as described in chapter 1.3.6.

1.3 Ambiguity elimination

Even though original EVM could parse real-world programming languages, it could not do so without ambiguities. As such, before the resulting parse-tree could be used, it needed to be filtered by disambiguation filters [vdBSVV02], which would remove the parse nodes that represent invalid parse paths. This approach causes two main issues:

- The invalid parse paths still needed to be parsed, thus potentially wasting EVM's performance on invalid parse paths.
- It increases the overall parser complexity, because additional code is needed to perform ambiguity elimination at parse-tree level.

To reduce the impact of both of these issues, some of the ambiguity elimination may be performed *during* parsing. This chapter details several of the techniques used in SEVM to perform such ambiguity elimination.

1.3.1 Negative reductions

Negative reductions are an adaptation of SGLR's reject reductions [EKV09] for SEVM. In Scannerless GLR family of parsers, reject reductions/productions are used to disambiguate reserved keywords from identifiers.

In general, negative reductions work by annotating every reduction in SEVM with *reduction kind*. Reduction kind specifies, if a reduction is *normal* or a *reject* reduction. When a new reduction occurs, as part of exponential parse complexity mitigation, the parser runtime checks if a *matching reduction* happened before. If there already exists a matching reduction then any further reduction processing (such as resuming suspended tasks) is aborted.

Table 1: Reduction kind values

Reduction kind	**Value**
`REJECT`	0
`PREFER`	1
`NORMAL`	2
`AVOID`	3

Figure 8: A grammar rule that defines an identifier followed by a space

```
rule ident() {
  reject ("if", " ", R1);
  parse (r"a-zA-Z"+, " ", R1);
}
```

An existing reduction A and a new reduction B are considered to match if any of the following statements are true:

- They have the same *reduce_id*, *reduce_kind* and *length*:

  $A_{reduce\_id} = B_{reduce\_id} \wedge A_{reduce\_kind} = B_{reduce\_kind} \wedge A_{length} = B_{length}$

- They have the same *reduce_id*, but the new reduction has a higher *reduce_kind*:

  $A_{reduce\_id} = B_{reduce\_id} \wedge A_{reduce\_kind} < B_{reduce\_kind}$

The first condition is used for identical reduction de-duplication. The 2nd condition implements reduction priorities: if there already exists a reduction with higher priority then the new, lower priority reduction is rejected. For this approach to work, all code that implements higher priority reductions must to be executed first, otherwise it is possible for lower reductions to "slip through". If this does indeed occur during parsing, then such an event is called a *reduction slip*. Reduction slips can only happen in ill-formed grammars with recursive negative reduction cycles.

In order to be able to compare reduction kinds, each reduction kind is assigned a unique integer value (see table 1). Then the reduction kinds are compared by these integer values.

From the user's perspective, negative reductions can be defined in grammars with `reject` keyword, which is then followed by a grammar expression. If this grammar expression matches, then all subsequent reductions that happen in the same rule are rejected.

The grammar shown in 8 defines a rule for parsing identifiers, which may be composed from lower case or upper case characters followed by a space. However, if the identifier matches the keyword `if`, then a negative reduction is produced, which prevents any other normal identifier reductions from being added. This effectively disambiguates identifiers from keyword `if` (see fig. 9 for MIR of the same grammar rule). By adding more complex grammar expressions to the `reject` statement, it is possible to

Figure 9: Unoptimized MIR for grammar rule shown in fig. 8

```
#0: CtlFork #1, #5

#1: CtlMatchChar 'i' => #2
#2: CtlMatchChar 'f' => #3
#3: CtlMatchChar ' ' => #4
#4: StmtRewind 1
    StmtReduce REDUCE_ID(:ident), REJECT
    CtlStop

#5: CtlMatchClass 'a'..'z' => #6, 'A'..'Z' => #6
#6: CtlFork #5, #7
#7: CtlMatchChar ' ' => #8
#8: StmtRewind 1
    StmtReduce REDUCE_ID(:ident), NORMAL
    CtlStop
```

disambiguate several keywords or even more complex grammar expressions from identifiers.

1.3.2 Strict execution ordering in SEVM runtime

EVM, much like the original Earley parser [Ear70], performs mostly breadth-first search (with the exception when fiber priorities were involved, which were used primarily to implement regular look-ahead). Other than this, the rest of the execution of the parser was unordered: `i_fork` instruction for creating duplicate fibers would queue the fiber for execution, but in arbitrary order. `i_call` family of instructions also behaves similarly: the newly created tasks are also queued in an unspecified order.

While this arbitrary execution model works well in EVM, it's no longer suitable for SEVM: SEVM has to ensure that the reductions with higher priority execute first to avoid reduction slips. To that end, the entire execution model for SEVM must be shifted to depth-first execution:

- `CtlFork` $B_1, B_2, ..., B_N$ instruction has to ensure, that the basic blocks $B_1, B_2, ..., B_N$ complete in the same order as they are given to the `CtlFork` instruction.
- `StmtCall` family of instructions has to ensure that the callee will begin execution immediately after the current task completes or is suspended with `CtlMatchSym`.

Internally, this is implemented by a two layer stack:

1. Primary call stack stores all chart entry indices that have at least one active task.
2. Each chart entry has a secondary call stack, which ensures proper execution ordering in entries that have more than one active task.

When a new task is created, it is added to the top of appropriate secondary stack. Then the index of that chart entry is added to the top of primary call stack if the index does not already exist in the primary stack.

To avoid having to perform linear search in primary stack to check if an index already exists, each chart entry contains an indicator *queued*, which is set to *true* whenever the corresponding chart entry index is added to the primary call stack.

Then the algorithm for executing SEVM tasks is comprised out of the following steps:

1. Locate the currently active chart entry by retrieving it's chart index from the top of the primary call stack. If the primary stack is empty, then parser terminates.
2. If the secondary stack is empty, then attempt to populate it by *failing* the current entry (see chapter 1.3.3). If failing yields no new tasks, then remove the top element from primary stack index and go to step 1.
3. Pop a task from the secondary stack stored in the current chart entry.
4. Resume the task.
5. Go to step 1.

This algorithm in essence simulates how call stacks work in traditional imperative programming languages, but also adds an ability to execute several tasks "in parallel".

Because of the `north` grammar to MIR translation rules and the above SEVM execution algorithm, the following grammar expressions now have ordered execution:

- Members grammar expressions $E_1, E_2, ..., E_n$ of the alternative grammar expression $E_1|E_2|...|E_n$ now complete in the order in which they are given.
- Call grammar expression *C*, where *C* is a valid call target, now fully completes (all of the possible alternative parse paths are analysed), before resuming the caller. This happens because each call grammar expression is translated into a pair of `StmtCall` and `CtlMatchSym` instructions. The first one queues the callee for immediate execution and the 2nd one causes the current task (caller) to be suspended, effectively yielding execution control to the callee. If the callee creates new subtasks (for example, as a result of `CtlFork` or `StmtCall`), they are placed on the top of the current secondary stack (if the call is left recursive) or on top of another secondary call stack, which causes causes the currently active chart entry to shift.
- Reject statements `reject` *E*, where *E* is another grammar expression now complete before any subsequent statement completes. This is because reject statements fork execution with `CtlFork` into two parse paths: the primary parse path, which contains the code for grammar expression *E* and terminates with `REJECT` reduction, and the secondary parse path, which contains the remainder of the current parse rule. Because of this, it is guaranteed that the `REJECT` reductions will always happen before `NORMAL` reductions, thus fulfilling the strict execution ordering requirement for negative reduction implementation.

1.3.3 Negative matches

The strict execution ordering when applied to rule calls has an additional positive side-effect that may be used to implement negative non-terminal matching: because the caller of a grammar rule is only resumed when all of the callees and their subtasks fully complete, it is possible to determine if a particular non-terminal *failed* to match.

In order to detect such negative matches at MIR level, match specifiers in `CtlMatchSym` instruction are split into two parts: positive and negative match part. Each part lists the conditions for resuming the suspended task. Each condition is *match_id*, *min_prec*, *state_id* tuple: *match_id* indirectly represents a set of accepted reductions and *min_prec* specifies the minimum precedence value of those reductions and finally *state_id* indicates task state index in which the suspended task should be resumed. The positive part of match specifiers is used only when resuming tasks as a result of new reductions. The negative part is used during *chart entry failure*.

In SEVM negative (failed) matches are detected when selecting a task for execution: when the secondary stack of a chart entry is empty and the runtime attempts to pop a task from it, the following conclusions can be made as a result:

- That all active tasks from the current chart entry have been completed (because the secondary stack is empty).
- That the current chart entry is active (it's index is stored at the top of primary call stack).

In other words, the parsing process at the current entry/position has reached a dead-end, because all of the possible parse paths starting at $CE_{position}$ have been explored to their completion, where CE is the current chart entry. At this point during parsing SEVM *fails* the current chart entry by performing the following steps:

1. The newest suspended task T from the current chart entry CE is selected.
2. If the suspended task has at least one negative match (it's negative match specifier is not empty), then go to next step. Otherwise discard the current suspended task, because all of it's subtasks have failed and then go to step 1.
3. The last entry MS of negative match specifier of task T is selected.
4. MS is matched against the list of all reductions of CE. If there is at least one positive match, that indicates that the suspended task T was resumed at least once and negative match cannot be performed. As a result, MS is removed from the negative match specifier of T. Then continue to step 2, otherwise proceed to the next step.
5. If no positive match for MS was found that means that the specific *match_id* with minimum precedence *min_prec* failed to match at position $CE_{position}$. As a result, task T is resumed in state *state_id* by pushing a copy of T to secondary stack of E. Further chart entry failure is aborted.

In essence, during *chart entry failure*, each suspended task from newest to oldest is failed in turn: each suspended task is either discarded if no negative matches have been detected, or resumed otherwise. The process is continued until at least one task is resumed or all the list of suspended tasks in the current chart entry becomes empty.

It is important to note, that because of the negative matches, the order of suspended tasks must be preserved in order for recursive negative matches to work correctly. Also, the order of resume conditions in negative match specifiers is also important.

Negative matches in SEVM/`north` aren't directly accessible to user, but they are used to implement greedy non-terminal repetition operators.

1.3.4 Greedy non-terminal repetition

Greedy repetition in SEVM is accessible via `parse_g` statements, which are similar to regular `parse` statements, but certain operations within provided grammar expression are replaced with greedy equivalents.

Greedy non-terminal repetition is implemented by using negative matches: call rule grammar expression *R*, where *R* is a valid call target is normally compiled as a pair of `CallRuleDyn` and `CtlMatchSym` instructions. However, if the *R* grammar expression is a descendant of `parse_g` and child of one of the repetition operators (`?`, `*` or `+`) then the call is compiled differently: `CtlMatchSym` instruction now contains the the callees *match_id* both in positive and negative parts of match specifier. This means that the statement `parse_g (A*, B)` fully completes parsing the sequence of *A* non-terminals and only when parsing *A* fails the control is transferred to parse *B*, effectively enabling to parse greedy sequences of non-terminals.

This, however, has an undesirable side effect: because *A* and *B* are parsed separately, their prefixes cannot be merged. This may potentially lower the performance of SEVM and thus greedy non-terminal repetitions should be used sparingly to avoid interfering with optimizer's subset construction.

1.3.5 Strict execution ordering in SEVM optimizer

So far we described how the runtime `north` preserves the strict execution order that is required to implement negative reductions and negative matches. However, ensuring proper execution ordering just in runtime is not enough: SEVM relies heavily on it's optimizer, which can merge multiple grammar rules by performing a variation of subset construction on MIR graphs (the algorithm for which is inspired by Efficient Earley Parser [JM10]). In this chapter the description is given how the strict execution ordering is preserved during optimizer's subset construction.

This is a simplified version of subset construction algorithm used by original EVM:

1. Add the instruction pointers to be merged into a initial set S_I.

2. Add this set into resolution queue Q.

Table 2: Rules for computing SEVM ε-closures

Instruction	**Action**
`CtlBr` *target*	`VISIT` *target*
`CtlFork` $B_1, B_2, ..., B_N$	`VISIT` B_1 `VISIT` B_2 ... `VISIT` B_N
`CtlMatchChar` ...	`RELEVANT`
`CtlMatchClass` ...	`RELEVANT`
`CtlStop`	`IGNORE`
`StmtCallRuleDyn` *T*, *min_prec*	`VISIT` *T*, if the call is at origin `RELEVANT`, otherwise `VISIT` *next*
`StmtReduce` *reduce_id*, *kind*	`RELEVANT` `VISIT` *next*
`StmtRewind` *num*	`RELEVANT`

3. Remove one set S_0 from the Q.

4. Find ε-closure of the set S_0 and store it as set S_1.

5. Go back to step 2 if S_1 was merged already by looking up it's entry in subset construction cache C.

6. Store the mapping S_1 to ip_{end} into subset construction cache C, where ip_{end} refers to the end of the grammar program; this is where the merge result of S_1 will be stored.

7. Merge the instructions of the set S_1 and write result to ip_{end}. This step may queue additional elements to Q.

8. Continue until Q is empty.

Much like the original subset construction for converting NFAs to DFAs [RS59], the one used for EVM uses sets to represent instruction ε-closures and a queue to control the order of individual subset construction steps.

Because SEVM has strict execution ordering, sets no longer suitably represent SEVM ε-closures. Instead, ε-closure in SEVM is a sequence of unique MIR node indices. ε-closures in SEVM optimizer are constructed recursively, essentially by simulating function call behaviour of imperative programming languages. Because of this, it is possible to have several distinct ε-closures with same elements, but with different orderings of those elements.

Rules for constructing ε-closures in SEVM are given in table 2. Whenever one of the given instructions is encountered, the appropriate actions are executed:

- `VISIT` E recursively visits the entity E:
  - If E is an instruction, then it's visited according to the rules provided in table 2.

Figure 10: A grammar for parsing identifiers and keywords

```
rule ident() { parse (r"a-z"+, " ", R1); }
rule kw_self() { parse ("self", " ", R1); }
```

Figure 11: A modified grammar for parsing identifiers and keywords

```
rule ident() { parse (r"a-z"+, " ", R1); }
rule kw_self() { parse ("self", dom_g " ", R1); }
```

  - If E is a basic block, then the first instruction of that basic block is visited.
  - If E is a concrete rule, then the first basic block of that rule is visited.
  - If E is an abstract rule, then all of it's implementations are visited.

- `IGNORE` ε-closure construction.
- `RELEVANT` I adds instruction I to the resulting ε-closure.

Once an ε-closure is obtained, it's instructions are merged much like in original EVM. One key difference in SEVM subset construction is that the initial merge sequence may only contain other concrete rules. In other words, during SEVM subset construction, one or more concrete rules are merged into a new rule, which remains entirely separate from the rules constructed in previous iterations. As a result, the constructed and optimized rules are entirely independent and isolated from any other code.

The primary advantage of this is that the calls that start at rule origin may be partially incorporated, thus increasing the reduction performance.

The main disadvantage is that this results in a significantly higher amount of code generated: in original EVM generated rule suffixes were reused possibly several times across entire grammar. In SEVM, the reuse may only happen internally within one generated rule. In other words, if optimizer constructs merged rule for parsing `A | B` and later for `A | C`, then no code between these two generated rules will be shared, whereas EVM may reuse some part of `A | B`, which represents a unique suffix of `A` in `A | C`. To combat this duplication of code, matching state transition tables in deterministic DFAs are cached and de-duplicated, as described in chapter 1.4.3.2.

1.3.6 Token level ambiguity elimination

An unexpected side-effect of dominating terminals implementation is that dominating terminals can have an effect beyond just a single rule in which they are used: because the optimizer may potentially merge multiple rules into one combined rule, a terminal from one rule may dominate over nodes found in the other rules. Because of this, dominating terminals may be used to disambiguate identifiers from keywords without using more computationally expensive negative reductions.

Figure 12: A grammar that uses token groups to disambiguate keywords from identifiers

```
rule_dyn kw();

#[token_group]
group _: kw(0) {
  rule ident() { parse (r"a-z"+, " ", R1); }
  rule kw_if() { parse ("self", dom_g " ", R1); }
  rule kw_self() { parse ("self", dom_g " ", R1); }
}

rule expr_if() {
  parse (kw_if, ...);
}
```

Table 3: Identifier-keyword disambiguation performance cost comparison

Approach	Resulting symbol	Total reduction count
Negative reductions	Identifier	1
Negative reductions	Keyword	3
Token groups	Identifier	1
Token groups	Keyword	1

Consider the grammar shown in fig. 10. It defines two grammar rules: one for parsing identifiers and another for parsing a reserved keyword `self`, both of which must be followed by a space. If these non-terminals are used in a grammar expression like `ident | kw_self`, then the result would be ambiguous, because both rules would match. To resolve this ambiguity, negative reductions can be used.

Alternatively, the grammar may be modified as shown in fig. 11. In this case, the terminating whitespace symbol (in practise an alphanumerical boundary symbol is typically used instead) is changed to be dominating with **`dom_g`** specifier. As a result, when `ident | kw_self` expression is encountered, `ident` and `kw_self` rules are merged. Subset construction continues until the terminating symbol is encountered, at which point the lower priority (non-dominating) terminating symbol for `ident` is filtered-out, allowing only `kw_self` reduction to occur, thus eliminating the ambiguity.

This scenario, however, only works when it is guaranteed that `ident` is merged with all other keywords (in this example `kw_self`). Under normal circumstances no such guarantee can be made, however, SEVM can be extended to enforce this condition.

For this reason, `#[token_group]` attribute is introduced to `north`, which can be used to annotate named precedence groups. When a call is made to a rule that is part of a `#[token_group]` group, then the call to that rule is replaced with a call to the whole group, without changing the way `CtlMatchSym` instruction is generated. This means that the members of a token group are always guaranteed to be merged during subset construction. As a result, combining token groups with dominating terminals allows to effectively disambiguate keywords from identifiers.

Comparing this approach to negative reductions reveals significant performance gains for parsing reserved keywords. Table 3 shows performance cost (in terms of total reductions needed) to recognize disambiguated keywords and identifiers with both of the described approaches.

Disambiguating keywords from identifiers with negative reductions requires 3 separate reductions to be performed:

1. Keyword reduction (`kw_if`, `kw_self`, etc).
2. Negative identifier reduction that is performed as a result of matching keyword within `reject` statement. This reduction may be avoided by manually listing all keywords within identifier definition, but such approach is impractical and unergonomic.
3. Positive identifier reduction that gets eventually rejected.

When using token groups (in combination with dominating terminals), only 1 reduction is needed. Another positive effect of token groups is that it results in less significantly less generated code, because of the following reasons:

- No separate parse path for matching keywords and performing negative reduction is needed.
- Token group disambiguation can happen as part of DFA extraction process (see chapter 1.4.3.2 for more), which reuses matching transition tables across different rules.
- Replacing all direct calls to individual keyword rules into corresponding token groups results in lower number of unique call specifiers, which means that less optimized rules need to be generated and translated to machine code in total (but the ones that include any keyword become larger, because instead of parsing a single keyword, these rules will be capable of recognizing every keyword defined in a token group).

1.4 Parser optimizations

1.4.1 Profiling EVM

During research and development of SEVM/`north`, the following profiling methods were used:

- Built-in performance counters. During various steps of `north` execution, execution times for most important components are measured and stored. This information is then optionally displayed after the execution to the user.
- `callgrind` code profiler. This is a tool designed to profile program performance. It works by instrumenting input programs and keeping detailed logs of their execution. As a result, the input program is executed significantly slower, but the

additional instrumentation allows to obtain detailed metrics about the entire process of program execution.

- `massif` heap profiler. This is a tool that allows to measure and observe the changes of overall memory usage.

- `bench_parsers` tool. It was developed as part of the `north` implementation and allows to compare the performance of different parser implementations with great accuracy.

Built-in performance counters were used to quickly measure and detect changes in performance as a result of `north` implementation/configuration or input grammar adjustments.

`callgrind` was used to identify the critical paths of `north` execution. It allows to observe how many times each function is called, how long each call on average takes and similar. This tool enabled to identify the parts of `north` that were running the slowest and thus focus optimization attempts at such locations, either by optimizing such functions, or by adjusting the parsing method to reduce the number of calls to such functions.

`massif` was used to identify the parts of code that allocate the most memory. As a result of `massif`'s measurements, the garbage collector for SEVM was implemented to significantly reduce memory usage of `north`.

1.4.2 Just-in-time grammar compilation

To minimize the overhead of interpreting EVM's instructions, in `north` a just-in-time compiler is used to translate optimized rule MIRs into native machine code that can be directly executed by the processor. The machine code in `north` is generated by LLVM library: at first SEVM's MIR is translated into LLVM's IR (intermediate representation), which then is translated by LLVM into machine code.

Some changes have been made to SEVM to simplify translation of MIR to LLVM IR:

- MIR instructions are organized into basic blocks: each basic block contains 0-or-more statement instructions and must terminate exactly by 1 control instruction. All operations that affect the flow of the execution are control instructions. This approach somewhat mimics LLVM design, where instructions are also organized in basic blocks.

- MIR rules are composed out of basic blocks, instead of instructions. This matches LLVM functions, which are composed out of LLVM basic blocks.

Each task is compiled into a single native function, which takes the parser's context and a pointer to the current task as parameters. This function is referred as the `resume` or task resumption function.

`resume` function of a rule always starts with a preamble, which loads commonly used values into temporaries to reduce code duplication and terminates with a **`switch`** statement, which transfers execution to the appropriate state based on task's *state_id* value. *state_id* values correspond to matching MIR basic block indices to ease debugging process. Each SEVM basic block is translated by translating individual instructions of that basic block directly into LLVM IR. Some MIR instructions can be translated into several LLVM IR instructions or even several LLVM IR basic blocks.

Most of statement instructions (such as `StmtCallRuleDyn`, `StmtReduce`, `StmtRewind`) are compiled into LLVM IR function calls (`call`), which invoke `north` runtime. The context of the parser runtime and a pointer to the current task as well as instruction-specific operands are passed as arguments to those functions.

`CtlMatchChar` instructions are compiled into several LLVM instructions:

1. `load`: Firstly, the current input position pointer is loaded from the current task.
2. `icmp`, `br`: Current input position pointer is checked against end of input pointer, a conditional jump is made as a result.
3. `load`: Input character at current position is loaded.
4. `getelementptr`, `store`: The current position pointer is increased by 1 and written into the current task.
5. `icmp`, `br`: Input character is compared with target character and a conditional jump is made as a result.

`CtlMatchClass` instruction is compiled similarly: the first 4 steps are the same as `CtlMatchChar`, but the input character is compared using unrolled binary search: each bound of search space is compared with a pair of `icmp` and `br` instructions.

`CtlReduce` instruction is translated into a call to appropriate runtime function and unconditional jump to target location.

`CtlMatchSym` is translated into a `call` instruction to appropriate runtime function, which takes ownership of the current task and (potentially) adds it to the list of suspended tasks, and `ret` instruction, which stops the current task.

`CtlStop` is translated into a single `ret`, which terminates the current task.

1.4.3 DFA extraction

1.4.3.1 Terminal symbol matching shortcomings

In current version of SEVM terminals are matched with `CtlMatchChar` and `CtlMatchClass` instructions. `CtlMatchChar` can match a single input character against another character, where `CtlMatchClass` can match a single input character against several different symbols. By analogy, `CtlMatchChar` can be viewed as an imperative `if` statement, whereas `CtlMatchClass` would be a **`switch`**.

Both of these instructions get replaced with `CtlMatchClass` during subset construction, which later gets translated into LLVM IR. The compiled `CtlMatchClass` performs binary search to match the input character against several possible alternatives. As a result, the resulting LLVM IR code contains at least:

- 3 basic blocks.
- 3 comparison instructions: 1 to test for end-of-stream, 1 to test the lower bound, 1 to test the upper bound.
- 3 conditional jumps.
- 1 addition: used to increase the position value of current task.
- 2 memory loads: used to load current position and the character at the current position.
- 1 memory store: used to store the updated position of the current task.

This means that matching a single input character, when translating SEVM MIR to LLVM IR requires significant amount of instructions and basic blocks. The Rust language grammar for `north`, as of the time of writing this, contains 41 distinct keywords and 48 operators. On average, each keyword contains 4.3 and each operator 1.5 characters. All keywords and operators when concatenated occupy 250 characters. Some of these characters would be merged during subset construction. However, even if all keywords and operators required 125 distinct `CtlMatchClass` instructions, they would occupy at least 375 LLVM IR basic blocks. This does not include other token-like non-terminals, such as comments, literals and whitespace.

The problem is further compounded due to the way subset construction works: only complete rules are merged to form another complete optimized rule. Because of this, each distinct operator precedence level would be optimized at least once, each time including every keyword of the grammar, resulting in massive amounts of generated code.

Another yet unsolved issue in SEVM is extensible way for disambiguating operators. Keywords from identifiers can now be effectively disambiguated with token groups and dominating terminals, but this method only enables to disambiguate tokens of same length. As a result, additional method is needed to disambiguate operator `&&` from a pair of `&`. For example expression `a && b` in C language without any disambiguation can be interpreted both as `a && b` (logical and) and as `a & (&b)` (bitwise and where right hand side is the address of variable `b`).

A simple solution to his problem would be to add negative lookahead to operator `&`, so it may not be followed by another `&`. This can effectively be implemented in SEVM with rewind directive `R1`, but such approach requires for the grammar author to know all the possible operators before hand, thus making extensions to the language more limited. Another issue is that such operator definition breaks rule encapsulation, because the rule for parsing operator `&` has to contain knowledge about operator `&&`.

Figure 13: A `north` grammar for matching keywords

```
rule_dyn kw();

group _: kw(0) {
  rule kw_self()   { parse ("self", " ", R1); }
  rule kw_static() { parse ("static", " ", R1); }
  rule kw_struct() { parse ("struct", " ", R1); }
}
```

Figure 14: Optimized MIR for rule `kw` shown in fig. 13

```
 #0: CtlMatchClass 's' => #1
 #1: CtlMatchClass 'e' => #2, 't' => #7
 #2: CtlMatchClass 'l' => #3
 #3: CtlMatchClass 'f' => #4
 #4: CtlMatchClass ' ' => #5
 #5: StmtRedind 1
     StmtReduce REDUCE_ID(:kw_self), NORMAL
     CtlStop
 #7: CtlMatchClass 'a' => #8, 'r' => #14
 #8: CtlMatchClass 't' => #9
 #9: CtlMatchClass 'i' => #10
#10: CtlMatchClass 'c' => #11
#11: CtlMatchClass ' ' => #12
#12: StmtRedind 1
     StmtReduce REDUCE_ID(:kw_static), NORMAL
     CtlStop
#14: CtlMatchClass 'u' => #15
#15: CtlMatchClass 'c' => #16
#16: CtlMatchClass 't' => #17
#17: CtlMatchClass ' ' => #18
#18: StmtRedind 1
     StmtReduce REDUCE_ID(:kw_struct), NORMAL
     CtlStop
```

All of these issues described in this chapter can be solved (to an extent) by comparing the current method for matching terminals in SEVM with traditional lexers: during subset construction, SEVM optimizer essentially constructs an embedded lexer each time a terminal symbol (or terminal symbol sequence) is to be matched. By isolating these deterministic fragments of instructions sequences, it would be possible to extract them and to perform terminal symbol matching in a lexer-like environment, isolated from the rest of VM. We call this approach of separating terminal matching as *deterministic finite automata extraction* or *DFA extraction* for short.

1.4.3.2 Simple DFA extraction

Consider the grammar shown in fig. 13. It defines 3 keywords: `self`, `static` and `struct`. During subset construction shared prefixes of these keywords will be merged and MIR shown in fig. 14 will be produced. This MIR may also be visualised as a deterministic finite automaton as shown in fig. 17, which captures the essence of DFA extraction method: the segments of deterministic source MIR get extracted into a separate DFA, which is then

Figure 15: Traditional DFA for MIR shown in fig. 14

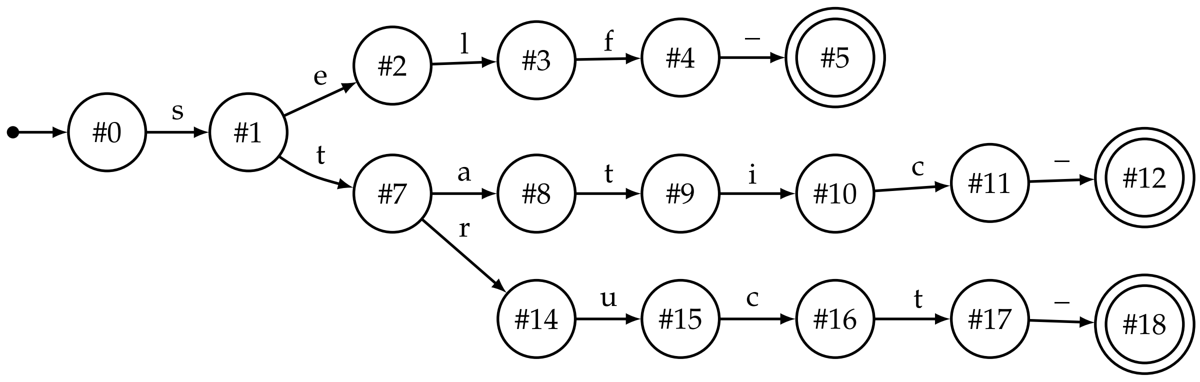


Figure 16: Optimized MIR for rule `kw` shown in fig. 13 (with DFA extraction enabled)

```
#0: CtlExecDFA <DFA:0>, 0 => #1, 1 => #2, 2 => #3
#1: StmtRedind 1
    StmtReduce REDUCE_ID(:kw_self), NORMAL
    CtlStop
#2: StmtRedind 1
    StmtReduce REDUCE_ID(:kw_static), NORMAL
    CtlStop
#3: StmtRedind 1
    StmtReduce REDUCE_ID(:kw_struct), NORMAL
    CtlStop
```

used for matching terminal symbols. Then, instead of `CtlMatchClass` (and `CtlMatchChar`) instructions, the resulting MIR contains a new `CtlExecDFA` instruction, which executes the DFA and transfers the control based on success or failure of DFA match result.

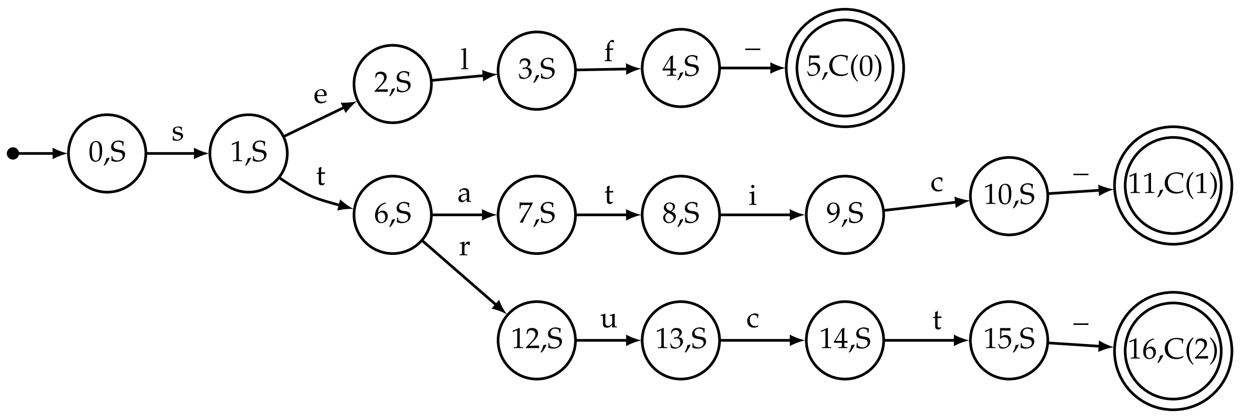
Optimized MIR for abstract rule `kw` with DFA extraction enabled is shown in fig. 16. `CtlExecDFA` instruction takes 2 operands: the DFA to be executed and transition table that pairs the result of DFA with the target *state_id* of the task. Note the significant reduction of basic blocks in this version of optimized MIR.

Every `CtlExecDFA` instruction is translated to 2 LLVM IR instructions: single call to `north` runtime, which simulates the DFA and returns the result; and a switch statement, which transfers the control of execution based on DFA simulation result.

It is also important to note, that states in SEVM DFA are classified by their type:

- *Shift* ("S") states only consume a single input symbol and move to a different state. Each shift state contains a transition table.
- *Fail* ("F") states fail the DFA simulation immediately upon entering them. Typically each DFA contains exactly 1 fail state, which is reachable from all other states with unexpected terminals. They are not shown in any of DFA visualisations, because there would be an edge from each *shift* state to the fail state with all other characters from ASCII range 0..255.
- *Complete* ("C") states terminate the DFA simulation with given result. The result is a

Figure 17: SEVM DFA for MIR shown in fig. 13



number that then is used in MIR to transfer control of execution.

- *Lookahead* ("L") states are used to implement lookahead. See chapter 1.4.3.4 for more.

Furthermore, shift state transition tables are split into two parts: transition index table and transition state table: transition index table store indices of transition state table, which store actual destination state indices. This two layer transition-table approach allows to de-duplicate and reuse transition index tables. All transition index tables have 256 entries (1 byte each), where 1 entry is reserved for each possible input character. Transition state table is variable sized and it's size corresponds to the number of unique transition destinations from a specific state.

This significantly reduces the size of generated DFAs, because the largest parts of each DFA can be reused: the largest DFAs used to parse Rust language is composed out of 237 distinct states, 136 of which are shift states, ≈95% of which are reused in at least one other DFA.

1.4.3.3 Dominating terminals in extracted DFAs

Dominating terminals in extracted DFAs work just like with original `CtlMatchClass` instructions, because the `north` optimizer uses the same ε-closure computation algorithm for both subset construction and DFA extraction. As a result, token group based approach for identifier-keyword disambiguation works with DFA extraction without any additional modifications.

1.4.3.4 Greedy tokens

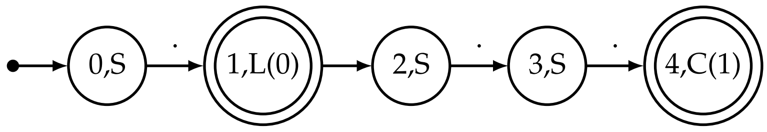
Extracting the terminal matching algorithm from SEVM VM has one additional benefit: it allows us to implement greedy token matching: this would enable to disambiguate operator `&&` from a pair of `&`s, a pair of divisions `/` from one-line comment start and similar.

They way the extracted DFA works already resembles traditional lexers. By extending this analogy can farther we can implement *greedy tokens*: in case where there are

Figure 18: A north grammar for parsing `.` and `...` operators

```
rule op_dot() { parse shift_p "."; }
rule op_dot_dot_dot() { parse shift_p "..."; }
rule main() { parse op_dot | op_dot_dot_dot; }
```

Figure 19: SEVM DFA for grammar shown in fig. 1.4.3.4



multiple token matches available (such as `&` and `&&`), select the longest.

In SEVM this can be done by adding an additional indicator to `CtlMatchChar` and `CtlMatchClass` instructions to specify the longest match preference. At the grammar level, **`shift_p`** (*prefer shift*) directive is needed to express the desire to traverse only the longest match when multiple character-level parse paths are available.

Normally, the divergence of two parser paths is detected immediately after constructing ε-closure when building DFAs: if all members of constructed ε-closure are `CtlMatchChar` or `CtlMatchClass` instructions, then they are merged into a single shift state in the DFA. If there are additional instructions (such as `StmtCallRuleDyn`, `StmtReduce` or `CtlReduce`), then a complete state is generated instead, which hands the execution control back to SEVM, which then will fork the execution (typically) into two different to paths: one task with another DFA that performs character matching, and another that contains other instructions.

To implement longest input match in SEVM DFA, the *completion* states can be replaced with *lookahead* states: each lookahead state will recursively start another DFA at a given state: if child DFA completes successfully, then it means that a longer match has been found and the result of that DFA is returned from primary DFA. However, if the child DFA fails, it means that matching an alternative parse path with potentially longer input was unsuccessful, and the original completion value is returned instead.

An example grammar for parsing and disambiguating operators `.` and `...` is shown in fig. . During subset construction, terminal symbol matching will be extracted into DFA shown in fig. 19. After matching a single dot character (`.`), lookahead state 1 will be reached, which will spawn a child DFA that will start in state 2. If two additional dots are found, then the child DFA will complete with result 1 in state 4, otherwise it will fail, which will cause the main DFA to complete successfully with result 0.

It's important to note, that there can be several levels of lookahead states, allowing to disambiguate complex tokens. For example, greedy tokens are used in the grammar of Rust programming language to disambiguate *all* of Rust's tokens:

- Raw string literals `r"text"` are disambiguated from identifier `r` and text literal `"text"` sequence.

- Operators of varying length are disambiguated (`.`, `..`, `...`, `..=`, `=`, etc).
- In combination with dominating terminals, base-16 integer literals such as `0 x1234ABCD` are disambiguated from base-10 integers with a suffix (`10i32`).

Because of SEVM greedy tokens, the `north` parser can fully replicate the behaviour of a lexer in a scannerless parser, thus allowing to parse the languages that depend on such behaviours without ambiguities.

1.4.4 Partially incorporated reductions

1.4.4.1 Reduction incorporated parsers

The LR family of parsers [Knu65] use a stack to track the execution of overall parsing process. The stack contains state indices which represent the path through which the current parser position was reached from the initial parsing position. Additional elements to the stack are added with *shift* actions, and multiple stack entries are removed and consolidated into one with a *reduce* action. The top element of the stack always represents the current parsing state. Out of the two actions, computationally more expensive is the reduce action.

During a single reduction of length N, the following steps are performed:

1. Top N elements from the stack are removed.
2. Newly exposed top element is used to determine the current parser state.
3. Transition table, the current parser state and the reduced non-terminal is used to determine the next parser state.
4. This state is pushed to the top of the stack.

Because a reduction is so expensive performance-wise, the performance of LR parsers is typically entirely bound by the total number of reductions performed during parsing. In general, the performance of LR parsers can be said to be bound by the amount of *stack activity* needed to parse the input. As such, there have been numerous approaches to reduce the overall stack activity during parsing to increase parsing throughput: typically left recursion is favoured over right recursion, as it leads to lower stack growth.

A more involved and recent approach for reducing stack activity is reduction incorporated parsers [SJ05]. At the cost of significantly increased number of parser states (and thus transition table), it is possible to record target state index as part of reduction entry in transition tables. As a result, such parsers in many cases no longer need two separate transition table lookups to perform a single reduction. Where a typical reduction entry contains only the non-terminal symbol being reduced and the reduction length, an incorporated reduction entry additionally contains a target state index which determines the next state of the parser, thus eliminating the third step of reduction sequence.

Figure 20: A simple `north` grammar

```
rule A() { parse "a"; }
rule B() { parse "b"; }
rule C() { parse "c"; }
rule D() { parse "d"; }
rule AB() { parse A | B; }
rule CD() { parse C | D; }
rule ABCD() { parse AB | CD; }
rule main() { parse ABCD; }
```

1.4.4.2 The cost of a reduction in SEVM

Just like in LR parsers, reductions in SEVM are computationally too quite expensive. During each reduction, the following steps are executed:

1. The newly created reduction is checked against existing reductions. If a matching reduction exists, further reduction processing is aborted.
2. The positive match specifier part of each suspended task is checked against the new reduction and if any matches are found, the task is resumed.
3. The new reduction is added to the reduction list of the current chart entry.

The 2nd step of the reduction process is very costly in particular: each suspended task may be awakened more than once, if the suspended task positive match specifier contains several abstract rules that match the same reduction. Also, under certain conditions this step due to the way subset construction works in SEVM can be avoided entirely.

1.4.4.3 Reduction incorporation in SEVM

Optimized rules in SEVM are entirely defined by a call specifier and the currently used grammar. Consider the grammar shown in fig. 20. When optimizing `main` rule, the direct call to rule `ABCD` would be replaced to a dynamic call with call specifier `[REDUCE_ID(: ABCD), 0]`. Then optimizer would use this call specifier to drive subset construction and generate the optimized version of `ABCD`.

Because `ABCD` starts with both `AB` and `CD` rules, both of these rules would be merged into optimized version of `ABCD`. Continuing this process recursively, optimizer would merge `ABCD`, `AB`, `A`, `B`, `CD`, `C` and `D` rules in this order. Eventually MIR code shown in fig. 22 would be produced (to make the MIR more readable, DFA extraction was disabled).

Consider this step sequence, which would be executed for parsing character `a`:

1. Task 0 starts execution in basic block `#0`.
2. Task 0 is forked (queued) into task 1 with state `#2`.
3. Task 0 is suspended in `#1`.

Figure 21: Optimized MIR rule `ABCD` shown in fig. 20

```
#0: CtlFork #1, #2
#1: CtlMatchSym :AB => #9, :A => #7, :B => #7, :CD => #9, :C => #8, :D =>
    #8
#2: CtlMatchClass 'a' => #3, 'b' => #4, 'c' => #5, 'd' => #6
#3: StmtReduce REDUCE_ID(:A), NORMAL
    CtlStop
#4: StmtReduce REDUCE_ID(:B), NORMAL
    CtlStop
#5: StmtReduce REDUCE_ID(:C), NORMAL
    CtlStop
#6: StmtReduce REDUCE_ID(:D), NORMAL
    CtlStop
#7: StmtReduce REDUCE_ID(:AB), NORMAL
    CtlStop
#8: StmtReduce REDUCE_ID(:CD), NORMAL
    CtlStop
#9: StmtReduce REDUCE_ID(:ABCD), NORMAL
    CtlStop
```

4. Task 1 executes `#2` and matches character `a`, which transfers control to `#3`.
5. Task 1 reduces `A`, spawning a copy of task 0 (now named task 2) in state `#7` as a result.
6. Task 1 is discarded with `CtlStop`.
7. Task 2 reduces `AB`, spawning a copy of task 0 (now named task 3) in state `#9`.
8. Task 2 is discarded with `CtlStop`.
9. Task 3 reduces `ABCD`, causing the callee of `ABCD` to be resumed.
10. Task 3 is discarded with `CtlStop`.

It is important to node, that tasks 2 and 3 were created only to perform a single reduction, after which both were terminated.

Another important observation to be made is that all merged rules `ABCD`, `AB`, `A`, `B`, `CD`, `C` and `D` **share their origin**. In other words, they start at the same input position during parsing. Because of this, we can statically determine which *internal reductions* lead to which states.

An *internal reduction* is reduction that occurs within optimized MIR, but that is not also part of the call specifier. In the current example, reductions for `A`, `B`, `C`, `D`, `AB` and `CD` are internal. Reductions which are part of call specifier are called *external reductions*, because the effect of the reduction will be transferred beyond current optimized rule.

All control transfers for internal reductions during reduction process may be resolved statically: by definition, the effect of internal reductions does not extend beyond the current rule. As such, the rule which performs internal reduction must have been also invoked from the same optimized (merged) rule. Furthermore, only rules with are part of

Figure 22: Optimized MIR rule `ABCD` shown in fig. 20 with partial reduction incorporation

```
#0: CtlMatchClass 'a' => #1, 'b' => #2, 'c' => #3, 'd' => #4
#1: CtlReduceShort REDUCE_ID(:A), NORMAL => #5
#2: CtlReduceShort REDUCE_ID(:B), NORMAL => #5
#3: CtlReduceShort REDUCE_ID(:C), NORMAL => #6
#4: CtlReduceShort REDUCE_ID(:D), NORMAL => #6
#5: CtlReduceShort REDUCE_ID(:AB), NORMAL => #7
#6: CtlReduceShort REDUCE_ID(:CD), NORMAL => #7
#7: StmtReduce REDUCE_ID(:ABCD), NORMAL
    CtlStop
```

rule prefix are merged into optimized rule. Because of this, `StmtReduce` of internal reductions will always match the match specifier of `CtlMatchSym` instruction, which will be always located at the start of optimized MIR.

To statically resolve reductions in SEVM, a new instruction is needed: `CtlReduceShort`. In addition to performing a shortened version of reduction process, which is only suitable for internal reductions, this instruction will also transfer control to statically resolved target state, bypassing the normal reduction process of SEVM.

Fig. 22 shows the optimized MIR for `ABCD`, but with partial reduction incorporation enabled. With this MIR, parsing character `a` is significantly more straightforward:

1. Task 0 starts execution in basic block `#0`.
2. Task 0 executes `#0` and matches character `a`, which transfers control to `#1`.
3. Task 0 internally reduces `A`, transferring control to `#5`.
4. Task 0 internally reduces `AB`, transferring control to `#7`.
5. Task 0 reduces `ABCD`, causing the callee of `ABCD` to be resumed.
6. Task 0 is discarded with `CtlStop`.

Only a single instance of a task is now needed (instead of 4). Furthermore, calls that are part of the prefix no longer require `CtlMatchSym`, because the control transfer of internal reductions is handled directly by `CtlReduceShort`. As a result, the reduction incorporated version of `ABCD` performs 3 less reductions and 1 less task suspension. This optimization also yields significant performance gains in real-world programming languages, as shown in chapter 2.7.3.

On final note, reductions in SEVM are "only" partially incorporated is because only reductions that are part of optimized rule prefix (and are not part of call specifier) are incorporated. All other reductions are processed normally.

1.4.5 Garbage collection

The purpose of the garbage collector in SEVM, just like EVM, is to remove no longer needed information from memory, so it may be reused again. Because of the changes in SEVM structure, the original garbage collector of EVM is no longer suitable.

The memory in SEVM is freed-up by removing potentially unneeded chart entries from the parser's chart. The condition for removing entries from the chart is based on a heuristic, and as a result may remove entries that may still be needed later during parsing. Ideally, such situation would not occur often, and if it did, these chart entries would have to be recreated by re-parsing fragments of input.

The heuristic for determining the usefulness of a chart entry is based on the following observations:

- Entries that have active tasks within them will always be needed later and as such must not be freed.
- Parsing is typically done sequentially, with relatively little significant jumps due to ambiguities/backtracking.
- Ambiguities and backtracking are typically localized.

The current parsing position can be determined by inspecting the currently active chart entry, whose index will be stored on top of the primary execution stack. All entries, whose positions are lower than the current position and which do not contain any active tasks are marked for removal during garbage collection.

Garbage collection occurs every GC_{iter} number of `resume` invocations. Higher GC_{iter} means that the garbage collector will run more rarely and thus may lead to higher memory usage. Too low GC_{iter} may lead to premature chart entry elimination, which may cause SEVM to re-parse the same input fragments repeatedly.

The desired GC_{iter} is chosen by manually inspecting the parse times of sample inputs and setting it to a value higher than GC_{min} (typically $3 * GC_{min}$). GC_{min} refers to the minimum value of GC_{iter}, below which a significant number of premature entry removals occur.

The number of premature entry removals can be measured in `north` by running in a mode, which partially disables the garbage collector: in this mode, the garbage collector instead of removing those entries, only marks them as removed. If an entry with remove flag set is reused in the future, then the remove flag is unset and the number of premature entry removals is increased by 1.

Such strategy of garbage collection may not be optimal (and may lead to significant slow-downs in worst-case ambiguity scenarios), however it works well when used with real-world programming language grammars: heap usage and processor time profiling reveal that the parser's runtime uses only minor amounts of memory (with the garbage collector enabled, the parse-tree becomes the largest memory consumer in `north`

, followed by the index map of the chart), while taking insignificant amount of time to execute (below 5% of total execution time with ANSI C and Rust grammar tests).

1.5 Avoiding exponential complexity

Original EVM had one primary way to avoid exponential parsing complexity: the trace. The trace in EVM was a set stored in each state containing fiber snapshots of previous parse positions. Whenever a new fiber was created, the contents of that fiber were checked against the trace: if the new fiber was a duplicate of a previously created fiber, the creation process was aborted, otherwise, the copy of the new fiber was added to the trace and the new fiber was readied for execution.

This had several positive effects:

- Any form of infinite left recursion was eliminated, because it would result in two identical fibers in the same state.
- Exponential complexity of parsing was eliminated, when using trace with reduction duplication and incremental parse-tree construction: multiple reductions of the same type and length were merged together, multiple resumptions of same task but with different reduction were also merged when the task was resumed in the same instruction pointer.

However, while using trace for reduplicating fibers was simple and powerful, it also meant that creating new new fibers was performance-wise expensive, because each fiber had to be checked against trace first, and then a copy of that fiber had to be made in case the newly created fiber was unique. As a result, a new method for avoiding exponential complexity is needed.

Firstly, it's important to identify the situations that can lead to exponential complexity (and potentially hidden infinite recursion). We call these situations *conflicts* (the term is inspired by shift/reduce and reduce/reduce conflicts of (G)LR parsers [Knu65]), as they potentially may lead to multiple parse paths. There are four types of conflicts in SEVM:

- **Reduce/reduce conflict**. These conflicts occur when two reductions of same type and length occur at the same starting origin. Resuming a task with both reductions may lead to exponential parsing complexity, because the same task will be resumed with duplicate reduction twice, which may lead to further conflicts.
- **Resume/resume conflict**. These conflicts are closely related to reduce/reduce conflicts. They occur, when two reductions occur of the same length, but with different *reduce_id* and result in resumption of the same task, in the same *state_id* twice. The existence of resume/resume conflict indicates that a rule has an ambiguous, but fixed-length prefix with matching suffix. Performing both resumptions means that the matching suffix is parsed multiple times, potentially leading to further conflicts.

- **Call/call conflict**. These conflicts occur, when the same non-terminal rule is called multiple times at the same position. Executing both calls would mean that the same input segment is parsed multiple times with same grammar rule(s). The callees may perform further calls, which may lead to more conflicts and/or infinite hidden recursion.
- **Match/match conflict**. These conflicts occur, when the same task is suspended at the same position twice. Accepting both matches may lead to scenario, where one reduction would awaken both tasks, which may lead to other conflicts (and exponential parsing complexity).

Reduce/reduce conflicts can be solved by merging reductions: when a ambiguous reduction occurs (this can be trivially detected, because the list of all reductions is stored in each chart state), then the *tree_id*s of both reductions can be merged to form an ambiguous node, representing two alternative parse paths. Then further reduction processing is aborted: that way ambiguous reductions do not wake additional suspended tasks.

Resume/resume conflicts can be eliminated by keeping a list of resumptions in each suspended task. Only *reduce_id*, reduction length and reduction *tree_id* need to be stored. Whenever a duplicate resumption occurs (with same *reduce_id* and length pair), instead of resuming the task again, the corresponding *tree_id*s are merged, forming an *ambiguous shift* node in the parse tree.

Call/call conflicts can be eliminated by making use of the following observation: whenever a new task is called, it's callee is soon after suspended with a `CtlMatchSym` instruction. As a result, it is possible to reconstruct the list of called tasks at a specific position based on match specifiers stored in the list of suspended tasks. This can be implemented by pairing each match specifier with a bit mask, where the bit mask represents the set of concrete rules that were called. By performing bitwise-or operator between these masks it is possible to efficiently recreate the set that represents all the concrete rules that have been called so far at this position. If the newly called task is a subset of the previously called concrete rules, then the completion of the call can be aborted, because all of the currently called rules have been called before.

Finally, merge/merge conflicts can be eliminated by ensuring that newly added suspended tasks are unique to that state. However, it is possible to completely remove this conflict mitigation (or make it optional) to increase the overall parsing performance, because merge/merge conflicts are quite rare and only happen when a task is resumed twice at the same position, but with reductions of two different lengths.

1.6 Parse-tree construction

ParseTree.tex

2 Evaluation

2.1 Overview of evaluation process

In this chapter we present evaluation of SEVM. The primary focus is to evaluate the relative performance of SEVM compared to other parsing implementations.

2.2 Language selection

Because one of the goals of `north` is to prove that a scannerless generalized parsing algorithm may be used for parsing in practise, two existing programming languages were chosen to be used in comparison:

- **ANSI C**. It is one of the most widely used programming languages and as such any parsing algorithm with the goal of parsing programming languages should be able to parse such language. It is also commonly used for comparing parser performance.
- **Rust**. Rust is relatively new programming language, but one that is quickly gaining popularity. It's grammar is significantly larger in size compared to ANSI C and it is also mostly ambiguity-free (when viewed as a context-free grammar).

An additional note regarding parsing ANSI C: it is often claimed that ANSI C is a simple language: and this statement is true in respect to the grammar size of ANSI C (when compared to other programming languages). However one key aspect that makes parsing ANSI C deterministically more complicated is the fact that most grammars used to parse ANSI C (including the one specified in the ANSI C standard) depend on the ability to disambiguate identifiers from type names during lexing/parsing. In other words, in order to parse ANSI C code deterministically, the parsing method needs to perform limited version of semantic analysis (namely, name resolution) during parsing. Otherwise, statements such as `a * b;` may be both interpreted as multiplication and as a declaration of pointer `b` with type `a`. This happens to be the case where generalized methods become more useful: they are capable of parsing this input with both interpretations and to produce parse forest, which then can be filtered *after* parsing based on semantic predicates. As such, using generalized parsing methods to parse C programming language allows to separate parsing from semantic analysis and thus to improve separation of concerns.

In this comparison, ANSI C parser implemented with `bison` performs limited semantic analysis during parsing, because it is used as a LALR(1) parser. Other ANSI C parser implementations support generalized parsing and instead produce parse forests when ambiguities are encountered.

Rust programming language in this sense is a complete opposite of ANSI C: it's grammar is larger, but it does not require performing any semantic analysis during parsing.

As a result, these two languages, ANSI C and Rust, should sufficiently cover both ambiguous and unambiguous use cases of parsing.

2.3 Implementation selection

The following parser implementations are included in this evaluation:

- **north**: it's the implementation of SEVM described in this work, written in Rust programming language.
- **bison** with **flex**: `bison` is a yacc-compatible LALR(1) parser generator. It is perhaps de-facto LALR(1) parser generator. It it/was used in various prominent open-source projects, such as: Bash, GCC before v3.4, Perl 5, PHP and others. It is commonly taught in universities and has integrations for wide variety of programming languages. Because `bison` works only with tokens (it's not a scannerless parser), a lexer is needed in order to be able to parse textual inputs. As such, lex-compatible `flex` was chosen, which is commonly used in conjunction with `bison`.
- **yaep** with **flex**: `yaep` is one of the few complete (as of writing this work) implementations of Earley parser with various optimizations to make it suitable for use in practise. It is also a non-scannerless implementation, and thus is used in conjunction with `flex` during evaluation.
- **dparser**: `dparser` is a scannerless implementation of GLR parsing algorithm. It is one of the very few still maintained projects capable of generalized scannerless parsing. Therefore `dparser` is the closest match in this list to `north`.
- **syn**: `syn` is a parser for Rust programming language, implemented with a hand-written recursive descent parser. It is a non-scannerless parsing method, but comes with it's own lexer and as such, no external lexer is needed. `syn` is primarily used as a library for developing language extensions for Rust programming language.

2.4 Comparison method

In order to compare multiple parser implementations, a tool called `bench_parsers` was created. The tool works by executing a series of scenarios, where each scenario is repeated multiple times to gain reliable measurement data. See appendix Nr. 1. to learn how to use the tool or how to reproduce the results of this evaluation.

Each scenario is comprised of the following steps:

1. An input file containing the source code to be parsed is read.
2. An accurate measurement of system time is made called *start_time*.
3. The input file is lexed, if the parsing method being tested requires a dedicated lexer. Otherwise this step is skipped.
4. The input is parsed. Some of the parsing methods may produce parse trees or abstract syntax trees during parsing.

5. An accurate measurement of system time is made called *end_time*.
6. *end_time − start_time* of each scenario is added to a vector.

As mentioned above, each scenario is run multiple times. After these runs are complete, the results are stored to a CSV file, which later can be analysed. Before each set of scenarios, the current parsing implementation is run for at least 3 seconds (potentially by repeating the current test multiple times) as a warm-up to avoid any result irregularities related to input/output caching (either at hardware level, or at kernel/file system level), dynamic CPU frequency scaling and others.

To evaluate `north` on it's own, an additional tool called `north_cli` was developed. It allows to observe internal state of SEVM parser and to obtain other metrics (such as the amount of shortened reductions that were performed during parsing). See appendix Nr. 2. for more information about this tool.

2.5 Test environment

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

- **Processor**: Intel i9-3930k.
- **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.

2.6 Test data

Two primary files are used as inputs for benchmarking `north` and other parsing methods:

1. `input_gcc_470k.i` is ANSI C source file taken from `yaep` parser benchmark suite. It contains preprocessed source-code of entire GCC 4.0 compiler. The file is 14.8 MB in size and consists of ∼475000 lines of code.

Table 4: A chart showing the median time needed to parse sample inputs

Parser	Language	N	IQR	% Outliers	Median
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

2. `input_rust_650k.rs` is Rust source file that contains the entire implementation of the Rust Compiler. The file is created by concatenating every Rust source file (excluding tests, some of which may not be syntactically correct) of GitHub Rust repository. Minor modifications were performed to the resulting file, to ensure that the concatenated file is still syntactically correct (some Rust language constructs may only appear in the beginning in the file, and thus not all source files can be simply concatenated and result in a valid Rust source code). These modifications were primarily performed so the `syn` parser without any additional modifications would be capable of parsing the resulting file. The file is 22.3 MB in size and consists of ∼650000 lines of code.

Both of these input files represent larger than average projects and should cover every use of ANSI C and Rust grammars.

2.7 Test results

2.7.1 Relative performance comparison

The relative performance comparison results of different parser implementations are shown in table 4.

Out of all tested ANSI C parsing methods, `bison` was unsurprisingly the fastest. It's token-based, fully deterministic parsing method that performs no variable-length lookahead or backtracking. Because it's a LALR(1) parser, limited form of semantic analysis was performed during parsing to disambiguate identifiers from type names. It's also important to note that ANSI C `bison` parser only performs recognition and constructs no parse-tree or AST as result.

`yaep` parser is slightly less performant than `bison`, but it's significantly more general, as it's an Earley parser. It still requires the use of dedicated lexer, however, no semantic analysis was performed during parsing, because Earley parsers can produce ambiguous parse forests to represent different parse paths.

`north` is ≈9.3 times slower than `bison`, but it's the first parser in the list that's not only fully general, but also scannerless. Just like in the case of `yaep`, SPPF is used to represent ambiguous parses.

Finally, the scannerless, GLR-based `dparser` comes last in this list.

Table 5: Table showing the median time needed to parse `input_gcc_470k.i` with and without garbage collection in `north`

Benchmark	N	IQR	% Outliers	Median
ANSI C	2	0.0150	0.0	5.3165
ANSI C (with GC)	2	0.0035	0.0	4.6139

Table 6: Table showing the median time needed to parse `input_rust_650k.rs` with and without garbage collection in `north`

Benchmark	N	IQR	% Outliers	Median
Rust	2	0.0069	0.0	6.9651
Rust (with GC)	2	0.0198	0.0	6.3965

For testing Rust grammars, only one other parsing method was tested, because Rust is a relatively new programming and complex language and beyond the parser used in the Rust compiler itself, there exists only one additional Rust parser implementation: the `syn` parser, which is a hand-written predictive recursive descent parser. While it is faster than `north` for parsing Rust, it is so only by a narrow margin.

It should be also noted the amount of time if takes for `north` to optimize, JIT and otherwise pre-process grammars is included in the final running time in all of the `north` benchmarks. If all of the preprocessing was done statically before parsing, then significant gains of parsing performance may be achieved, at a cost of sacrificing grammar extensibility, which is one of the key factors that sets SEVM/`north` apart from other parsing algorithms and implementations.

2.7.2 Performance influence of garbage collector

The primary purpose of garbage collector in SEVM/`north` is to reduce memory usage of the parser. It works, as described in section 1.4.5, by periodically scanning all of the currently existing chart entries and removing the ones that are believed to be no longer needed. Because the unneeded entries are identified by a heuristic, it is possible that the garbage collector may remove a chart entry that will be needed in the future. When that happens, SEVM runtime has to recreate the required chart entries by reparsing corresponding input fragments.

As such, initially it may seem that that the garbage collector should reduce the overall parsing performance because of the following reasons:

- Scanning all the existing chart states and deleting the unneeded ones takes additional processor time.
- In case a required entry is removed that entry will have to be recreated in the future.

To see the actual performance impact of parsing ANSI C and Rust, additional tests were carried out: ANSIC (`input_gcc_470k.i`) and Rust (`input_rust_650k.rs`) in-

Table 7: Table showing the time needed to parse `input_gcc_470k.i` with and without reduction incorporation in `north`

Benchmark	N	IQR	% Outliers	Median
ANSI C	2	0.0054	0.0	6.9844
ANSI C (with RI)	2	0.0060	0.0	4.6619

puts were parsed both with and without using garbage collector. The results of these tests are displayed in tables 5 and 6.

Surprisingly, enabling the garbage collector not only lowered the overall memory usage, but also improved overall performance: ANSI C and Rust input parsing times are faster by approximately 13% and 8% respectively. This can be explained by the following reasons:

- Enabling the garbage collector allows reusing previously allocated memory fragments and therefore is more processor cache-friendly, which significantly improves the overall performance of the parser.
- Because the parser uses less overall memory, it means that less system calls are needed for allocating new memory blocks.

Because of the improved performance and lower memory usage, the garbage collector in `north` is enabled by default.

2.7.3 Performance influence of incorporated reductions

Partial reduction incorporation (described in chapter 1.4.4) is a further optimization made possible by performing MIR subset construction. In short, whenever a reduction is performed, SEVM runtime checks the list of suspended tasks in the origin entry of the task that is performing the reduction and resumes the appropriate task. This process is consists of several steps that are computationally expensive:

- Iterating though all suspended tasks requires N_{susp} steps, where N_{susp} is the number of suspended tasks in the origin entry.
- In order to determine if a suspended task needs resumed, it's match specifier needs to be matched against the *reduce_id* of the reduction. This matching is performed by using a hash table.
- Finally, when it is know that a suspended task can be resumed, a copy of it is made in the target state.

In order to test the effect of reduction incorporation on parsing performance, additional tests were carried out: ANSI C and Rust inputs (`input_gcc_470k.i` and `input_rust_650k.rs`) were parsed both with and without enabling partial reduction incorporation. The results of these tests are shown in tables 7 and 8.

Table 8: Table showing the time needed to parse `input_rust_650k.rs` with and without reduction incorporation in `north`

Benchmark	N	IQR	% Outliers	Median
Rust	2	0.0101	0.0	9.1468
Rust (with RI)	2	0.0076	0.0	6.4659

Reduction incorporation on average improves the parsing times in both tests approximately by 33% and 29% respectively. This significant performance boost comes from two primary sources:

- The short reductions make up a significant part of all reductions and are less expensive computationally.
- Rules with all reductions partially incorporated no longer need to be suspended at origin position. Therefore each call to a rule with partially incorporated reductions results in one less task suspension. This in turn means that there are overall less suspended tasks, which causes new (normal) reductions to execute faster, because each reduction needs to traverse a shorter suspended task list.

Reduction incorporation has one key negative effect: the optimized MIR grammars become language (*grammar_id*) dependant and can be no longer reused when dynamically switching to other grammars. As such, in workloads where a parser has to parse input which is described by several closely related grammars it may be desirable to disable partial reduction incorporation.

2.7.4 Performance influence of recursion type

In order to test the performance influence of recursion type, two additional synthetic test inputs were created:

- `input_a_1k.txt` contains 1000 characters `a`, followed by a semicolon and a newline.
- `input_5a_10k.txt` contains 10000 lines of text, where each line contains `aaaaa;`.

The first file is meant to test the worst-case scenario with deep recursion. The second file is designed to test a more realistic scenario, where recursion depth is not as high, however there are more instances of recursion use, such as binary expressions of various programming languages.

The inputs are then parsed with grammars shown in figures 23 and 24 both with and without reduction incorporation.

The results for parsing `input_a_1k.txt` are shown in table 9. This test scenario triggers quadratic complexity when performing right recursion, and therefore right recursion is on average two orders of magnitude slower than left recursion. This is a

Figure 23: Left-recursive test `north` grammar

```
rule_dyn expr();

group _: expr(0) {
  rule expr_base() { parse "a"; }
  rule expr_suffix() { parse (expr!, expr_base) }
}

rule main() {
  parse ((expr, ";")+, "\n");
}
```

Figure 24: Right-recursive test `north` grammar

```
rule_dyn expr();

group _: expr(0) {
  rule expr_base() { parse "a"; }
  rule expr_prefix() { parse (expr_base, expr!) }
}

rule main() {
  parse ((expr, ";")+, "\n");
}
```

Table 9: Table showing the benchmark results for parsing `input_a_1k.txt` with left and right recursive grammars

Benchmark	N	IQR	% Outliers	Median
Left assoc.	10	0.0001	0.0	0.0089
Left assoc. (with RI)	10	0.0001	0.0	0.0076
Right assoc.	10	0.0101	10.0	0.7660
Right assoc. (with RI)	10	0.0020	0.0	0.7527

Table 10: Table showing the benchmark results for parsing `input_5a_10k.txt` with left and right recursive grammars

Benchmark	N	IQR	% Outliers	Median
Left assoc.	10	0.0001	20.0	0.0371
Left assoc. (with RI)	10	0.0003	10.0	0.0311
Right assoc.	10	0.0002	10.0	0.0522
Right assoc. (with RI)	10	0.0004	0.0	0.0373

well known characteristic of Earley parsers, and is inherited by SEVM/`north` as well. Optimizations to eliminate quadratic complexity of right recursion in Earley parser exists [Leo91], however they are not implemented in `north`.

The results for parsing `input_5a_10k.txt` are shown in table 10. In this scenario, the difference in parsing times between left and right recursion is significantly lower, because the recursion depth is limited to 5 layers of rule calls (as opposed to 1000 in the previous test). This represents a more realistic scenario, because of the following observations:

- Repetition in SEVM (unlike in most other parsing methods) is performed with repetition operators and not recursion.
- Recursion is still used for binary expression operators, however most operators in common languages (such as C, C++, Java, Rust) are left recursive.

As expected, left recursion is faster than right recursion in all scenarios, however when the recursion depth is low, then the difference is not that large (≈17% when recursion depth is 5).

Partial reduction incorporation also provides a significant performance improvement (15% to 30%) in both left and right recursive grammars. This is because whenever a call to a rule is part of the caller prefix (when it is part of the FIRST set), that call can be incorporated: in right recursive grammars calls from `main` to `expr` and from `expr_prefix` to `expr_base` can be incorporated.

2.8 Validity

2.8.1 Internal validity

The following steps were taken to ensure internal validity of the evaluation results:

- All benchmarks are run in Linux runlevel 3. This means that no desktop applications were running in the background while executing the tests. That way any potential unwanted influences of the operating system and the environment are minimized.
- All tests were run in the same environment with the same configuration.
- Each benchmark/test scenario was run multiple times to obtain more consistent data.
- Before running a set of benchmarks, each test scenario was warmed up for at least 3 seconds to reduce any influence of hardware level/file system level caching, as well as to ensure that dynamic CPU frequency scaling policy does not influence the results.
- After running all of the scenarios, outlier detection (IQR method) was carried out to ensure the consistency of the data: even though the tests were performed in a fairly isolated environment, it was still possible for operating system background services to awaken during execution of the tests and interfere with the execution, potentially lowering the performance of an individual test run and causing an anomaly in the test results. As such, large number of outliers would suggest the existence of unwanted external influences.

- Other tests that are not performance dependant are deterministic and only depend on the parser's input and grammar. As such no external influences can interfere with such tests.

2.8.2 External validity

To test the performance of `north`, two grammars of popular programming languages were chosen as test objects: ANSI C and Rust. Both of these languages are widely used in practise (especially ANSI C). As such, there are two primary questions regarding generalization of results:

1. Do benchmark results of `north` generalize to other inputs in the context of ANSI C and Rust languages?
2. Do benchmark results of `north` generalize to other untested languages and their grammars that are used in practise?

The first question is easier to answer: the test inputs (the source files used for parsing) that were chosen represent significantly larger than average inputs. The input files are made of unique concatenated input source files and as such cover the majority (if not entirety) of the input grammars. That means that it is highly likely that any potential slow paths that negatively effect the performance of the parser would have been reached during parsing of these files. And indeed, during early stages of development of `north` there were several occurrences of exponential complexity behaviour, but that was before current exponential complexity avoidance techniques were implemented.

It is still possible that some edge cases remain in existing parser implementation that may result in unexpected performance loss, however they would then be regarded as implementation bugs rather than systemic issues with the overall parsing method of SEVM or it's implementation `north`. Another important observation is that the only way to achieve a significant performance loss in `north` is to increase the ambiguity of the input grammar. Otherwise, the performance of `north` would be linear. To lower probability of such performance issues occurring, additional metrics are generated during parsing in `north`, which would highlight potential areas of ambiguity within test inputs. These metrics primarily indicate the number of suspended tasks and completed reductions per each chart entry. High average values of suspended tasks and reductions indicate high overall ambiguity of input grammars, while unexpected high peaks of suspensions and reductions indicate a potential problem area, with higher than linear asymptotic complexity. However, during testing all of the collected metrics remained in line with the expectations.

It is also important to note that parsing performance is a concern only when parsing such large inputs, because parsing anything several orders or magnitude smaller would result in insignificant CPU time. As such no tests with tests of minor size were carried out.

As such, it may be indeed concluded that the performance of `north` will generalize to other inputs of ANSI C and Rust.

The other question is whether or not the performance of `north` will generalize to other grammars used in practise?

To answer this question, additional observations need to be made:

- Many existing programming and mark-up languages have been designed with simpler parsing methods in mind: primarily, many of such languages can be parser either with simple LALR(1) parsers or with recursive descent parsers.
- Very few languages require any semantic analysis to be performed during parsing (C/C++ are the exception to this rule). The languages that do require semantic analysis for parsing, can be parsed in `north` or other generalized parsing methods ambiguously and filtered after parsing [vdBSVV02]. As such, ANSI C language and it's input can be considered as the worst-case real-world scenario regarding the ambiguity of the input grammar in `north`.

As a result, ANSI C covers the ambiguous case of inputs and tests the code paths in `north` that deal with such ambiguities. Conversely, Rust represents the non-ambiguous case, where the input is deterministic and covers the real-world languages and inputs that are non-ambiguous.

Further differences of performance in `north` arise from different recursion use (left versus right recursion) and depth of overall grammar.

While left recursion is more efficient in SEVM, primarily due to the fact that left-recursion can be partially incorporated and can avoid much of the complex machinery of new reduction handling, right recursion is still offers acceptable performance (as indicated by synthetic tests, the performance of right recursion is lower by a constant factor).

The grammar depth is another factor that affects overall parsing performance, but this happens in every parsing algorithm and implementation: recursive descent parsers require more calls and returns to parse grammars with higher depth, while bottom-up parsers such as LR/LALR/GLR require more reductions.

Finally, it is important to note that the important takeaway of these test results is not the exact absolute performance values, but relative performance of `north` to other parsing implementations, as the goal of SEVM is to be a suitable replacement for such parsers. As such, even if minor performance fluctuations were to occur, they would not significantly impact the overall result of this study: that SEVM is becoming a viable alternative to more traditional parsing methods, even though it still requires some further research and improvements in certain areas.

2.9 Conclusions

We have created an implementation for SEVM parser called `north`. Then we implemented ANSI C and Rust grammars for `north`, which then were used for performance evaluation. We compared `north`'s performance against several other parser implementations and found that a proper SEVM may be indeed used in practise to parse real-world programming languages.

However, further research is needed in the following topics:

- Further SEVM optimizations. SEVM may be further optimized by eliminating external stacks and driving execution with native recursion, much like it is done in Packrat parsers [For02].
- Greedy calls and ordered choice. These additional operations should be added to SEVM to boost its disambiguation capabilities.
- Error reporting. `north` currently implements no error reporting, but this can be done by analysing the contents of *susp_list* in the final chart entry.
- Error recovery. SEVM aborts execution upon encountering first parse error. It should be modified, so the parsing process may continue (by skipping fragments of invalid input). Error recovery algorithms for Earley parser exist, but none of them are designed for scannerless parsing [AB81]. There have been some work on error recovery in SGLR parsers [Val07], but it's uncertain how well such method may translate to SEVM.

References

[AB81] S. O. Anderson and R. C. Backhouse. Locally least-cost error recovery in earley's algorithm. *ACM Trans. Program. Lang. Syst.*, 3(3):318–347, July 1981.

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

[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.

[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.

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

[Leo91] Joop M.I.M. Leo. A general context-free parsing algorithm running in linear time on every lr(k) grammar without using lookahead. *Theoretical Computer Science*, 82(1):165 – 176, 1991.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

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

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

[Val07] R Valkering. Syntax error handling in scannerless generalized lr parsers. *Physica D-nonlinear Phenomena - PHYSICA D*, 01 2007.

[vdBSVV02] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized lr parsers. In R. Nigel Horspool, editor, *Compiler Construction*, pages 143–158, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

Appendixes

Appendix Nr. 1.
bench_parsers utility

`bench_parsers` utility is designed to obtain accurate benchmarks of various parsing methods, `north` included.

`bench_parsers` may be run with `cargo` utility of Rust programming language with:

```
cargo bench -p bench_cli -- <TEST_NAME>
```

The following benchmarks are available:

- `assoc_a`: recursion performance test A.
- `assoc_b`: recursion performance test B.
- `benches`: relative parser performance comparison.
- `gc`: a benchmark for testing the impact of garbage collection to parsing performance.
- `ri`: a benchmark for testing the impact of partial reduction incorporation to parsing performance.

Appendix Nr. 2.
north_cli utility

`north_cli` utility enables to test `north` implementation of SEVM. Users can inspect SEVM parsing process, resulting parse-tree and additional metrics by providing an input grammar file and input data file.

In order to parse a sample file with a provided grammar, `north_cli` must be launched by supplying the following required options:

```
./north_cli parse -g <grammar_file> -i <input_file>
```

This will cause the input grammar file to parsed and analysed, after which the grammar MIR will be generated, which then will be used during parsing/subset construction to parse the provided input file. No output will be printed if the parsing was successful.

There are additional options that can be supplied to `north_cli` to augment the parsing process and/or reported information:

- `-G` disables the garbage collector. This will cause the parser to use significantly more memory.
- `-I` disables the partial reduction incorporation.
- `-p` prints the timing information for significant parts of the parsing process.
- `-m` shows the unoptimized MIR which is directly constructed from the input grammar.
- `-o` shows optimized MIR fragments immediately after they are constructed.
- `-r` shows reduction trace. This allows to trace the execution of the parser.
- `-t` shows the resulting parse-tree. It will only be printed if the parsing process was successful.
- `-e` shows the additional metrics after parsing. Some of the shown metrics are: number of reductions per chart entry histogram, number of suspended tasks per chart entry histogram, total number of reductions, number of duplicate reductions, allocator information and others. Some of this information may be meaningful only when parsing with garbage collector disabled.

It should be noted that MIR printed by `north_cli` will shown in a slightly different dialect compared to the rest of this work. The dialect for visualising MIR was simplified to make it more compact and suitable for embedding fragments of it to this work.