VILNIUS UNIVERSITY

Audrius
ŠAIKŪNAS

Extensible parsing with Earley virtual machines

DOCTORAL DISSERTATION

Technological Sciences,
Informatics Engineering T 007

VILNIUS 2019

The dissertation has been prepared at Vilnius University from 2015 to 2019.

Scientific Supervisor:

Prof. Dr. Albertas Čaplinskas (Vilnius University, Technological Sciences, Informatics Engineering – T 007).

VILNIAUS UNIVERSITETAS

Audrius
ŠAIKŪNAS

Earley virtualiųjų mašinų panaudojimas plečiamai programavimo kalbų sintaksinei analizei

DAKTARO DISERTACIJA

Technologijos mokslai,
Informatikos inžinerija T 007

VILNIUS 2019

Disertacija rengta 2015–2019 metais Vilniaus universitete.

Mokslinis vadovas:
prof. dr. Albertas Čaplinskas (Vilniaus universitetas, technologijos mokslai, informatikos inžinerija – T 007).

ABSTRACT

Almost every programming language implementation contains a parser, which is the software component that takes the program source code and builds a data structure that can be used to process and translate the input programs. One recent category of programming languages is extensible programming languages. These are languages whose syntax (and possibly semantics) can be extended without having to modify the compiler. Because the syntax of the language can be changed dynamically, special parsing algorithms that support dynamically changing grammar are needed to analyse such languages. After carrying out a detailed analysis of existing parsing algorithms, we found that no single parsing algorithm (as of the time of writing this work) fully satisfies our requirements for parsing extensible languages. Therefore, we set out to create a new parsing method that would be suitable for parsing such languages. The Earley virtual machine (EVM) is the first iteration of such a parsing method. It is a generalised context-free parsing algorithm that can parse dynamically changing grammars. We carefully describe how such an algorithm was constructed. To ensure that the EVM can parse languages with acceptable performance, a successor to EVM was created: the scannerless Earley virtual machine (SEVM). The SEVM is an enhanced version of the EVM with a focus on optimisation and parsing performance. Finally, to ensure that the SEVM can be used for practical applications, an implementation of the SEVM was developed and compared to various existing parser implementations.
Keywords: Adaptive grammars, extensible programming language, extensible parsing, Earley parser, generalised parsing, just-in-time compiler, reflective parsing, scannerless parsing.

CONTENTS

Abstract 5

Contents 6

List of Figures 10

List of Tables 12

Glossary 14

Acronyms 15

1 Introduction 17
- 1.1 Importance and Motivation . . . 17
- 1.2 Problem Statement . . . 19
- 1.3 Research Goal and Objectives . . . 20
- 1.4 Defended Claims . . . 21
- 1.5 Research Methods . . . 21
- 1.6 Results . . . 22
- 1.7 Scientific Contribution of the Research . . . 22
- 1.8 Practical Significance of the Results . . . 23
- 1.9 Approbation . . . 24
- 1.10 Publications . . . 24
- 1.11 Outline . . . 25

2 Prerequisites 27
- 2.1 Compilers and Programming Languages . . . 27
- 2.2 Compiler Architecture . . . 29
- 2.3 Lexing and Parsing . . . 30
- 2.4 Code Generation . . . 31
- 2.5 Extensibility . . . 32

3 State of the Art 35
- 3.1 Parsing Methods . . . 35
  - 3.1.1 Requirements for a reflectively extensible programming language parser . . . 35
  - 3.1.2 The LR(k) parsers . . . 37
  - 3.1.3 The generalised LR family of parsers . . . 39

3.1.4 Recursive descent parser 40
3.1.5 Packrat parser 42
3.1.6 Adaptable parsing expression grammars 44
3.1.7 Specificity parser 45
3.1.8 Earley parser 46
3.1.9 Parsing reflective grammars 48
3.1.10 Efficient Earley parsing 51
3.1.11 Yakker parser 52
3.1.12 Parsing method summary 56
3.2 Related Tools and Languages 56
3.2.1 Katahdin 56
3.2.2 SugarJ 59
3.2.3 Neverlang 61
3.2.4 Tool and language summary 61
3.3 Conclusions 62

4 Extensible Parsing with the Earley Virtual Machine 63
4.1 Earley Virtual Machine 63
4.1.1 Introduction to the Earley virtual machine 63
4.1.2 Earley virtual machine grammars 64
4.1.3 Earley virtual machine states 65
4.1.4 Earley virtual machine fibers 66
4.1.5 Earley virtual machine interpreter 66
4.2 Compiling Basic Earley Virtual Machine Grammars 69
4.3 General-Purpose Computation in the Earley Virtual Machine . 69
4.4 Improving Source Grammar Flexibility 71
4.4.1 Regular right-hand sides in production rules 71
4.4.2 Rule and operator precedence 72
4.4.3 Specifying operator associativity 74
4.5 Parsing with Regular Lookahead 75
4.5.1 Fixed-length lookahead 75
4.5.2 Variable-length lookahead 76
4.6 Parsing with Data-Dependent Constraints 77
4.6.1 Earley virtual machine grammar language 77
4.6.2 Matching input against dynamic content 79
4.7 Abstract Syntax Tree Construction 81
4.7.1 Automatic abstract syntax tree construction 81
4.7.2 Manual abstract syntax tree construction 84
4.7.3 Delayed semantic actions 86
4.8 Parsing Reflective Grammars 91
4.8.1 Dynamic grammar composition 93

4.8.2 Extensions of the Earley virtual machine grammar language . . . 94
4.8.3 Compiling Earley virtual machine grammars with domains . . . 95
4.8.4 Loading multiple grammar modules in Earley virtual machine . . . 96
4.8.5 Parsing reflective grammars in the Earley virtual machine 97
4.9 Earley Virtual Machine Performance Improvements . . . 97
4.9.1 Garbage collection of suspended fibers . . . 98
4.9.2 Eliminating dynamic non-terminal call indirection . . 99
4.9.3 On-demand instruction subset construction . . . 99
4.10 Conclusions . . . 104

5 Implementation of Scannerless Early Virtual Machine 106
5.1 Scannerless Earley Virtual Machine . . . 106
5.1.1 Flaws of the original Earley virtual machine . . . 106
5.1.2 Overview of the internal structure of the scannerless Earley virtual machine . . . 108
5.2 Improving Grammar Expressiveness . . . 111
5.2.1 Abstract grammar rules . . . 111
5.2.2 Named precedence groups . . . 113
5.2.3 Dominating terminals . . . 116
5.3 Ambiguity Elimination . . . 118
5.3.1 Negative reductions . . . 119
5.3.2 Strict execution ordering in scannerless Earley virtual machine runtime . . . 121
5.3.3 Negative matches . . . 123
5.3.4 Greedy non-terminal repetition . . . 125
5.3.5 Strict execution ordering in the scannerless Earley virtual machine optimiser . . . 125
5.3.6 Token-level ambiguity elimination . . . 128
5.4 Parser Optimisations . . . 130
5.4.1 Profiling the Earley virtual machine . . . 130
5.4.2 Just-in-time grammar compilation . . . 131
5.4.3 Deterministic finite automata extraction . . . 133
5.4.4 Partially incorporated reductions . . . 140
5.4.5 Garbage collection . . . 145
5.5 Avoiding Exponential Complexity . . . 146
5.6 Parse-tree Construction . . . 148
5.7 Conclusions . . . 149

6 Evaluation of the Scannerless Earley Virtual Machine 151
6.1 Language Selection 151
6.2 Implementation Selection 152
6.3 Comparison Method 153
6.4 Test Environment 154
6.5 Test Data 154
6.6 Test Results 155
6.6.1 Relative performance comparison 155
6.6.2 Performance influence of garbage collector 156
6.6.3 Performance influence of incorporated reductions . . . 157
6.6.4 Performance influence of recursion type 159
6.7 Validity 161
6.7.1 Internal validity 161
6.7.2 External validity 162
6.8 Conclusions 164

7 General Conclusions 165

Appendices 166
A bench_parsers Utility 166
B north_cli Utility 166

References 168

LIST OF FIGURES

1 Naive extensible parsing algorithm 36
2 The LR(0) parser algorithm 38
3 A grammar for a language that supports $+$, $*$ operators and the variable a 40
4 A corresponding Ruby program that parses the provided grammar with the help of the `accept()` and `expect()` functions . . 41
5 Two example parsing expression grammars that define conditional expressions and their unions 44
6 Earley parser algorithm 47
7 The modified Earley algorithm 49
8 An example expression that uses the reflective capabilities of the modified Earley parser to add the binary infix $+$ operator . 49
9 An example of a grammar that supports infix + and * operators with the appropriate operator precedence 51
10 Earley items for expanding non-terminal E in position j with the grammar from the expression grammar 52
11 Earley deterministic finite automata for production rules from the expression grammar 53
12 Interconnected Earley deterministic finite automata for production rules from the expression grammar 54
13 An example Yakker grammar that allows parsing fixed-length numbers 55
14 An example Katahdin program that uses Fortran and Python language extensions 57
15 An example Katahdin extension that implements the unary suffix increment operator ++ 57
16 An example of a SugarJ extension that implements the unary suffix increment operator ++ 60

17 A grammar that defines simple expressions with binary operators 72
18 Rewritten grammar that defines simple expressions with binary operators 75
19 Grammar rule that defines identifier using fixed-length lookahead 76
20 Simplified XML tag grammar rule 81
21 Grammar rule for parsing the argument list separated by commas 87
22 The replay function for binary addition in the Ruby programming language 92

23 The replay function for binary addition in the C programming language . . . 92
24 Example domain usage . . . 95

25 Abstract grammar rule example . . . 112
26 A simplified fragment of C99 grammar . . . 114
27 A fragment of C99 grammar rewritten in North . . . 115
28 A `north` grammar rule for parsing ANSI C multi-line comments (Attempt 1) . . . 117
29 A `north` grammar rule for parsing ANSI C multi-line comments (Attempt 2) . . . 117
30 A `north` grammar rule for parsing ANSI C multi-line comments (Attempt 3) . . . 117
31 Unoptimised MIR for the ANSI C multi-line comment rule . . . 118
32 A grammar rule that defines an identifier followed by a space . 120
33 Unoptimised medium-level intermediate representation for a grammar rule that defines an identifier . . . 121
34 A grammar for parsing identifiers and keywords . . . 128
35 A modified grammar for parsing identifiers and keywords . . . 128
36 A grammar that uses token groups to disambiguate keywords from identifiers . . . 129
37 A `north` grammar for matching three keywords . . . 135
38 An optimised medium-level intermediate representation for matching three keywords . . . 136
39 Traditional deterministic finite automata for matching three keywords . . . 136
40 Optimised medium-level intermediate representation for matching three keywords (with the deterministic finite automata extraction enabled) . . . 137
41 Scannerless Earley virtual machine deterministic finite automata for matching three keywords . . . 137
42 A `north` grammar for parsing `.` and `...` operators . . . 139
43 SEVM DFA for the triple dot grammar . . . 140
44 A simple `north` grammar . . . 142
45 Optimised medium-level intermediate representation for the `ABCD` grammar rule . . . 143
46 Optimised medium-level intermediate representation for the `ABCD` grammar rule (with partial reduction incorporation) . . . 144

47 Left-recursive `north` grammar test . . . 159
48 Right-recursive `north` grammar test . . . 159

LIST OF TABLES

1 Summary of the analysed parsing methods . . . 56
2 Summary of analysed tools and languages . . . 61

3 Terminal symbol sequence parsing example . . . 67
4 Non-terminal symbol sequence parsing example . . . 68
5 Basic source grammar compilation rules . . . 70
6 Regular operator compilation rules . . . 72
7 Updated non-terminal symbol compilation rules . . . 75
8 Fixed-length lookahead example . . . 76
9 Fixed-length lookahead compilation rules . . . 76
10 Variable-length lookahead compilation rules . . . 77
11 Extended grammar language elements and their compilation rules 78
12 Parsing fixed-length fields . . . 80
13 Rules for compiling capturing dynamic match grammar expressions . . . 81
14 Grammar rule for parsing and abstract syntax tree node construction of binary addition . . . 85
15 Rules for compiling grammars with delayed semantic actions . 89
16 The grammar rule and corresponding instruction sequence for binary addition when delayed semantic actions are used . . . 91
17 Additional grammar language elements to support reflective grammars . . . 94
18 Rule for compiling domain activation statements . . . 96
19 Rules for computing instruction closures . . . 102
20 Rules for computing instruction merger keys . . . 103
21 Subset construction example . . . 105

22 Reduction kind values . . . 120
23 Rules for computing scannerless Earley virtual machine ε-closures 127
24 Identifier-keyword disambiguation performance cost comparison 129

25 The median time needed to parse sample inputs . . . 155
26 The median time needed to parse `input_gcc_470k.i` with and without garbage collection in `north` . . . 156
27 The median time needed to parse `input_rust_650k.rs` with and without garbage collection in `north` . . . 156
28 The time needed to parse `input_gcc_470k.i` with and without reduction incorporation in `north` . . . 158

29 The time needed to parse `input_rust_650k.rs` with and without reduction incorporation in `north` 158
30 The benchmark results for parsing `input_a_1k.txt` with left and right-recursive grammars 160
31 The benchmark results for parsing `input_5a_10k.txt` with left and right-recursive grammars 160

GLOSSARY

abstract syntax tree It is a tree representation of the abstract syntactic structure of source code written in a programming language. 15, 29

deterministic finite automata extraction A scannerless Earley virtual machine optimisation, during which deterministic sequences of terminal symbol matching instructions are extracted into external deterministic finite automata, which are then invoked from the main parser. 22, 135

execution trace set An Earley virtual machine data structure that stores parser execution history to prevent duplicated input fragment parsing, infinite recursion, and exponential parsing complexity. 15, 65, 69, 98

lexer Also called a lexical analyser or tokeniser, it is a program that breaks down the input source code into a sequence of lexemes. 20, 30, 31, 36, 37, 55, 61, 62, 152, 153, 155

local grammar extension A grammar extension that is (temporarily) applied for a limited scope of input. 20, 37, 62, 150

medium-level intermediate representation Scannerless Earley virtual machine intermediate grammar representation that uses instruction sequences to represent grammars. 15, 108

reflectively extensible programming language A programming language whose syntax and semantics can be modified at compile time by providing syntactic and semantic extensions with the regular code. 16, 20, 27, 34, 165

scannerless parser It is a parser that performs tokenisation (breaking a stream of characters into words) and parsing (arranging words into phrases) in a single step, rather than breaking it up into a pipeline of a lexer followed by a parser. 20, 21, 23, 37, 151, 152, 164, 165

shared packed parse forest A graph representation of a parse forest (family of related parse trees) used in parsing with ambiguous/conjunctive grammars. 16, 82

ACRONYMS

APEG Adaptable parsing expression grammars. 44

API Application programming interface. 34

AST Abstract syntax tree. 29–31, 36, 48, 55, 58, 59, 63, 81–87, 90, 109, 148, 149, 153, 155

BNF Backus-Naur form. 30

CPU Central processing unit. 153, 161, 163

DFA Deterministic finite automaton. 24, 26, 52, 100, 126, 128, 130, 135, 138–140, 142, 149, 150, 165

ETS Execution trace set. 65, 104

EVM Earley virtual machine. 21, 25, 63–65, 68, 73, 76, 77, 80–86, 89–91, 93, 95–97, 99–101, 104, 105, 107–109, 111, 118, 121, 126–128, 131, 145, 146, 150

GLR Generalised LR. 22, 37, 39, 48, 106, 119, 155, 163

GSS Graph structured stack. 39

IQR Interquartile range. 155, 161

IR Intermediate representation. 31, 32, 131–135

JIT Just-in-time (compile/compiler). 149

LALR Lookahead LR (parser). 163

LR Left-to-right (parser). 37, 163

MIR Medium-level intermediate representation. 108–110, 118, 120, 122, 123, 125, 126, 131, 132, 134, 135, 138, 142–144, 157, 158, 165–167

NFA Nondeterministic finite automaton. 52, 100, 101, 126

PEG Parsing expression grammars. 44, 58

REP Reflectively extensible programming (language). 19–22, 25, 27, 34, 35, 37, 39, 46, 55, 56, 58, 59, 61, 91, 104, 150, 165

RIGLR Reduction incorporated generalised LR. 40

RNGLR Right-nulled generalised LR. 40

SEVM Scannerless Earley virtual machine. 19, 21, 22, 25, 26, 108, 109, 117–119, 121–127, 129–135, 139–142, 144–153, 156, 157, 160, 162–166

SGLR Scannerless generalised LR. 164, 165

SPPF Shared packed parse forest. 48, 82–84, 90, 148, 149, 155

1 INTRODUCTION

1.1 Importance and Motivation

Programming languages is one of the earliest topics of computer science. Over the last 70 years, this field arose from non-existence to the abundance and variety of programming languages we have today. This is arguably one of the most important topics of computer science even today. A good programming language enables the programmer to avoid mistakes while making the process of programming and maintaining existing projects easier and cheaper. That becomes even more apparent when considering the current direction of where computing is headed: bigger and more complicated systems, the Internet of things, and more distributed and parallel systems that are unlike anything seen in our history. Even the tiniest home appliance may have a microprocessor that runs a fragment of computer code written in one programming language or another.

Much like everything else relating to computer science, programming languages are subject to constant change and evolution. It is worthwhile to remember that even ideas that are taken for granted today, such as structural programming, procedures, and even variables, were a novelty at one point in computing history. However, updating an existing programming language to support new features is often a difficult and time-consuming endeavour.

One of the most popular and widely used programming languages, C++ has had four standards over the last two decades: C++98, C++11, C++14, and C++17 (not counting the upcoming C++20). Hundreds of people from all around the world participated in creating each of these four standards. Every proposal had to be submitted in a specific format and had to be reviewed by a committee comprising experts from various technology corporations, such as Microsoft, Google, and Red Hat. To many people, this process may appear daunting or off-putting, which could push potential participants away from developing a future version of this language. Although it may appear that it is certainly possible for a user to modify one of several existing open-source C++ compilers and implement personalised changes, practically, it is infeasible because advanced knowledge of poorly documented compiler internals is required to implement the desired changes. Furthermore, it is even more diffi-

cult to maintain these changes because compiler fixes and updates are created hourly for a language as large and complex as C++.

This is one of the reasons new programming languages are created every year. Many developers find it easier to develop a new programming language rather than to adapt an existing one to suit their particular needs. While this is not inherently an issue because specialised programming languages are often better suited to solve more particular problems, it does present a unique set of challenges.

With larger and more complex projects being created every day, it is not unusual for a project to use multiple programming languages at the same time. Sometimes snippets of one computer language are embedded into another. Even a website of moderate complexity may use five or more computer languages at the same time: HTML for data structuring, CSS for page appearance, JavaScript for defining client-side behaviours, Ruby for page generation, SQL for data lookup, and so on. Because of this, the aspect of integrating different computer languages becomes increasingly important.

Often, the code of one programming language is represented as character string literals in another. This is particularly common when using SQL from another general-purpose programming language to access databases. When processing text, regular expressions are used in a similar fashion. This approach of different integrations of computer languages is neither convenient nor error-proof because errors in an embedded language code can only be detected during runtime, and special symbols used in these 'second-class' languages often must be manually escaped.

However, what if one language could be properly embedded into another? What if some desired functionality could *be added* to a target programming language without having to modify the source code of the compiler? These questions are a few of the primary motivators for researching a class of programming languages called *extensible programming languages*. The core idea behind such languages is that the language designers and implementers will never be able to conceive of all the possible use cases of their programming language. As such, the extensible language or its implementation should provide a means for the user to adapt and extend it without having to understand every aspect of the language or its implementation and without having to modify the source code of the language compiler.

Depending on the level of extensibility provided, extensible languages allow users to define new linguistic features. Some of these features may contain completely new syntax and semantics that are not present in the base programming language. More powerful extensibility methods may even allow the definition of a new programming language within an existing one, thus enabling a composition of programming languages that was previously impossible.

Unfortunately, the topic of extensible programming languages is a fairly new one; thus, very few extensible programming languages exist. One of the reasons for this is the lack of suitable parsing algorithms for such languages. As a result, the focus of this work is the first part of implementing such a language: parsing. It is a process that gives structure and meaning to an otherwise seemingly random sequence of characters. Parsing them requires specialised parsing algorithms because extensible languages can change while they are being used.

In this thesis, the scannerless Earley virtual machine (SEVM) method is presented. It is a new parsing method that is scannerless,[1] can parse all context-free languages, and supports dynamically changing (adaptable) grammars while maintaining an acceptable parsing performance. Furthermore, the grammar definition language for the SEVM is designed with extensibility in mind and offers constructs that enable extending and reusing the existing grammars without requiring manual modifications to the original grammars. All of these features make the SEVM a perfect candidate for parsing extensible languages.

1.2 Problem Statement

The problem: virtual-machine-based scannerless parsing of reflectively extensible programming languages, where dynamically changing grammars with local grammar extensions are supported and grammars can be decomposed into several smaller grammars and their extensions.

Reflectively extensible programming (REP) languages are programming languages whose syntax and semantics can be extended dynamically during compilation without any compiler modifications. Because the grammars of such programming languages can change dynamically during parsing, spe-

[1] Scannerless parsers perform tokenisation (breaking a stream of characters into words) and parsing (arranging words into phrases) in a single step, rather than breaking it up into a pipeline of a lexer followed by a parser.

cialised parsing methods are needed, which must satisfy the following requirements (see Section 3.1.1) for more a detailed explanation):

1. support for dynamically changing grammars,
2. scannerless parsing,
3. unrestricted context-free grammar support,
4. support for local grammar extensions, and
5. reasonable performance.

In traditional parsing methods, grammars are internally represented using transition tables, which are used to encode the structure of a pushdown automaton. These tables are then used to drive the parsing process. However, such simple internal grammar representations are insufficient to encode the more complex actions needed to support dynamically changing grammars.

For the parsing method to support dynamically changing grammars during runtime, both the grammar definition language and its internal representation must contain the additional elements needed to manipulate the actively used grammar during parsing. Furthermore, to enable the grammar definition via a smaller grammar composition, the parsing method must not use a dedicated lexer (i.e. it must be scannerless). This further complicates the parsing process because the lexical ambiguities that are normally resolved during the lexical analysis must be eliminated within the parser. This further necessitates an additional grammar definition language and its internal representation extensions. As a result, a new internal grammar representation is proposed: instruction sequences, which are executed or interpreted by a virtual machine.

1.3 Research Goal and Objectives

The **research object** is the extensible programming language syntax analysis. The **research goal** is the creation of a syntax analysis method suitable for parsing reflectively extensible programming (REP) languages.

Objectives:

1. Definition of a grammar definition language,
2. Creation of a virtual machine that would be suitable for generalised context-free parsing with local grammar extensions,

3. Definition of the overall parsing method (SEVM),
4. Lexical analysis integration (scannerless parsing),
5. SEVM proof-of-concept implementation, and
6. SEVM performance evaluation.

1.4 Defended Claims

1. No existing parsing algorithm matches the criteria needed to implement a general REP language.
2. The Earley parser or its derivatives can be extended to support parsing REP languages.
3. The proposed SEVM parser offers acceptable parsing performance for practical use.

The requirements for an REP language parser are presented in Section 3.1.1. The parsing performance is considered acceptable if the parse time is within one order of magnitude of similar parser parse times. In the case of the SEVM, the parsing performance would be considered acceptable if it can parse input at a speed (within one order of magnitude) similar to other generalised context-free parsing methods.

1.5 Research Methods

The first claim is proved by performing a critical literature survey and using conceptual analysis methods. First, the review criteria are defined (Section 3.1.1). Then, all parsing methods that satisfy even some of the defined requirements are analysed. Because no parsing method that satisfies all our requirements is found, one parser family is chosen as a basis for constructing an improved parsing method.

In the second phase of this work, research by design is applied. Two new parsing methods based on the chosen parser family (Earley) are created. First, the Earley virtual machine (EVM) is constructed. The EVM is a parsing method that satisfies the first four REP language analysis requirements. Then, the flaws of the EVM are identified, which are then resolved in an improved version of the EVM – the scannerless Earley virtual machine (SEVM).

In the third and final phase of this research, a controlled experiment is conducted. The SEVM is implemented and its performance is compared to other parsing methods to demonstrate that the SEVM provides sufficient parsing performance for practical application. To perform this comparison, two research tools are implemented: `bench_parsers` and `north_cli`.

1.6 Results

The research results are the following:

- The new parsing method (SEVM) is suitable for REP language syntax analysis.
- The SEVM grammar definition language is not only suitable for defining real programming languages but REP languages as well.
- The `bench_parsers` research tool can be used to compare the performance of different parser implementations.
- The `north_cli` research tool can be used to analyse and inspect the internal state of the SEVM parser.

1.7 Scientific Contribution of the Research

- The SEVM is a virtual-machine-based parsing method. Whereas virtual-machine-based parsing methods have existed before, this is the first instance in which a virtual machine is used to parse grammars as complex as C or Rust. In parsing approaches based on virtual machines, grammars are internally represented by a low-level computer language. These grammars can then be subjected to domain-specific optimisation and transformation that would allow an increase in the parsing expressiveness or performance (for example, by inlining the grammar rule call targets).
- The deterministic finite automata extraction method that is used to speed up the SEVM grammars can be adapted to other parsing methods (in particular: Earley and generalized LR or GLR) to enable performance-wise cheaper (faster) token-level disambiguation.
- SEVM grammar definition language enables more flexible computer lan-

guage and their extension definition using grammar composition.

1.8 Practical Significance of the Results

The key practical result of this research is the SEVM parsing method, which has significant benefits over the existing parsing methods:

- Good parsing performance (as shown in Section 6.6.1).
- Generalised context-free parsing. This enables writing grammars more concisely because the grammar developers no longer need to abide by arbitrary parser limitations (such as no left recursion in recursive descent parsers, which makes defining the infix and postfix operators needlessly cumbersome).
- Because it is a scannerless parser, the entirety of the input grammar can be defined using a single language (unlike the commonly used LEX/YACC approach, where tokens are defined in one language and then grammars that use these tokens are defined using a separate language), thus further simplifying the grammar implementation.
- The SEVM grammar language allows defining grammars in a modular fashion. The base grammars can be extended by adding additional abstract rule implementations (e.g. by defining additional statement types and expression types separately from the main grammar). This would enable easier language extension development because users no longer need to rewrite the entirety of the grammars they are trying to extend. Such a grammar definition approach could be used already in compilers that support procedural macros (e.g. the Rust programming language allows implementing macros in external modules, which could use the SEVM to parse their input, which then can be transformed into a valid Rust code).
- The SEVM parser supports dynamically changing grammars. Using the SEVM in a new programming language (or an existing one) would at the very least allow more flexible macro systems, where the syntax of each macro can be defined by the user. More importantly, the SEVM is one of the few available parsing methods that can be used to implement extensible programming languages.

- Using virtual machines for grammar representation offers an additional benefit. It is possible to include general-purpose computations within grammar byte code, which enables one to drive the parsing process manually using user-written procedural code, thus further extending the recognised grammar class.

Some parts or ideas of the SEVM may be used independently of the SEVM itself:

- The DFA extraction method may be used to speed up (or expand the recognised classes of grammars) the existing parsing algorithms by allowing a simpler token-level disambiguation scheme.
- Using virtual-machine instructions to represent grammars internally allows the use of domain-specific optimisation to further optimise grammars (possibly even by mixing different parsing algorithms and selecting the one that is most appropriate for each situation).
- The SEVM implementation proves that it is feasible to use just-in-time compilers to transform grammars into native machine code for increased parsing performance.

1.9 Approbation

The results of this dissertation were presented at the following international conferences:

- FedCSIS 2017, 6th Workshop on Advances in Programming Languages (WAPL'17), Prague, Czech Republic, 2017.09.03–07.
- FCSIT 2019, European Conference on Frontiers of Computer Science and Information Technology, Amsterdam, Netherlands, 2019.09.22–24.

1.10 Publications

The main results of this dissertation were published in the following papers:

- Šaikūnas A. (2017). Critical Analysis of Extensible Parsing Tools and Techniques. Baltic J. Modern Computing, Vol. 5 (2017), No. 1, pp. 136–145.

- Šaikūnas A. (2019). Parsing with Scannerless Earley Virtual Machines. Baltic J. Modern Computing, Vol. 7 (2019), No. 2, pp. 171–189.

Other papers:

- Šaikūnas A. (2017). Parsing with Earley Virtual Machines. Communication Papers of the 2017 Federated Conference on Computer Science and Information Systems, Vol. 13 (2017), pp. 165–173.
- Šaikūnas A. (2019). Just-in-time Parsing with Scannerless Earley Virtual Machines (*accepted for publishing*).

1.11 Outline

This thesis is split into seven main chapters:

- In Chapter 1, the research context is given: the introduction, motivation, research goal, and so on.
- In Chapter 2, the basic concepts for understanding this work are presented, including the definition of the REP term.
- In Chapter 3, the current state of art is provided. First, the requirements for an REP language parser are formulated. Then, various existing parsing methods and related tools are analysed to find the closest one capable of parsing an REP language. After an exhaustive search, we conclude that no single parsing method satisfies our requirements; however, two main candidates (the Earley and Yakker parsers) are found that can be used as a basis for building a more suitable parsing method.
- In Chapter 4, we incrementally construct a new parsing method, EVM, that is based on the Earley/Yakker parsers and that satisfies our requirements for parsing REP languages. Furthermore, the EVM grammar language for defining new (extensible) languages, EVM optimisations, and other considerations are presented in this chapter.
- In Chapter 5, we use the observations made by testing the EVM prototype and the knowledge learned by constructing the original EVM to create a successor to EVM called SEVM. One of the primary goals of the SEVM is practical parsing performance (while maintaining the previous requirements for parsing REP languages). To achieve this per-

formance, additional optimisations and changes are implemented (most notably, DFA extraction, token-level disambiguation, and just-in-time grammar compilation) that enable the SEVM to achieve this practical performance.

- In Chapter 6, we evaluate an implementation of the SEVM parser called `north`. The performance influence of various described and implemented optimisations is measured, and the SEVM parser implementation is compared with other parsing methods. Additionally, arguments for the internal and external validity of the achieved results are given as well.
- Finally, in Chapter 7, the general conclusions of this thesis are presented.

2 PREREQUISITES

In this chapter, we present the basic concepts required to understand the rest of this work. Additionally, the term *reflectively extensible programming (REP) language* is introduced in this chapter.

2.1 Compilers and Programming Languages

In the early days of computers, computer programs were written in machine languages that directly corresponded to the underlying hardware. Such programs were read from punched cards, magnetic tape, or even physical hardware switches that encoded the underlying program in binary form. Such languages are now called first-generation programming languages.

As computers became more advanced, so did the programming languages that were used to program them. Because writing programs in binary (or similar) form was difficult and error-prone, the idea of a compiler was introduced. A compiler is a program that reads another program written in a human-readable text form and produces the corresponding binary code that can be executed by the computer hardware. This was the principle behind the second-generation programming languages. Languages from this generation still closely mimicked the underlying computer architecture and were called assembly languages. Assembly languages primarily consist of instructions that directly manipulate computer processor registers, memory, and other devices. Most of the assembly instructions, when compiled, are directly translated into corresponding binary code. Because of this one-to-one correspondence from assembly instruction to binary code, assembly languages are used even today in compiler code generators and to visualise binary code in a more readable form.

As computer memory and processing power became more abundant, computer programs became larger and more complex, and as such, it became much more difficult to write and maintain these kinds of programs. Additionally, competing computer architectures, each with their own assembly languages, emerged. To make a program that was written in a second-generation programming language work on a different computer architecture, such a program had to be rewritten in a different assembly language, often from scratch. To solve these issues, third-generation programming languages were created. Lan-

guages from this generation often mimicked mathematical notation and provided features that did not exist in underlying computer hardware. Variables, subroutines, and data structures are all abstract elements that had no direct-to-hardware correspondents. Instead, they existed only as abstract constructs in third-generation programming languages that would essentially disappear into a sea of instructions when compiled into binary programs.

Almost all programming languages that are in use today are third-generation programming languages. However, not all languages that are currently in use are classified as programming languages. As computers became more capable of solving increasingly advanced problems, new computer languages were created to ease the solution of these problems. For example, the computer language CSS is used to describe the appearance of webpages. Another language called JSON is used to structure, transfer, and store arbitrary information that can then be manipulated within other programming languages. Yet another language, Markdown, is used to describe formatted text. It is not uncommon for one or several specialised languages (such as JSON, XML, etc.) to be used within a program or system written in one or more programming languages. Specialised languages that are used to encode or describe information concerning a specific problem or domain are called *domain-specific languages*. Most programming languages can be split into two categories: interpreted programming languages and compiled programming languages.

Compiled programming languages, as the name implies, are translated from human-readable text into low-level machine or binary code. Most early third-generation programming languages were compiled into machine code. However, later, new languages appeared that were compiled into a kind of binary code called *byte code*, which was either interpreted or further translated into machine code within a program called a virtual machine. Java and C# are two well-known examples of such languages. The primary advantage that these languages provide is portability, the ability to run the same program across different computer architectures or platforms without having to recompile the program itself.

Interpreted programming languages, on the other hand, instead of compiling the source code of a program into binary or machine code, attempt to directly execute it. A program that executes or *interprets* such programs is called *an interpreter*. Some interpreters still use compilation for byte code or ma-

chine code to execute the source programs, but this process is hidden from the user. Interpreted programming languages also tend to provide additional features (such as the possibility of dynamic typing) that are not found in traditional compiled programming languages. This and the lack of a separate compilation step makes it faster to develop programs written in interpreted programming languages, although often with a significant runtime performance penalty because interpreted programs tend to be much slower compared to their compiled alternatives.

2.2 Compiler Architecture

Many new programming languages have been created over the years. Wikipedia (as of the time of writing this) lists at least 700 that are publicly available. Many of these languages have several compilers or implementations. Naturally, during the development of these languages, some common patterns have emerged. Usually, a compiler of a programming language is split into four major parts:

- *A lexer* is a compiler component that converts the source code of a program (a string of characters) into a sequence of *tokens*. This process is called *tokenisation*. A token is essentially a word of a programming language. Common token types include identifiers, numeric constants, string constants, and operators. Lexers are also responsible for removing comments and whitespace (all characters that do not represent a visible symbol, such as spaces, tabs, etc.) from the analysed source code.

- *The parser* takes the sequence of tokens and produces an *abstract syntax tree* (AST). As the name suggests, an AST is a tree that structurally represents the current program. Each node within this tree corresponds to a basic element of the language within the input program. Common examples of AST nodes are nodes that represent constants, variables, function calls, declarations, and so on.

- *A semantic analyser* is primarily used in preparation for code generation. It adds enough information to the AST to be suitable for code generation. Commonly, during semantic analysis, variable references and function calls are resolved and types are checked (in statically typed programming languages). In addition, this is where semantic errors are detected (such

as attempts to use a variable or a function that does not exist).

- *A code generator* traverses the final AST and, if no errors were found, produces the corresponding binary code for the source code that is being compiled.

2.3 Lexing and Parsing

Formal languages are defined using formal grammars. A formal grammar primarily consists of production rules for strings in an input language. While formal grammars are widely used in formal language theory to analyse formal languages and algorithms that deal with formal languages, they are not convenient enough to define computer languages in practice. Instead, special *grammar languages* are used to define computer languages, which resemble formal grammars but provide additional features that make language definition easier.

Many of the existing parsing (and lexing) algorithms during analysis rely on special data derived from the language grammar. Because writing lexers and parsers (and creating the respective parser data) manually is difficult and error-prone, parser (and lexer) generators are used to generate the source code of a parser (or a lexer). Parser/lexer generators read the language grammar specified in a grammar language and produce the source code used to compile the resulting parser/lexer. Lexer and parser grammar languages are generally distinct. Lexer tokens are typically defined using regular expressions, whereas parser productions are defined using a variant of the BNF language.

Even though most computer languages are analysed using both a lexer and a parser, a dedicated lexer is not really required, as tokens used within a lexer can be expressed in terms of parser productions. Most lexers usually run in linear time and can recognise only regular languages, which is enough to define the tokens of most languages. Parsers generally recognise a subset of context-free languages but often have much higher algorithmic complexity. This is one of the reasons the distinction between a lexer and parser exists. By offloading some of the more trivial syntactic analysis steps to a lexer, the overall parsing and compiler performance is increased.

However, in more modern times with the advent of more powerful computers, new parsing algorithms have emerged that no longer require a separate lexer step, and instead, tokenisation is merged into the parser. Such parsers are

called *scannerless*. Removal of the lexer reduces the overall compiler complexity and increases the variety of possible input grammars at the cost of reduced parsing performance.

Most of the existing parsing algorithms can be divided into the following three categories:

- *Top-down parsers* attempt to perform input recognition from the top of the parse tree by rewriting the rules of the language grammar. Essentially, the goal of a top-down parser is to find a sequence of rewrite rule applications, which starts with a single non-terminal symbol that represents the whole program and ends with the terminal symbol sequence that represents the initial source code.
- *Bottom-up parsers* work in the opposite way compared to top-down parsers. The algorithms start with a sequence of terminal symbols that represent the initial source code and attempt to merge (or *reduce*) a sub-sequence of these symbols into a single non-terminal symbol. The process repeats until there is only a single non-terminal symbol left that represents the whole program.
- *Hybrid parsers* try to combine both of these approaches.

In other words, one of the jobs of a parser is to recognise whether or not the input source code matches the specified grammar. However, to use such a parser within a compiler, it also needs to construct the AST for the parsed input. Parsers that do not construct the AST and only perform recognition are called *recognisers*.

2.4 Code Generation

Each compiler has at least one target architecture for which it generates byte code or machine code. Early compilers were very specialised and supported only a single target architecture. However, with multiple competing processor architectures becoming prominent, rewriting the whole compiler just to support some new processor architecture or even rewriting the code generator portion of the compiler was becoming increasingly difficult. To make it easier to port compilers to new platforms, *intermediate representation* (IR) languages were created.

Instead of supporting multiple different architectures, compilers would only

have to support a single low-level intermediate language to which all of the source code would be compiled. Then, the code generator for a specific target would translate the generated intermediate code into the final machine code. The use of the IR code not only eases the design and porting of compilers but also makes it easier to perform optimisations of the compiled code. Currently, the most prominent IR language/code generation framework is the LLVM toolchain,[1] which provides a custom IR language and libraries that allow the optimisation of this code, the generation of equivalent machine code, the provision of debugging tools, etc. Many new languages and compilers are based on this toolchain, such as the new C++ compiler, Clang, Apple's general-purpose language Swift, Mozilla's systems programming language Rust, and others.

2.5 Extensibility

Most of the existing programming languages have fixed syntax and semantics. The syntax of a programming language is usually defined using grammar languages when generating a parser for that language compiler, and semantics are expressed as arbitrary code that performs checks and transformations on the AST of a compiled program. This limitation of having a fixed syntax and semantics was understood even in the early days of third-generation programming languages. Attempting to use a language that is ill-equipped to solve a specific problem usually results in code that is sometimes trivial but repeating and difficult to modify and maintain, which is sometimes referred to as *boilerplate* code. To avoid having to manually write boilerplate code, several solutions are used in practice:

- **Specialised/domain-specific languages**. General-purpose languages, such as C/C++, are ill-suited for defining computer-language grammars. This is why grammar languages and parser generators are used to create lexers and parsers. String pattern matching is difficult to perform in general-purpose languages as well. Therefore, to ease this task, instead, regular expressions are used that make it easier to define the structure of a string pattern that is being searched/matched. Query languages, such as SQL, make it more convenient to access and extract specific information from databases.

[1] LLVM compiler infrastructure homepage: https://llvm.org/

- **Macros** allow the definition of rule (or pattern) and replacement pairs, which are then applied to compiled source code. When a compiler detects macro invocation by finding matching patterns, it replaces (expands) the found code sequence into the appropriate body of a macro. This allows reducing the amount of repeating code in the source files, as commonly used patterns can be defined as macros. Multiple variations of macrosystems exist; for example, the C programming language performs macro substitution only on a textual level, while the Rust programming language allows defining patterns that operate directly on the AST of the compiled program.

- **Templates** can be considered a more advanced version of macros, which also have access to the type information. They allow a parametrised definition of various language objects (such as functions and structures), which then can be instantiated by invoking a template and providing parameter values, which are inserted into the original object definition. This way, each template invocation may result in the creation of a new language object, which, in a language without macros or templates, must be defined manually. Templates allow further reduction of code duplication but often result in additional code complexity.

- **Metaprogramming** is a method that allows treating computer programs as their data. Multiple types of metaprogramming exist, but in this work, we refer to metaprogramming as an ability of a code fragment (*metaprogram*) to write a new program. Some languages, such as Haskell, have built-in support for compile-time metaprogramming, which allows programmers to invoke metaprograms that generate parts of the program that are currently being compiled. Many scripting languages provide a function called `eval()`, which allows the dynamic evaluation of the language code in the provided text strings. This way, a metaprogram can construct a code fragment of a program in a string and then pass it to the `eval()` function, which then could include the provided code fragment in the currently running program. This approach to metaprogramming is also referred to as *generative programming*.

- **Compiler plugins** allow even more free-form changes to the language syntax and semantics. Compilers that support compiler plugins typically

provide an API that could be used to implement these plugins. The power and flexibility of a plugin are directly determined by the API, which differs on a compiler-to-compiler basis.

Next, there are *extensible programming languages*. Some early third-generation languages were considered to be extensible if they supported even one of the previously listed features (most notably macros). Even languages that supported procedures at one time were considered to be extensible when procedural programming was a novelty. Standish [27] provided one of the first definitions of an extensible programming language; an extensible language simply allows users to define new language features. However, such a general definition is not that useful, and several new terms have been created to differentiate between languages with varying degrees of extensibility.

Extensible syntax programming languages are languages that allow their syntax to be extended, often using a specialised grammar language. Languages that allow their syntax extensions to be specified within the normal code and inside external files/plugins both fall into this category. However, in this work, we primarily focus on the former type of languages. To further differentiate between these two types of languages, we introduce a new term: *reflectively extensible programming languages*. Reflectively extensible programming (REP) languages are languages whose syntax and semantics can be modified at compile time by providing syntactic and semantic extensions with the regular code.

3 STATE OF THE ART

3.1 Parsing Methods

In this chapter, we investigate various existing parsing methods to determine whether any can be used or extended to parse REP languages.

3.1.1 Requirements for a reflectively extensible programming language parser

It is fairly obvious that an REP language requires a specialised parser. First, an REP language parser must support mutable grammars. This requirement arises from our definition of REP languages. Theoretically, it is possible to adapt any existing parsing method to support partially mutable grammars using the algorithm displayed in Fig. 1. This algorithm simply divides the input source code into blocks and then uses a separate parser to parse each block. In practice, however, there are several challenges to using this algorithm:

- **Poor grammar mutation performance.** Many parsing algorithms rely on data that are derived from the original language grammar. For example, LR parsers use transition tables that are generated from the initial grammar productions. In most cases, this table generation is performed by parser generators; however, it is possible to embed the algorithm that computes the required parser data into the REP language parser itself. However, this means that every time the language grammar is updated, all of the parser data must be regenerated. Even trivial syntactic additions to the initial language would result in having to re-analyse the entire input grammar. Furthermore, several successive grammar modifications even in the same source file would result in an equal number of parser data regenerations. This would make adding syntactic extensions to the base language prohibitively computationally expensive and thus would defeat the purpose of using an REP language.
- **Clear block boundaries.** For the naive parsing algorithm to transition from one grammar to the next, it must be able to identify where the scope of the first grammar ends and where the scope of an updated grammar begins. In other words, there must be a clear and unambiguous boundary

```
1. Let G0 be the initial grammar and A0 the respective
   parser data (e.g., such as transition tables used in LR
    parsers).
2. Divide input source into n top level blocks B0 − Bn−1 (
   such as top level declarations in C/C++).
3. Parse and semantically analyse Bi with current parser
   data Ai. If the current block contains a new syntactic
   extension, then produce a new grammar composition Gi+1
   based on Gi and the extension. Update the new parser
   data Ai+1 based on grammar Gi+1.
4. Parse the subsequent block Bi+1 using parser data Ai+1.
5. Repeat steps 2–4 until completion.
```

Figure 1: Naive extensible parsing algorithm

between the original and updated language segments within the initial source code.

- **Limited support for scoped grammar mutations.** In some cases, it may be necessary to enable a syntactic extension only for a limited portion of the AST. For example, a user may wish to enable a specific grammar extension only for the next statement within the initial program. Such a grammar mutation would be impossible in the native extensible parsing algorithm because it only allows grammar modifications between top-level AST nodes.

- **Limited local ambiguity support.** In the event that the chosen base parsing algorithm supports ambiguities within the selected language, all of these ambiguities would need to be resolved before the current top-level block terminates. This requirement arises from the fact that every top-level block could be parsed with an updated grammar; therefore, the internal structure of the original parser that represents the ambiguity would be lost when transitioning from one grammar to the next.

Second, we wish for the parsing algorithm to support scannerless parsing. The elimination of a dedicated lexer allows the use of a single unified language to define both tokens and regular grammar productions. This makes it easier and more concise to specify new syntactic extensions.

While having a separate lexer does have some advantages, the primary of which is increased performance, the introduction of syntactic extensions with lexical ambiguities means that all of the ambiguities must be propagated to-

wards the parser, which must be specially modified to support such ambiguous tokens. This would result in increased lexer and parser performance.

Third, we require the REP language parsing algorithm to support unrestricted context-free grammars. One of the primary reasons for restricting the allowed input grammars is, yet again, increased performance. Generalised parsing algorithms, such as the GLR and Earley algorithms, are too slow to be used practically. However, with improvements to computer hardware and further refinements of parsing algorithms, we believe that the historical performance motivations for restricting allowed input grammars no longer apply. Additionally, the users of the REP language may not be parser experts, and they should not be forced to understand the inner workings of the parsing algorithm just so they can write a syntactic extension.

To summarise, we propose the following requirements for an REP language parser:

1. support for dynamically changing grammars,
2. scannerless parsing,
3. unrestricted context-free grammar support,
4. support for local grammar extensions, and
5. reasonable performance.

3.1.2 The LR(k) parsers

The LR(k) is a family of table-based, bottom-up parsers. It is one of the earlier parsing algorithms and is indirectly still widely used even today. It was first described by [18]. It runs in linear time when parsing deterministic context-free languages.

The algorithm starts the parse in an initial state. Then, it reads a single symbol from the input and looks up the appropriate action from the action table AT_s. Three possible actions can be taken:

- **The shift** action, denoted by $S(n)$, indicates that the current symbol a must be pushed onto the stack, a new symbol must be read from the input, and the parser must move to state n.
- **The reduce** action, denoted by $R(r)$, indicates that the reduction based on grammar rule r must be performed. If the rule is denoted by $lhs \leftarrow rhs$,

```
a ← read_sym()
s ← 0
loop
    action ← AT_s(a)
    if S(s_1) ← action then                                    ▷ Shift
        push(a)
        s ← s_1
        a ← read_sym()
    else if R(r) ← action then                                 ▷ Reduce
        (lhs → rhs) ← rule_r
        pop(sizeof(rhs))
        push(lhs)
        s ← GT_s(lhs)
    else if A() ← action then                                  ▷ Accept
        return
    else
        error( )
    end if
end loop
```

Figure 2: The LR(0) parser algorithm

then the top $|rhs|$ stack symbols must be replaced with a single non-terminal product lhs. Additionally, the parser must move to a state indicated by $GT_s(lhs)$, where GT_s is the go-to table for state s.

- **The accept** action, denoted by $A()$, indicates that the input has been successfully recognised and that the parsing algorithm must terminate.

The tables AT_s and GT_s used in the parsing algorithm can be constructed from the DFA built from the initial grammar. An algorithm allows dynamically growing or shrinking these tables, as described in [7], thus making it possible to mutate the grammar that is used during parsing.

The letter k in LR(k) determines how many symbols the algorithm can lookahead before deciding on which action to take. The LR(0) parsers perform no lookahead and select the action to be taken immediately based on the current input symbol (see Fig. 2). In most cases, this makes LR(0) practically inapplicable because the algorithm cannot distinguish input `x + y` from `x` (when + is right-associative) because the parser must lookahead a single symbol to determine which action to take (reduce `x` as an expression or attempt to read the next symbol). This situation when a single action table cell has both a shift and a reduce action is called a *shift/reduce conflict*.

Therefore, in practice, LR(1) or lookahead LR(1) (LALR(1)) parsers are

used instead. The LALR(1) is a modified version of LR(1) that accepts a smaller class of grammars compared to LR(1) but uses significantly smaller parse tables. Because the size of the parse tables increases exponentially with k, any value higher than 1 is generally not used. Even though $k = 1$ for some existing languages is enough, plenty of languages require higher or even unbounded k values.

Another limitation to LR(k) parsers is that they cannot be used for scannerless parsing. The LR(k) parser cannot differentiate conditional `if(a)` from a function call `if(a)` because no simple way exists to reject identifiers that overlap with reserved keywords. As a result, if scannerless parsing is required, more general parsing algorithms (such as GLR) are used. Because of the fixed lookahead and inability to use the algorithm for scannerless parsing, LR(k) does not meet our criteria for REP language parsing.

3.1.3 The generalised LR family of parsers

The generalised LR (GLR) parser is an extension to the LR parser that allows parsing most non-nullable context-free languages. Masaru Tomita [29] first described it and intended to use it for natural language parsing, but since then, it was also adapted and used to parse computer languages.

The GLR parsers share most of the key ideas with LR parsers. They are still based on tables, which are used to select an action to be performed based on the current input symbol. Tables also contain shift, reduce, and accept actions. In addition, tables for GLR parsers are generated almost exactly the same as LR(0) tables.

The primary difference between LR(k) and GLR parsers is how they treat conflicts. A single GLR action table item may contain a single shift and several reduce actions, all of which are executed when an appropriate symbol is found. This means that the GLR parser is no longer deterministic and may be in multiple states at once. Additionally, GLR parsers use a graph-structured stack (GSS) instead of a regular stack to represent alternative parse paths. Because of this change, the parser no longer requires a lookahead to operate correctly (even though LR(1)/LALR(1) parse tables with conflicts may still be used to reduce parsing ambiguity for performance reasons). This also means that the parser is now suitable for scannerless parsing.

Unfortunately, the original GLR algorithm contains a flaw that prevents the

```
E → F + E | F
F → I * F | I
I → a
```

Figure 3: A grammar for a language that supports $+$, $*$ operators and the variable a

algorithm from terminating when the initial grammar contains hidden left recursion. This was discovered by [24], and a modification of the GLR algorithm was proposed (called the RNGLR), which correctly handles the hidden left recursion and supports more effective handling of ε-reductions. However, the modified parser uses a different variation of LR tables, which means that the incremental LR table generation approach that was described by [7] is no longer directly applicable to RNGLR parsing. Thus, to support the mutable grammars that are required for REP language parsing, the algorithm for incremental LR table generation would need to be modified first to allow the dynamic generation of right-nulled parse tables.

The authors of the RNGLR parser also presented an even more radical modification of the RNGLR called the reduction incorporated GLR (RIGLR) [23], which incorporates additional information into the RNGLR parser that reduces the parser stack activity to further boost parsing performance. However, in our opinion, the performance gains observed while testing the RIGLR do not warrant a significant increase in the parse table size. As such, the only viable candidate from the LR/GLR parser family for REP language parsing is the RNGLR parser.

3.1.4 Recursive descent parser

The recursive descent parser [6] is a top-down parser. The parser for a specific grammar is split into several mutually recursive functions, where each function parses one non-terminal symbol from the grammar. To parse the whole input, a function corresponding to the initial grammar symbol is invoked, which in turn calls other functions that correspond to other non-terminal symbols, which consume terminal symbols from the input upon encountering them.

See Fig. 3 for an example grammar and Fig. 4 for the corresponding recursive descent parser implementation. The function `accept(a)` used by the parser consumes the next non-terminal symbol if it matches the provided sym-

```
def E()
  F()
  if accept('+'); E(); end
end

def F()
  I()
  if accept('*'); F(); end
end

def I()
  expect('0')
end
```

Figure 4: A corresponding Ruby program that parses the provided grammar with the help of the `accept()` and `expect()` functions

bol a. The function `expect(b)` consumes the next input symbol and fails if it does not match the provided symbol b.

Because of the simplistic nature of the implementation, the recursive descent parsers are often implemented manually without resorting to using a parser generator. The provided parser example is called a *predictive* recursive descent parser because it does not use backtracking; thus, it executes in linear time. However, in cases of more complex grammars, the use of backtracking may be necessary and would result in an exponential execution time because some fragments of the code may be analysed multiple times with the same function.

It is also possible to use the backtracking recursive descent parser to implement mutable grammars. This can be achieved using a single parsing function with three arguments: first for the current grammar, second for the current parsing position, and third to indicate the non-terminal symbol that must be parsed next. A positive return value of this function would indicate the current parsing position after the provided symbol has been parsed. The parsing starts by calling the function with the initial grammar and the initial grammar symbol at position zero. Then, this function calls itself to parse other non-terminals from the current grammar, while consuming all terminal symbols. Failure to consume an expected terminal symbol leads to backtracking.

Such a parsing algorithm is not only easy to implement but also allows parsing languages where grammar extensions are applied to specific scopes. Furthermore, it can be used without a dedicated lexer and can operate directly with

characters from the initial source code, thus fulfilling two out of four requirements for an REP language parser. Unfortunately, that is where the advantages of the recursive descent parser end.

Backtracking, as mentioned before, leads to exponential performance. Furthermore, because the parser is implemented as a series of mutually recursive functions (or a single function in the mutable grammar case), it does not support left recursion. Attempting to use left recursion in grammars would cause infinite recursion in the corresponding parse functions, eventually exhausting the stack memory and thus terminating the parser program.

Another issue is ambiguous grammars. With the current parsing algorithm, it is impossible to support ambiguous parses because each function for the corresponding non-terminal must always terminate after a fixed amount of input characters (even if that number is 0). However, with ambiguous grammars that may not be the case because, depending on the selected production rule alternative, the parse for the specified non-terminal symbol may terminate at differing positions.

As a result, the recursive descent parser, even with mutable grammar support, is not applicable for implementing the REP language because it does not provide reasonable performance and restricts the allowed class of grammars too severely.

3.1.5 Packrat parser

The primary problem that all context-free language parsing algorithms attempt to solve is the production rule selection. Context-free languages are defined using context-free grammars, which are composed of production rules. Depending on the grammar, some non-terminals may have multiple production rules. Selecting the correct one (or multiple ones in the case of ambiguous grammars) is the key context-free language parsing problem. Different parsing algorithms attempt to solve it differently. The LR algorithms build a table that lists all possible terminal symbols that may be encountered at any given moment and use it to perform reductions. The backtracking recursive descent parser tries each production in succession, essentially brute-forcing the possible solution. Moreover, the Earley parser tries to mix those two approaches together.

However, it has been noted that using context-free grammars to define com-

puter languages may not be the most intuitive way to do it. Therefore, a new grammar definition formalism was created called parsing expression grammars (PEGs) [12]. The PEGs eliminate the source of context-free grammar ambiguity by replacing the choice operator | with an *ordered choice operator* $\backslash$. In the production $A \leftarrow B\backslash C\backslash D$, this indicates that the non-terminal B is matched first. If B matches successfully, then the remaining production alternatives are ignored. Otherwise, C is matched next.

It is fairly easy to spot the correspondence between the ordered choice operator and the way the recursive descent parser works. As a result, backtracking recursive descent parsers can be used to parse all PEGs. However, a modification of the recursive descent parser called the packrat parser allows parsing PEGs in linear time, which is one of the reasons PEGs have become so popular in recent years.

The packrat parser [11] is a modification of the recursive descent parser that memorises all intermediate results of non-terminal parser functions. Because of this, the same source location using the same parsing rule may be parsed only once in the packrat parser, which is one of the reasons the algorithm runs in linear time.

Unfortunately, a simple implementation of the parser and good performance comes at a price:

- No left-recursion support. Because the packrat parser is essentially a recursive descent parser, left-recursive rules (including indirect or transitive left recursion) would cause infinite recursion.
- High memory usage. Because the packrat parser must memorise all intermediate parsing results, this causes fairly high memory usage. Some variations of the parser do exist that attempt to optimise the memory management of the packrat parser, such as [13].
- No ambiguity support. The PEGs are unambiguous by definition, and it is impossible to represent ambiguous languages using them. As a result, parsing a language such as C++ using just PEGs is impossible.
- No true grammar union. Consider grammars G_0 and G_1 displayed in Fig. 5. Both grammars are valid and describe their respective languages correctly. What happens when both grammars are combined into one? Depending on the order of the union, we obtain different results. If we

G_0:
```
  E → "if" E "then" E "else" E
    / "if" E "then" E
```

G_1:
```
  E → "if" E "then" E
```

$G_2 = G_0 \cup G_1$:
```
  E → "if" E "then" E "else" E
    / "if" E "then" E
    / "if" E "then" E
```

$G_3 = G_1 \cup G_0$:
```
  E → "if" E "then" E
    / "if" E "then" E "else" E
    / "if" E "then" E
```

Figure 5: Two example parsing expression grammars that define conditional expressions and their unions

combine G_0 and G_1, then the resulting grammar G_2 is identical to G_0, as the newly appended rule is a duplicate of an existing one in G_0. However, if G_1 is combined with G_0, then we obtain G_3, which breaks all if-else conditionals in the G_0 language because the newly prepended rule from G_1 will consume all the input and thus the `'else' E` part will never obtain a match. This issue is explored in more detail in [17]. This means that the extension designer must be aware of all existing definitions of the target non-terminal and upon extension must correctly specify the order in which the existing and new non-terminal production rules are to be applied. Failure to do so may result in breakage of the base language grammar.

Because of these issues, we find that PEGs and the packrat parser are insufficient for implementing the REP language parser.

3.1.6 Adaptable parsing expression grammars

Adaptable parsing expression grammars (APEG) [22] comprise an extension to the PEG that allows parsing mutable or *adaptable* grammars. These are grammars whose production rules can be added, removed, or modified mid-parse. As a result, such grammars can be used to specify the syntax of extensible languages. Furthermore, APEGs contain additional extensions that enable

the specification of context-dependent constraints:

- Binding expressions that allow saving context-dependent information during parsing.
- Updated expressions that allow updating existing attribute bindings.
- Constraint expressions that use other attribute bindings to restrict specific parse paths.

The article's authors present an interesting approach regarding the APEG parser implementation [22]. First, a modified packrat parser is generated that is capable of parsing the base grammar of the language. This generated parser also contains hooks that can be used to invoke dynamically defined rules during parsing. This mixture of a statically generated base language and dynamically interpreted grammar extensions allows the parser to very efficiently recognise the base language while also recognising the extensions to the base language at a somewhat reduced performance. Because of this, such a parsing model is applicable to languages that contain a fairly large base language and possibly several smaller language extensions. However, it is not so well suited where the base language is minimal and where the majority of the language is defined through extensions that (possibly) reside in external libraries or modules.

Although the APEG parser presents a viable option for implementing an REP language, the APEG model is still based on the original PEGs and thus inherits most of its restrictions, namely the following:

- no left-recursion support,
- no support for ambiguous languages, and
- no true grammar union.

3.1.7 Specificity parser

The metafront system [3] is a tool for program transformation that also supports extensible parsing. It employs a novel method for parsing, called a specificity parser. The specificity parser is a scannerless top-down parser. At any point during analysis, the parser keeps track of the remainder of the input that has not been parsed yet and a set of candidates, which are remainders (tails) of the production rules. The parsing process is divided into challenge rounds, and during each round, the most lexically specific candidate is selected and used to

advance the parser, which consumes some of the input and the matching parts of the current production tails. The process is repeated until the remaining string is empty.

Because the current candidate production tail set is maintained at any given moment during parsing, this information allows generating informative error messages in the event of a parsing error. This parsing method also prohibits ambiguities, which are resolved whenever new productions are added. If an ambiguity is found during the declaration of a new production that depends on an existing one, an error is generated forcing the user to adjust the newly defined grammar. While this method of handling ambiguities is convenient for defining fully deterministic languages, not all languages (and thus possible language extensions) that are used in practice are context-free and deterministic. As a result, the inability to support ambiguous grammars is a shortcoming.

Furthermore, the specificity parser has additional difficulties when parsing binary operators with matching prefixes, but with different precedences, such as C++'s logic `&&` and binary `&`. For example, the parser fails to recognise input `x && y`. Because the operator `&` has higher precedence than `&&`, it is parsed first and consumes the `&` symbol from the input, leaving `& y`, which then fails to parse. To resolve this issue, the parser's authors introduced a special form of lookahead called traps, which are then used to restrict the parsing of the operator `&` by ensuring that it is not followed by an additional `&` terminal symbol. This issue becomes even more relevant when considering the scenario in which a lower-precedence operator `&&` is added in an extension. Then, to ensure that this operator parses correctly, the original rule for the binary operator `&` would have to be modified with an appropriate trap. Because of the lack of ambiguous language parsing support, the requirement for a mandatory lookahead in certain situations, and no left-recursion support, we do not believe that this parsing method is a viable candidate for REP language parsing.

3.1.8 Earley parser

The Earley parser [8] is a top-down chart parser. The original algorithm can parse all non-nullable context-free grammars. A modified version of the Earley parser supports nullable grammars as well, but with reduced performance.

The parsing algorithm has two inputs: the source code that is meant to be parsed and a list of grammar production rules G used for parsing. Unlike most

```
for all input symbols a do
    for all (X → α • Yβ, j) ∈ S(k) do                          ▷ Prediction
        for all (Y → γ) ∈ G do
            add (Y → •γ, k) to S(k)
        end for
    end for
    for all (X → α • aβ, j) ∈ S(k) do                          ▷ Scanning
        add (X → αa • β, j) to S(k + 1)
    end for
    for all (X → γ•, j) ∈ S(k) do                              ▷ Completion
        for all (Y → α • Xβ, i) ∈ S(j) do
            add (Y → αX • β, i) to S(k)
        end for
    end for
end for
```

Figure 6: Earley parser algorithm

other parsing algorithms, these productions are not preprocessed in any way before parsing.

The Earley parser maintains a state $S(i)$ for every terminal input symbol a_i. The list of states for every input symbol is called *a chart*. Each state $S(i)$ contains one or more *items* in the form $(X \rightarrow \alpha \bullet \beta, j)$. Each item contains a production rule, the current parsing position within that rule (represented by $\bullet$) and the origin state j. Initially, $S(0)$ contains only the starting production $(S \rightarrow \bullet\alpha, 0)$. After executing the Earley algorithm (see Fig. 6), the chart S has enough information to construct the parse tree for the provided input.

The algorithm is split into three logical steps that are repeated in sequence for every input symbol:

- The **prediction** step finds all items in the current state in the form $(X \rightarrow \alpha \bullet Y, j)$, where Y is a non-terminal symbol, and adds every production rule with product Y to the current state. This is where the top-down nature of the Earley algorithm becomes apparent. If we view this algorithm from a procedural point of view, then this step may be considered a rule call, where the caller rule is suspended to complete the called productions.

- The **scanning** step finds all items in the current state in the form $(X \rightarrow \alpha \bullet a\beta, j)$, where a is current input symbol, and after advancing, adds those items to the following state. In other words, this is the step where the terminal symbols are matched with the appropriate production parts.

- The **completion** step finds all production rules that have been fully parsed in the current state and resumes the parsing of the caller productions that have been previously suspended in the prediction step.

Based on the steps that the algorithm performs, it becomes apparent that it is very easy to modify the list of production rules that are used for parsing. The grammar is accessed only in the prediction step when looking for appropriate product right-hand sides. By modifying the list of the production rules used while parsing, it is possible to augment the syntax of the parsed language mid-parse. This property of the Earley parser makes it very suitable for implementing an REP language parser. However, this flexibility comes at the cost of low overall parsing performance because the grammar and production rules it contains must be traversed by the algorithm many times during parsing.

The Earley algorithm makes no assumptions about the nature of the input symbols. These symbols can be characters of the original language or lexer tokens. However, when using the Earley parser as a scannerless parser, the performance considerations of using unprocessed grammar productions within the prediction step become even more important because defining language tokens as productions would further increase the depth of the AST and cause dramatically reduced performance. This makes the algorithm practically unsuitable for scannerless parsing.

Additionally, because the Earley parser supports all non-nullable context-free grammars, it means that it is possible to provide a grammar that results in parsing ambiguities. This means that, just like in the case of the GLR parsers, a single AST is not sufficient to express the structure of an ambiguous parse and that more sophisticated data structures, such as shared packed parse forests (SPPFs), are required. However, the original paper in which the parser was first described does not address this issue in enough detail. This discrepancy was first observed by [25], where a modified version of the algorithm is provided, which produces SPPF for ambiguous parses.

3.1.9 Parsing reflective grammars

The idea that the Earley parsing algorithm can be extended to support mutable grammars was noticed by [28]. The paper's authors presented a modified version of the Earley recogniser, which supports parsing *reflective grammars*.

Reflective grammars are a type of grammar that can modify themselves by

```
for all input symbols a do
  for all (X → α • Yβ, j, G) ∈ S(k) do                    ▷ Prediction
    for all (Y → γ) ∈ G do
      add (Y → •γ, k, G) to S(k)
    end for
  end for
  for all (X → α • aβ, j, G) ∈ S(k) do                     ▷ Scanning
    add (X → αa • β, j, G) to S(k + 1)
  end for
  for all (X → γ•, j, G0) ∈ S(k) do                       ▷ Completion
    for all (Y → α • Xβ, i, G) ∈ S(j) do
      add (Y → αX • β, i, G) to S(k)
    end for
  end for
end for
```

Figure 7: The modified Earley algorithm

```
plus(4, plus(5,
  {{ grammar <Expr>
      <Expr> ::= <SimpleExpr> "+" <Expr> ;
    end
  6 + plus(1, 2 + 3) }}
  ), 7)
```

Figure 8: An example expression that uses the reflective capabilities of the modified Earley parser to add the binary infix $+$ operator

adding additional productions mid-parse from the recognised symbols within the parsed input. This is achieved by adding the following modifications to the original parsing algorithm (see Fig. 7):

- State items now have an additional element that represents the current grammar. If the original Earley parser uses items in the form $(X \to \alpha \bullet \beta, j)$, then the modified recogniser uses items in the form $(X \to \alpha \bullet \beta, j, G)$, where G is the current grammar.

- The prediction step, instead of using a single global grammar for the whole input, now uses the grammar from the current item. This enables the algorithm to use multiple grammars at the same time.

- A special meta-grammar for defining grammars is introduced. This meta-grammar, among other things, provides a production rule that can be used to both define an extension and invoke it with a specified starting symbol. The non-terminal product for this production can be included in user-defined grammars, thus giving users the ability to control where the newly defined symbol can be placed within the initial language syntax extensions. Figure 8 shows an example of using this non-terminal to introduce a binary addition operator to the initial language.

Some important observations are as follows:

- The parsing algorithm handles even cases where the user-defined grammar overlaps with the extension grammar. In this case, ambiguities may arise, but the parsing algorithm would continue to work correctly.

- Even though the modified algorithm based on its definition requires the extension grammars to be provided together with their invocations, it is possible to further modify the algorithm and separate the extension definition from the invocation.

- The modified algorithm is almost identical performance-wise to the original Earley parser. Because of this, the same considerations for using this algorithm as a scannerless parser apply, thus making the reflective and scannerless version of the Earley parser just as unusable as the original in any practical environment.

```
(1) E → E + F
(2) E → F
(3) F → F * I
(4) F → I
(5) I → 0
```

Figure 9: An example of a grammar that supports infix + and * operators with the appropriate operator precedence

3.1.10 Efficient Earley parsing

The primary reason for not using the original Earley parser in practice is its lower unambiguous language parsing performance. Consider the grammar provided in Fig. 9. Every time an expression E is to be parsed at input position j, the items shown in Fig. 10 must be added to the current state. In other words, the whole expression hierarchy must be expanded every time a possibility exists to encounter an expression in the current input position. A similar situation arises when attempting to parse statements as well. It is common for a programming language to have more than 20 different operators, and the productions for each would have to be expanded every time an expression may begin.

When using the Earley parser for a scannerless parser, the performance decrease would be even grimmer. Because the parser does not have any mechanism to perform a lookahead, when parsing an identifier that is part of a larger expression, it must prepare to both continue parsing the current identifier and attempt to parse the operator that comes after the identifier ends. As a result, on every parsed character of an identifier or numeric constant, it must reduce the current identifier (or constant) to an expression and advance all the previous productions that depend on that expression. In the case of parsing C++, which has 16 different operator precedences, the Earley parser would have to perform at least 16 reductions and 16 completions for every identifier expression or constant expression in the whole input file. Obviously, such an implementation is simply infeasible.

Several modifications to the original Earley parser have been proposed to increase its performance. The first one [2] observes that the Earley sets closely correspond to LR(0) DFAs. By using DFAs computed from grammar productions instead of raw grammar productions to perform recognition, the parser

```
E → •E + F, j
E → •F, j
F → •F * I, j
F → •I, j
I → •0, j
```

Figure 10: Earley items for expanding non-terminal E in position j with the grammar from the expression grammar

no longer needs to traverse the entire expression/statement hierarchy when encountering such non-terminals. As a result, the Earley item now contains (q, j), where q is the state number of the corresponding DFA node, and j is the origin state (i.e. the position in which the parsing of the current non-terminal began).

While such an optimisation massively boosts the Earley parser performance, it also eliminates the simplicity of adding new grammar productions. Because the efficient variation of the Earley parser uses DFAs to internally represent the grammar structure, incremental construction to generate these DFAs on-demand must be applied to use the modified parser for REP language parsing.

3.1.11 Yakker parser

Another parsing algorithm that attempts to make Earley parsing more efficient has been described by [15]. The authors of this paper observed that each production rule can be represented by a nondeterministic finite automaton (NFA) like that displayed in Fig. 11. Additionally, treating production rules as automatons enables the use of regular expression operators in these productions to make their definition more convenient. Furthermore, these productions can be interconnected by *call* edges, which eliminate the need to dynamically look up productions of a specific non-terminal in the prediction step (see Fig. 12). Because the parser is now represented by a single NFA, it is possible to optimise it by performing specialised minimisation, which treats $S_a \xrightarrow{call} S_b \xrightarrow{call} S_c$ as $S_a \xrightarrow{call} S_b \xrightarrow{\varepsilon} S_c$. After applying such an optimisation, all items in Fig. 10 are merged into a single state, thus solving the previously described problem of having to traverse the entire expression hierarchy each time a possible expression is encountered.

At this point, the optimised DFA resembles the LR(0) DFA used by the optimised Earley parser described in 3.1.10. The same authors then use this new parsing algorithm as a basis for the Yakker parser [14], which introduces new

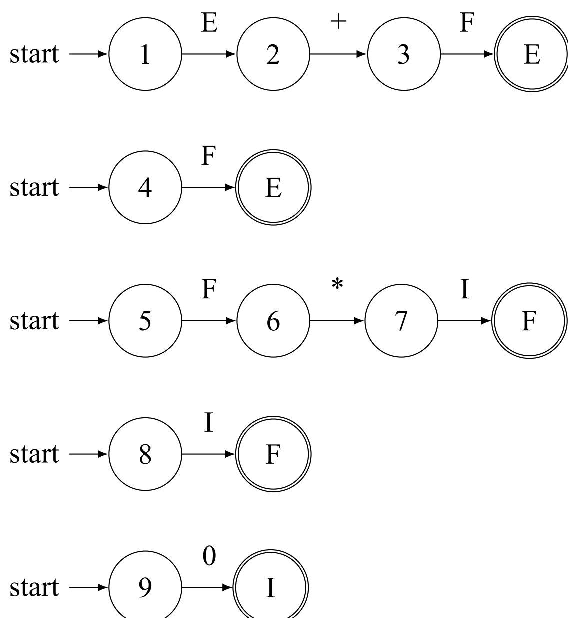

Figure 11: Earley deterministic finite automata for production rules from the expression grammar

features that are not present in the original or modified Earley parser, the primary of which is the ability to recognise data-dependent grammars. To support such grammars, new grammar definition primitives are introduced:

- Attribute bindings in the form $x = e$, where x is a variable and e is an expression.
- Non-terminal symbol invocations with bindings in the form $x = A(e)$, where A is a non-terminal symbol. This grammar construct allows it to not only parse non-terminal symbols by supplying them arguments but also store the result of a parse in a variable that may be later used to form a semantic data-dependent constraint.
- Constraints in the form $[e]$, which can be considered ε symbols, which are accepted only if expression e is true.

To support such grammar primitives, corresponding additional Yakker automaton nodes are introduced. Additionally, the Earley item is extended to hold environment E, which stores all local variable bindings, resulting in items in the form (q, j, E). An example Yakker grammar that describes the fixed-length

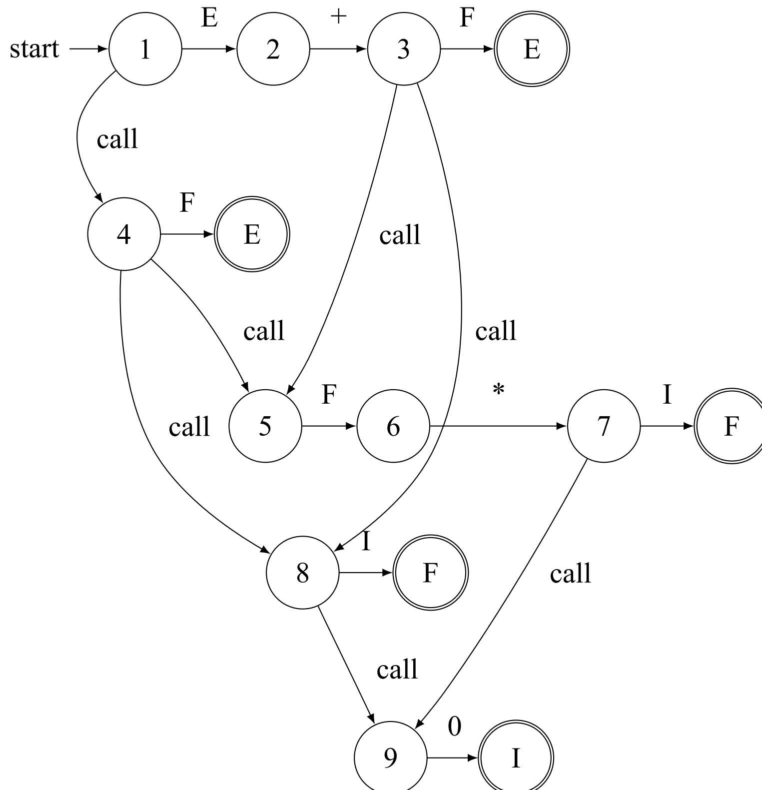

Figure 12: Interconnected Earley deterministic finite automata for production rules from the expression grammar

```
int(n) = [n = 0] | ([n > 0] digit int(n - 1))
```

Figure 13: An example Yakker grammar that allows parsing fixed-length numbers

numbers is provided in Fig. 13.

Yakker is the most general and flexible of all analysed parsing algorithms. It supports regular right-hand sides (the ability to use regular expression operators to define grammar productions). The parser exhibits acceptable performance even when used without a lexer and can parse all context-free languages even without using data-dependent constraints. Using data-dependent constraints allows it to parse an even wider class of grammars. For example, the parser may be used to recognise well-formed XML files without using external automatons to match the opening and closing tags of this language.

A common task that is performed during parsing is AST construction. In the case of LR/GLR parsers, the AST nodes for some parsed input are constructed either automatically during reduction execution or manually by invoking a user-defined semantic action during the reduction process. In the case of Earley or Yakker parsers, this can be done in the same way during the completion step. However, in the case of ambiguous grammars or no lookahead, many intermediate parse results might be constructed and then immediately discarded after entering an invalid parse path. Depending on the type of semantic action, this operation may be memory intensive and could be a resource drain, thus slowing the overall parsing process. To combat this, *delayed semantic actions* are introduced to the Yakker parser in [16], which can be executed to construct the AST after successfully parsing some or all of the input, thus eliminating the unnecessary construction of invalid AST nodes.

This parsing algorithm fulfils all the requirements for an REP language parser except one: mutable grammar support. To support mutable grammars, the automata of Yakker would have to be generated incrementally. Furthermore, new grammar constructs would have to be introduced similar to the ones described in [28] to support parsing reflective grammars. Even despite the required effort to implement such functionality, the Yakker parser is a good candidate for implementing an REP language parser because the provided feature-set, generality, and parsing performance combination cannot be matched by any other parsing algorithm.

Table 1: Summary of the analysed parsing methods

Parser	Parser family	Dynamic grammars	Scannerless	Generalized	Local extensions	Acceptable performance
LR(1)	(G)LR					✔
LALR(1)	(G)LR					✔
GLR	(G)LR			✔		
RNGLR	(G)LR			✔		✔
SGLR	(G)LR		✔	✔		✔
Recursive descent						✔
Packrat	PEG		✔			✔
APEG	PEG	✔				✔
Specificity		✔				✔
Earley	Earley			✔		
Reflective	Earley	✔		✔	✔	
Efficient Earley	Earley			✔		✔
Yakker	Earley			✔		✔

3.1.12 Parsing method summary

The summary of all reviewed parsing methods is shown in Table 1. For each listed parsing method, the parser family is shown together with the requirements that each parser satisfies.

3.2 Related Tools and Languages

3.2.1 Katahdin

Katahdin [26] is one of the very few REP languages that exist. It is a dynamically typed language that allows the mutation of the base language syntax and semantics. The dynamic nature of the language also allows the definition of extensions in external libraries. A simple Katahdin program that uses multiple language extensions is provided in Fig. 14.

Every language extension within Katahdin is composed of two elements: the syntax definition and evaluation rules, which are both placed within a class

```
import "fortran.kat";
import "python.kat";

fortran {
  SUBROUTINE ADD(A, B)
    INTEGER A
    INTEGER B
    A = A + B
    RETURN
  END
}

python {
  total = 0
  for i in range(10):
    ADD(total, i)
    print total
}
```

Figure 14: An example Katahdin program that uses Fortran and Python language extensions

```
class IncrementExpression: Expression
{
  pattern
  {
    option recursive = false;
    expression:Expression "++"
  }

  method Get()
  {
    value = this.expression.Get...() + 1;
    this.expression.Set...(value);
    return value;
  }
}
```

Figure 15: An example Katahdin extension that implements the unary suffix increment operator ++

that represents the AST node for the newly defined construct. Syntactic extensions are defined using PEGs, which are later used by a backtracking recursive descent parser. Evaluation rules are defined as a series of methods with direct access to the parse tree that allow the interpretation of the current AST node. For example, all expressions in the base language and in its extensions have a `Get()` method that evaluates the current node and returns the value for that node. Similarly, all statements have a `Run()` method that executes the provided statement. An example of implementing a simple extension is provided in Fig. 15.

While such a method for defining language extensions is very intuitive to use, it has quite a few limitations as well:

- Katahdin uses PEGs to define the syntax for new language constructs. This results in no support for left recursion, no local ambiguity support, and no true language union support. The limitations of PEGs in relation to the REP languages are explored in more detail in Section 3.1.5.
- The choice of a recursive descent parser is questionable as well. A recursive descent parser with backtracking, while easy to implement, is notorious for exhibiting poor performance because backtracking may take exponential time to execute.
- Syntax extensions are global. It is not possible to activate an extension for a selected scope only, like when using other parsing methods, such as the one described in Section 3.1.9.
- Katahdin is fully dynamically typed. The author of the language claims that this choice allows the language to support both dynamically and statically typed extensions because dynamically typed extensions provide more generality. However, this may not be true. It is entirely possible to mix static typing with dynamic typing by introducing dynamic types within a statically typed language. The choice of dynamic typing, besides having poorer performance, defeats one of the key motivators for using an REP language: compile-time error checking that ensures that two code fragments from two different languages integrate correctly.
- It has poor performance. Because Katahdin is a dynamic programming language, its libraries are provided in a textual (non-binary) format. That means that each time a language library is imported, it must be parsed

with a fairly inefficient parsing method. After imported libraries are parsed and the constructs within them are evaluated, the actual user program may only then begin execution. With the current Katahdin implementation, it takes several seconds just to parse the standard library. In addition, the performance of user programs even after parsing them is low because Katahdin uses AST interpretation to execute its programs.

Because of these restrictions, we do not believe that Katahdin is a true contender for a practical REP language.

3.2.2 SugarJ

SugarJ [10] is a programming language that supports library-based syntactic language extensibility. The language's authors introduce a new type of libraries, called *sugar libraries*, which in addition to exporting classes and functions, also export syntactic extensions. Even though the authors claim that the sugar library concept is novel, a less formal variation of sugar libraries was also implemented previously in Katahdin.

Unlike Katahdin, which uses interpretation to execute its programs, SugarJ transforms all of its programs into Java code than can then be compiled by a regular Java compiler. Syntax extensions in SugarJ are defined using new language constructs called *sugars*.

A sugar in SugarJ is a declaration (just like a class in Java), which defines the syntax of an extension using context-free grammars and provides desugaring rules, which rewrite the AST nodes of new constructs to mostly Java AST nodes (see Fig. 16 for an example of a sugar declaration). This allows adding only paraphrase extensions to the base language.

SugarJ is implemented by dividing the translation process into the following steps:

1. **Parsing**. The translator parses a single top-level declaration using Stratego/SDF with the current grammar. Stratego [5] is a language transformation framework that allows defining grammar using context-free grammars. It also provides the capability to define rewrite rules, which are used directly in SugarJ to transform ASTs. The actual parsing process within Stratego is done using a scannerless GLR parser [9].

2. **Desugaring**. This is the transformation step, where all sugar AST nodes

```
package pair;
import org.sugarj.languages.Java;
import concretesyntax.Java;
public sugar Sugar {
  context-free syntax
    "(" JavaType "," JavaType ")" -> JavaType{cons("PType")}
    "(" JavaExpr "," JavaExpr ")" -> JavaExpr{cons("PExpr")}
  desugarings
    desugar-pair-expr
    desugar-pair-type
  rules
    desugar-pair-expr:
      PExpr(e1, e2) -> |[ pair.Pair.create(~e1, ~e2) ]|
    desugar-pair-type:
      PType(t1, t2) -> |[ pair.Pair<~t1, ~t2> ]|
}
```

Figure 16: An example of a SugarJ extension that implements the unary suffix increment operator ++

within the previously parsed declaration are replaced with appropriate SugarJ nodes based on transformation rules in sugar definitions.

3. **Splitting**. At this point, the AST contains only SugarJ nodes. The AST is then split into fragments of Java, import statements, and sugar declarations. Fragments of Java contribute to the final translated Java program. The import statements are used to load external sugar libraries, while the sugar declarations are passed to Stratego.

4. **Adaptation**. During this step, new sugar declarations are merged with the current desugaring rules. In the same fashion, newly defined production rules are composed using the current grammar to form a new grammar that is capable of recognising newly defined sugars. This new grammar and new sugars are then used to parse the subsequent top-level declaration.

In other words, SugarJ uses naive extensible parsing in conjunction with a scannerless GLR parser. This means that sugars are applied globally to all subsequent top-level blocks. This prohibits the creation of extensions that only work in specific scopes. Furthermore, the performance of such a parsing method is not ideal (see Section 3.1.1 for more details). As such, there is room for improvement regarding syntactic extensibility.

In addition, because SugarJ supports only syntax extensions, it is not a true

Table 2: Summary of analysed tools and languages

Language/ tool	Dynamic grammars	Scannerless	Generalized	Local extensions	Acceptable parsing performance	Acceptable grammar change performance
Katahdin	✔					✔
SugarJ	✔	✔	✔		✔	
Neverlang	✔				✔	✔

REP language. However, the AST transformation method used in this language is rather general and thus applicable to REP languages as well. Unfortunately, the same cannot be said about SugarJ's parsing method.

3.2.3 Neverlang

Neverlang [30] is a framework for sectional compiler construction. It introduces the concept of splitting the definition of a compiler into slices, where each slice defines a single feature for the target language. Each slice contains the syntax definition, type-checking rules, and evaluation rules. Eventually, multiple named slices are composed into a single language, and the compiler for that language is generated. Neverlang uses a dedicated lexer with an incrementally generated LALR(1) parser to parse the target language, which means that it does not support scannerless parsers and arbitrary context-free grammars. Thus, it does not meet our criteria for REP language parsing. However, the idea of dividing the definition of a language into mostly self-contained slices provides a clean way to manage different extensions that may exist within the REP language and should be eventually investigated with more detail regarding using slices in an REP language compiler.

3.2.4 Tool and language summary

The summary of all reviewed related tools and languages is shown in Table 2. This table also includes a column on *acceptable grammar change performance* that indicates whether the selected language or tool is capable of making rapid changes to active grammars without regenerating the entire internal

grammar representation with each (potentially minor) grammar change.

3.3 Conclusions

In this chapter, we defined the requirements for an REP language parser and investigated various parsing algorithms to meet these requirements. After conducting the literature analysis, we found that no single parsing algorithm or programming language fully satisfies our requirements, but several were almost satisfactory.

Only one of the analysed extensible programming languages uses an algorithm specifically designed for extensible grammars: Neverlang. However, the parsing method of Neverlang is based on an incremental LALR(1) with a dedicated lexer and thus severely limits the grammar extension flexibility (because not all extension grammars may be LALR(1)). The other two analysed languages (Katahdin and SugarJ) use more traditional parsing methods, which were 'naively' extended to support extensible grammars (by either regenerating entire parse tables after each grammar change in SugarJ's case or using a very inefficient backtracking recursive descent parser that does not preprocess or optimise the grammar rules in Katahdin).

The closest parsing algorithms to meet our criteria are the reflective parser and Yakker. The reflective parser (as described in 3.1.9) is the only one that supports dynamic grammar updates with local grammar extensions, but it lacks support for scannerless grammars and, more importantly, is rather inefficient. On the other hand, Yakker does not support any kind of dynamic grammar updates; however, it is rather fast. Both of these parsing algorithms are derived from the Earley parser. As a result, we believe that the Earley parser (and its derivatives) would serve as a good basis for constructing a modified version of the algorithm that would fully satisfy all the requirements for parsing REP languages.

4 EXTENSIBLE PARSING WITH THE EARLEY VIRTUAL MACHINE

4.1 Earley Virtual Machine

4.1.1 Introduction to the Earley virtual machine

The Earley virtual machine (EVM) is a new approach to parsing that is based on virtual machines and is heavily inspired by the Earley parser. The core idea behind EVM is to separate the two grammar representations used by the parser: the user writes *source grammars* in a plain-text format, which are then parsed and compiled into *compiled grammars* that are then executed by the parser.

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

- **Source grammars** are parser grammars in plain-text format. These grammars are written by the user of the parser and describe the parsed language in terms of grammar rules. Additionally, source grammars may contain the AST construction instructions, which allow controlling the process of AST construction in fine detail.

- **Compiled grammars** or **grammar modules** are internal representations of source grammars. As the name implies, compiled grammars are compiled from source grammars. Compiled grammars contain a sequence of low-level instructions that drive the parsing process.

- The **interpreter** is one of the primary elements of the EVM. It *interprets* or executes the instructions contained in one or more grammar modules. As a result, an AST is constructed based on the parse input. The process of interpreting compiled grammars is synonymous to parsing the input data in the context of the EVM.

- The **states** of the EVM are internal structures used by the interpreter to track the execution of the interpreter. These EVM states have a close resemblance to the Earley parser states. One EVM state may exist per terminal symbol.

- The EVM **fibers** have a close relationship with the Earley parser items.

Each fiber represents a task of grammar rule execution. A fiber may be thought of as a thread of a general-purpose programming language in which one grammar rule is executed.

- The **fiber queue** is a queue of fibers that are ready for execution. The interpreter works by removing the first fiber from the queue and executes it until it yields. At this point, the next fiber is removed from the queue and the execution of it commences. An empty fiber queue indicates a parse error.

4.1.2 Earley virtual machine grammars

Much like formal grammars, *basic EVM grammars* consist of production rules, where each production rule defines how to parse a single non-terminal symbol. More formally, a *basic EVM grammar* is a set of productions in the form $sym \to body$, where *sym* is a non-terminal symbol and *body* is a *grammar expression*.

A *grammar expression* is defined recursively as follows:

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

The EVM *compiled grammar* is a tuple $\langle instrs, rule_map \rangle$. Moreover, *instrs* is the sequence of instructions that represent the source grammar. In addition, *rule_map* is the mapping from the non-terminal symbols to locations in the instruction sequence, which represents entry points for the grammar program. It is used to determine the start locations of compiled rules for specific non-terminal symbols.

4.1.3 Earley virtual machine states

An EVM state is a structure that tracks the progress of the interpreter at a particular point in the terminal symbol input sequence. Each EVM state has an index that corresponds to an appropriate position of the input sequence. Each EVM state S_i is a tuple $\langle susp, trace, reductions \rangle$, where the following conditions apply:

- The term i is the position of the input sequence.
- *Susp* is a list of suspended tasks at position i. When one rule calls another, the caller is suspended until one or more of the callees are complete. Each entry of the suspended task list is a pair $\langle fiber, symbol_map \rangle$, where $fiber$ is the suspended fiber. *symbol_map* represents the reason for the suspension: it contains the set of non-terminal symbols that the callee expects to parse. Upon parsing any of these symbols, the caller fiber is resumed by adding its copy to the fiber queue (thus signalling that the target non-terminal symbol has been parsed successfully and that the parsing of the caller rule may resume).
- $Trace$ is the *execution trace set* (ETS). It is a set of pairs $\langle ip, stack \rangle$. Whenever a new fiber is created (either by calling a new non-terminal symbol or by resuming a suspended fiber), the instruction pointer ip and the *stack* of the *candidate* fiber is checked against the ETS. If the pair is not present in the ETS, then the creation of the fiber commences, and this pair is added to the ETS. Otherwise, the creation of the fiber is aborted. This mechanism ensures that the input position is parsed with the same grammar rule and the same context only once, thus avoiding the exponential parsing complexity found in certain variations of the recursive descent parser. The ETS also blocks infinite left recursion.
- *Reductions* is a multimap that stores successful reductions that originate from the state/offset i. The key of the multimap is a non-terminal symbol that indicates the target symbol, where the value of the map is a tuple $\langle offset_1, priority, value \rangle$. In addition, $offset_1$ indicates the end position of the reduction, and the *priority* indicates the priority of the reduction. This value is used in conjunction with negative reductions and is described in more detail later. The *value* is the user-specified value of the

reduction. It usually contains the AST node of the reduction or, when delayed semantic actions are used, the label of the reduction. The primary purpose of the *reductions* is to store the intermediate parsing results. Additionally, it is used to merge reductions whose starting positions, ending positions, and non-terminal symbols match. This avoids the exponential complexity explosion in the case of ambiguous grammars or inputs.

4.1.4 Earley virtual machine fibers

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

- The *origin* is the origin input position of the fiber. It indicates the starting position of the target non-terminal symbol in the terminal symbol input sequence. This value is used when completing reductions. A successful non-terminal symbol reduction is recorded in the variable *reductions* of the S_{origin} of the state. Additionally, appropriate suspended threads of the S_{origin} of the state are resumed in the S_{offset} of the state.
- The $offset$ indicates the input position of the current fiber. When a single terminal symbol is parsed successfully, the current fiber is *advanced* by increasing this offset by 1.
- The term ip indicates the instruction pointer of the current fiber.

4.1.5 Earley virtual machine interpreter

Parsing terminal symbols

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

In most basic cases, this instruction can be used only with a single entry in its jump table. However, when using a subset construction optimisation, multiple `i_match_char` instructions can be merged into one by combining their jump tables. In the case of a successful terminal symbol match, the ip of the current

Table 3: Terminal symbol sequence parsing example

Grammar rule	Instruction sequence

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

fiber is set to the appropriate instruction pointer provided in the jump table. Additionally, the current fiber is advanced by increasing the $offset$ by 1. In the case of matching failure (when no terminal symbol in the jump table matches the one in the $offset$ position of the input), the current fiber is immediately discarded. The execution of the fiber is halted, and the fiber yields. An example of a simple source grammar and its instruction sequence is provided in Table 3.

Parsing non-terminal symbols

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

- `i_call_dyn` S is used to *initiate* parsing non-terminal symbol S. This instruction creates one or more fibers. The instruction pointers of newly created fibers are set to the entry points of the compiled rules that define the non-terminal symbol S. The *origin* of the new fibers is set to the $offset$ of the caller. Finally, newly created fibers are added to the fiber queue. It is important to note that the fiber creation process is still subject to the ETS rules. Multiple `i_call_dyn` invocations to the same non-terminal symbol S will not result in creating additional fibers. After executing the `i_call_dyn` instruction, the caller fiber continues its execution normally.
- `i_match_sym` $S_1 \rightarrow ip_1, ..., S_n \rightarrow ip_n$ is used to match successful non-terminal symbol parses that have been previously initiated by the `i_call` family of instructions. Whenever a `i_match_sym` is executed, the current fiber is suspended by adding it to the list of suspended fibers *susp* in the

Table 4: Non-terminal symbol sequence parsing example

Grammar rule	Instruction sequence

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

S_{offset} state. Additionally, the interpreter attempts to pre-emptively resume the suspended fiber in case any of the target non-terminal symbols have been successfully parsed prior to executing the current `i_match_sym` instruction.

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

A simple example of matching several non-terminal symbols is provided in Table 4.

Resuming suspended fibers

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

1. The suspended thread is duplicated.

2. The term ip of the copy is set to target the instruction pointer, which is retrieved from *symbol_map*.

3. The $offset$ of the copy is set to the $offset$ of the fiber that executes `i_reduce`. In the case of a pre-emptive resumption in `i_match_sym`, the new $offset$ value is retrieved from the *reductions* entry in the S_{offset} state.

4. The new fiber is traced by recording its presence in the state's S_{offset} execution trace set. If a matching entry already exists, the resumption of the fiber is aborted.

5. The new fiber is added to the fiber queue to be executed later by the interpreter.

4.2 Compiling Basic Earley Virtual Machine Grammars

The rules for compiling basic source grammars to corresponding instruction sequences are shown in Table 5. The notation *codeI* refers to the corresponding sequence of instructions when compiling grammar element *e*. Instruction `i_accept` signals the interpreter that a matching input has been parsed. Moreover, *main* is the name of the starting non-terminal symbol of the grammar that is being compiled.

4.3 General-Purpose Computation in the Earley Virtual Machine

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

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

- `i_br` ip. Unconditional branch to instruction pointer ip.

- `i_bz` ip. Conditional branch to instruction pointer ip. The branch con-

Table 5: Basic source grammar compilation rules

Grammar element	**Instruction sequence**

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

Production rule: $P \rightarrow e$	$code(e)$ `i_reduce` P, 0 `i_stop`
Terminal grammar expression: a	`i_match_char` $a \rightarrow ip_{next}$
Non-terminal grammar expression (dynamic): A	`i_call_dyn` A `i_match_sym` $A \rightarrow ip_{next}$
Epsilon grammar expression: ε	
Brace grammar expression: (e)	$code(e)$
Sequence grammar expression: e_1e_2	$code(e_1)$ $code(e_2)$

dition value is popped from the top of the current fiber stack.

- `i_pop`. Remove and discard the top stack element of the current fiber.
- `i_peek` n. Duplicate stack element n and push it to the top of the stack.
- `i_int_add`. Integer addition; pop two values from the top of the stack and add them as integers and push the result to the top of the stack.
- `i_int_sub`. Integer subtraction.
- `i_int_neg`. Integer negation.
- `i_int_push`. Push the immediate integer constant to the top of the stack.
- `i_int_more`. Integer comparison.

- `i_str_push`. Push the reference of the string constant to the stack.
- `i_call_foreign` *id*, *n*. Call the foreign method identified by index *id* with *n* arguments. Push the result of the call to the stack. Foreign methods are methods implemented in the host environment of the EVM and can be used to extend the functionality of the EVM without having to directly modify the way the EVM is implemented.

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

4.4 Improving Source Grammar Flexibility

4.4.1 Regular right-hand sides in production rules

Regular right-hand sides is a feature commonly found in the recursive descent and packrat family of parsers [11]. It allows the usage of regular operators on the right-hand sides of production rules. This simplifies the definition of new grammars because repeated and optional grammar elements no longer need to be expressed solely via alternation and recursion.

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

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

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

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

Table 6: Regular operator compilation rules

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

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

Figure 17: A grammar that defines simple expressions with binary operators

The rules for compiling the new operators into instruction sequences are provided in Table 6.

4.4.2 Rule and operator precedence

Almost every existing programming language supports the notion of binary operators with differing precedences. In grammars, such operators with different precedences are commonly implemented via operator expression hierarchies, as shown in Fig. 17. Each different operator precedence level has a separate non-terminal symbol, under which the operators with that precedence level are defined. While such an operator with this precedence definition method is simple and easy to understand, it quickly becomes cumbersome when dealing with real-world programming languages, such as C++, Ruby, and similar languages, which often have over 15 different levels of operator precedences.

Furthermore, extending such language grammars to include additional operators becomes difficult, especially when the new operator has a precedence level that is between two existing neighbouring precedence levels. In that case, a new non-terminal symbol for the new operator precedence level must be defined, and the existing rule that defines lower-precedence operators must be updated to use the newly defined operator.

Because the definition of operators (either unary or binary) is such a fundamental task when defining new grammars for programming languages, newer parser generators and language translation frameworks often allow specifying the precedences of operators directly by either assigning each operator a numeric precedence value or using the operator definition order to infer the precedence of each operator [9]. As such, it would be beneficial for the EVM to support the specification of operator precedence levels natively, especially because one of the goals of the EVM is to support adaptable grammars that can be extended dynamically during runtime.

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

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

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

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

the non-terminal symbols with a specified precedence:

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

4.4.3 Specifying operator associativity

Operator associativity can be considered a separate edge case of rule precedence. The left-associative operator $E + E$ means that the left non-terminal E can be expanded recursively into itself, while the right E must be expanded into an expression only with higher precedence. As such, the operator associativity specification can be implemented using the operator precedence mechanism.

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

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

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

The example grammar in Fig. 17 can now be rewritten using the explicit rule precedences and non-terminal associative symbols as shown in Fig. 18. New operators can be added as needed by specifying the new production rules with explicit precedences. When adding new operators, no existing rules need to be changed or altered in any way.

The updated rules for generating instruction sequences for non-terminal symbols are provided in Table 7. In addition, the *prec* value refers to the precedence of the current rule that is being compiled. By default, this value is 0 if it is not specified explicitly using square bracket notation.

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

Figure 18: Rewritten grammar that defines simple expressions with binary operators

Table 7: Updated non-terminal symbol compilation rules

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

4.5 Parsing with Regular Lookahead

4.5.1 Fixed-length lookahead

Parsing using lookahead is a useful feature that can simplify specifying grammars. When using a parser in scannerless mode, lookahead becomes mandatory because it is needed to implement greedy-matching when defining the language tokens. For example, an identifier can be defined as a sequence of alphanumerical characters that terminate on the first non-alphanumerical symbol. As such, to correctly specify the termination point of an identifier, a single-character lookahead is required.

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

- The `i_advance` n instruction advances the current fiber by n symbols.

Table 8: Fixed-length lookahead example

Grammar rule	Instruction sequence

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

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

Figure 19: Grammar rule that defines identifier using fixed-length lookahead

Table 9: Fixed-length lookahead compilation rules

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

This operand may be negative to perform fixed-length backtracking.

To use this instruction, the definition of a grammar expression must be extended to include the following:

- The $\&e$ expression is a positive lookahead grammar expression, where e is a grammar expression.

An example of the usage of a positive lookahead operator is provided in Table 8. Figure 19 shows an example where the positive lookahead feature can be used in a real-world scenario when defining identifiers.

The rule for compiling fixed-length lookahead grammar expressions is provided in Table 9. In addition, $length(e)$ refers to the character (terminal symbol) length of grammar expression e.

4.5.2 Variable-length lookahead

Variable-length lookahead in the EVM can be implemented in a similar fashion. However, in this case, the difficulty is not knowing how many terminal symbols to backtrack after performing the lookahead operation. As such, this information can be recorded and used dynamically by leveraging the general-purpose computation capability of the EVM.

Table 10: Variable-length lookahead compilation rules

Grammar element	**Instruction sequence**

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

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

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

The rules for compiling variable-length lookahead grammar expressions are provided in Table 10. Both fixed and variable-length lookahead expressions share the same notation. It is up to the source grammar compiler to determine when the lookahead operation is a fixed length and to use the appropriate compilation rule. It is also possible to use the variable-length lookahead operation even in situations where the fixed-length lookahead operation would be more suitable, but with an additional performance cost because the variable-length lookahead operation uses the fiber stack.

4.6 Parsing with Data-Dependent Constraints

4.6.1 Earley virtual machine grammar language

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

Table 11 presents the updated grammar elements and their instruction sequence compilation rules. The list of new grammar elements is non-exhaustive and does not include additional variations of existing elements (e.g. various integer operations can be implemented in a similar fashion to integer addition just by changing the final instruction).

Every variable defined within the rule body is assigned a *stack slot*. A stack

Table 11: Extended grammar language elements and their compilation rules

Element name	Syntax	Instruction sequence
Grammar rule	**rule** $sym(a_1, ..., a_n)$ $stmt_1$ ... $stmt_n$ **end**	$code(stmt_1)$ ... $code(stmt_n)$ `i_reduce` sym, 0 `i_stop`
Block statement	$stmt_1$ ... $stmt_n$	$code(stmt_1)$ ... $code(stmt_n)$
If statement	**if** $cond$ $body$ **end**	$code(cond)$ `i_bz` l_{end} $code(body)$ l_{end}:
Parse statement	**parse** g_expr	$code(g_expr)$
Return statement	**return** $expr$	$code(expr)$ `i_reduce_r` sym, 0 `i_stop`
While statement	**while** $cond$ $body$ **end**	l_{start}: $code(cond)$ `i_bz` l_{end} $code(body)$ `i_br` l_{start} l_{end}:
Variable decl. statement	**var** $v = expr$	$code(expr)$
Integer addition expression	$e_1 + e_2$	$code(e_1)$ $code(e_2)$ `i_int_add`
Integer constant expression	$value$	`i_push_int` $value$
Variable read expression	v	`i_peek` $stack_slot_v$
Variable write expression:	$v = e$	$code(e)$ `i_poke` $stack_slot_v$
Parameterized non-terminal grammar expression	$A(a_1, a_2, ..., a_n)$	$code(a_1)$ $code(a_2)$ ... $code(a_n)$ `i_call_dyn` A, $prec_{min}$, n `i_match_dyn` A, $prec_{min}$

slot is a position in the fiber stack where the value for the variable is stored. Moreover, $stack_slot_v$ refers to the stack slot number for variable v.

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

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

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

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

Parse and other control statements can be mixed and matched to parse complex data-dependent grammars that cannot be parsed with traditional context-free parsers. For example, Table 12 shows how to parse fixed-length fields commonly found in binary formats.

4.6.2 Matching input against dynamic content

While the mechanism for dependent parsing described in the previous chapter is powerful, it is not sufficient to parse languages like XML. To be able to

Table 12: Parsing fixed-length fields

<table>
<tr><th>Grammar rule</th><th>Instruction sequence</th></tr>
<tr><td><pre>
rule field(n)
  while n > 0
    parse "a"
    n = n - 1
  end
end
</pre></td><td><pre>
10: i_peek 0
11: i_push_int 0
12: i_int_more
13: i_bz 20
14: i_match_char a -> 15
15: i_peek 0
16: i_push_int 1
17: i_int_sub
18: i_poke 0
19: i_br 10
20: i_reduce "field", 0
21: i_stop
</pre></td></tr>
</table>

parse XML, it is necessary to be able to extract a fragment of the parsed input and then use that extracted fragment for further matching.

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

- The $v@e$ expression is a *capturing grammar expression*, where e is a grammar expression, and v is a name (identifier) for a new variable. After successfully matching e, this operator will store the range (the start and end offsets) of the matched input.

- The $= v$ expression is a *dynamic match grammar expression*, where v is a variable that stores the input range. This operator is used to match the input against the one that is referenced by the range.

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

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

A grammar rule example that can match the simplified XML tags is provided in Fig. 20. This rule combines multiple key elements of the EVM to successfully parse the XML tags: the fixed-length lookahead, associative non-terminals, and dynamic matching.

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

Figure 20: Simplified XML tag grammar rule

Table 13: Rules for compiling capturing dynamic match grammar expressions

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

Rules for compiling newly added grammar expressions into instruction sequences are provided in Table 13. In addition, $stack_slot_{v0}$ and $stack_slot_{v1}$ refer to the stack slot indices of the values produced by the `i_push_offset` instructions.

4.7 Abstract Syntax Tree Construction

4.7.1 Automatic abstract syntax tree construction

The EVM in its current iteration cannot be called a parser because it only currently performs input recognition. As such, for the EVM to be truly useful and applicable, a method is needed to construct the AST of the matched input. The AST can be constructed in the EVM in multiple ways, and in this section, we describe the *automatic* AST construction that requires no grammar modifications or any additional input from the user to construct the AST.

Such a method of AST construction can be implemented by augmenting the definition of the EVM fiber. An additional stack can be added to each fiber that can store the child nodes of the current non-terminal symbol that is being parsed. To use such a stack, the following instructions must be updated:

- In addition to performing reduction, `i_reduce` $A, prio$ constructs the AST

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

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

The EVM is capable of parsing ambiguous grammars, in which case, the size of the AST may grow exponentially. To avoid this, shared packed parse forests (SPPFs) can be used [25]. In SPPFs, subtrees that refer to alternative parse paths are packed into a single ambiguous node.

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

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

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

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

- If a reduction is *non-unique* (or ambiguous), then a normal child node

is created, and it is assigned a unique index. However, this time, the existing node is converted to an ambiguous packed node, and the newly created node is added as its child.

The conversion of a non-ambiguous node to an ambiguous node works by duplicating the target node, assigning it a new unique index, and changing the target node type to ambiguous. The duplicate of the original is then added as the only child of the converted node.

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

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

- Inclusion of undesirable AST child nodes. The EVM is primarily a scannerless parser and will be used to parse whitespace. It is common to define a non-terminal symbol for recognising whitespace and then use that within other grammar rules. During automatic AST construction, nodes that represent whitespace will be added to the resulting AST, possibly unnecessarily increasing the overall size of the AST and littering it with nodes that carry no semantic information.

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

4.7.2 Manual abstract syntax tree construction

Manual AST construction is the polar opposite of automatic AST construction. Instead of requiring the EVM to define and construct the AST automatically, the responsibility of the AST definition and construction is moved completely to the user. Because the EVM supports general-purpose computation, it would be logical to assume that this method could be extended to enable manual and imperative construction of the AST.

First, the EVM grammar language must be extended with the following constructs:

- $v : E$ is a capturing non-terminal grammar expression, where v is the variable name for storing the captured result, and E is one of the available non-terminal grammar expressions (plain or associative).
- <*name* arg_1 ... arg_n> is a node construction expression. The node is constructed with head *name* and arguments arg_1 ... arg_n. Arguments can be other nodes, integer values, or string values.

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

An example usage of a manual AST construction is provided in Table 14.

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

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

Table 14: Grammar rule for parsing and abstract syntax tree node construction of binary addition

Grammar rule	Instruction sequence

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

possibly within the host environment) how to construct the AST.

Although this approach has numerous advantages for automatic AST construction, one key flaw still persists:

- Wasted resources during speculative parsing. As the EVM performs parsing breadth-first, quite a few parse paths are discarded. Consider the parsing expression $2+3*4$. Upon parsing the $2+3$ portion of the input, a complete addition node would be constructed and stored within the *reductions* entry of S_1. However, this node would never be used because the remainder of the input would eventually be parsed, and two additional nodes would be constructed (one for $3*4$ and one for the whole expression). The problem is two-fold: the highly speculative nature of the EVM and the too-eager construction of the resulting nodes. The problem becomes even more significant when using more 'heavy' nodes that contain fields that are meant to be used during the later stages of compilation (such as source ranges for error reporting or typing information for semantic analysis). In that case, both the memory usage of the unused nodes and the time it takes to construct them may become a significant performance drain on the overall parsing process.

Thus, it would be useful if the node construction could be delayed only until the parser is sure that the node will not be discarded.

4.7.3 Delayed semantic actions

Arguments for delayed semantic actions

The delayed semantic actions [16] are an attempt to avoid the too-eager computation within non-terminal rules that may not contribute to the parsing result in the Yakker parser [14]. In this section, we present an adaptation of the delayed semantic actions for the EVM. The core idea behind the delayed semantic actions is to separate parsing into two distinct phases:

- The **early** and non-deterministic phase performs parsing and constructs an execution history.
- The **late** and deterministic phase consumes the execution history and uses it to execute any necessary semantic actions (possibly for AST construction).

Consider the example in Table 14. It contains three semantic actions, whose executions can be delayed: the assignment of the l variable, the assignment of the r variable, and finally, the construction of the AST node. In the case of the EVM, delaying these three actions would mean that the fiber stack in many situations would become optional, thus making the fiber-suspension process more efficient because it is no longer necessary to both allocate and store the stacks of suspended fibers.

The advantage of delaying the AST construction becomes even more apparent in the example provided in Fig. 21. Both operators in the EVM that provide repetition ($+$ and $*$) are implemented in the EVM using the `i_fork` instruction, which makes a copy of the current fiber with an altered instruction pointer. In the case that the actual argument list consists of n elements, the EVM will perform n forks and reductions in the *arg_list* rule alone. As a result, $n+1$ *arg_list* nodes will be constructed, out of which n will be never used again (assuming that the grammar is non-ambiguous). As such, delaying the AST construction is of vital importance in the EVM.

Constructing the execution history labels

As mentioned previously, the core idea behind the delayed semantic actions is to construct the execution history composed of *labels* that somewhat mirror

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

Figure 21: Grammar rule for parsing the argument list separated by commas

the structure of the AST but with one key difference. Whereas the AST nodes are heavyweight and contain a significant amount of information, the individual labels are small and lightweight. These labels can be *replayed* (either in a separate late phase or in parallel during parsing), thus executing the semantic actions that have been previously delayed.

Several different label types are required:

- The *tag label* is a unary label. It stores a reference to the previous label and a general-purpose numeric value. The semantic meaning of the numeric value depends on other nearby labels.
- The *call label* is a binary label that indicates a call branch. It stores a reference to the previous label and a reference to the reduction label of the callee.
- The *normal reduction label* is a unary label that indicates a successful non-ambiguous reduction. It stores a reference to the previous label and the reduction tag. The reduction tag is a value that uniquely identifies a reduction. The normal reduction label may be mutated to an ambiguous reduction label.
- The *ambiguous reduction label* is a binary label that indicates an ambiguous reduction. It stores two references to the reduction labels, which may also be ambiguous.
- The *resolved reduction label* is a 0-ary label that stores the result of the reduction, which is computed by executing the corresponding delayed actions. The normal and ambiguous reduction labels can be mutated into resolved labels after they have been replayed. The use of the resolved labels avoids replaying the same reduction labels several times.
- The *nil label* is a 0-ary label that terminates the tag or call label chain.
- The *range label* is a unary label that holds a source range. It is used

when parsing language tokens to hold the starting and ending positions of a token, thus avoiding the need for two separate tag labels.

To facilitate the construction of labels, the definition of a fiber is extended to include a *current label*. The general-purpose stack is not used for holding labels because the fiber stack is a variably sized structure, thus requiring separate allocation.

Furthermore, additional instructions and existing instruction changes are required:

- The `i_trace` *tag* constructs a new tag label $\langle label, tag \rangle$ and sets the label of the current fiber to the newly constructed one. This instruction is used to delay the execution of statements and expressions within the rule definition.

- The `i_trace_offset` instruction sets *label* to $\langle label, offset \rangle$. It is used to capture the current parsing location so that it may be used when replaying the labels.

- The `i_trace_range` instruction sets *label* to $\langle label, origin, offset \rangle$. It is used to capture the input range of the current non-terminal so that it may be used when replaying labels.

- The `i_reduce` A and `i_reduce_r` A instructions now construct a normal reduction label $l_1 = \langle label, A \rangle$. Then, this label is registered by checking whether the new reduction is ambiguous. If this is true, the existing reduction label l_0 is duplicated, and a new ambiguous label $\langle l_0, l_1 \rangle$ is constructed **in place** of the old one.

- The `i_match_sym`, `i_match_dyn`, `i_reduce`, and `i_reduce_r` instructions now construct a call label when resuming the suspended fibers.

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

Compilation of grammars that use delayed semantic actions

The rules for compiling grammars with delayed semantic actions are provided in Table 15.

A *fully capturing parse statement* is a parse statement that contains a single capturing grammar expression that captures the entire input of a non-terminal

Table 15: Rules for compiling grammars with delayed semantic actions

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

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

In Table 15, $label_{next}$ refers to the next label index. Labels in capturing non-terminal grammar expressions are indexed from 100 to differentiate them from the ones generated with the `i_trace_offset` instruction. These labels are referred to as *action labels* because they refer to a delayed action (in this case, the assignment of a variable). Action labels are specifically defined to be locally but not globally unique. That means that, in every non-terminal rule action, the labels are numbered from 100. This further aids in performing instruction subset constructions because the `i_trace` instructions with the same tag may be merged together.

Replaying labels

The execution history labels are created within the EVM, often using specialised label creation instructions. However, they can be replayed outside of the EVM, possibly in the host environment. This reduces the difficulty of the

AST construction because the native data structures and method or function calls may be used to construct the AST.

When the EVM completes parsing, the reduction label of the starting symbol may be found in the *reductions* entry of state S_1, whose length matches the total length of the input. This label is the result of parsing and can be used independently of the EVM to perform semantic action playback.

The label playback process consists of several steps:

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

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

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

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

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

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

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

Table 16: The grammar rule and corresponding instruction sequence for binary addition when delayed semantic actions are used

Grammar rule	Instruction sequence

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

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

The grammar rule example provided in Table 14 can now be compiled into a different instruction sequence, which is shown in Table 16, when delayed semantic actions are used.

The replay function for the rule, which is implemented in Ruby programming language, is shown in Fig. 22. The method `each_action` iterates over the collected labels (starting from the second label). In addition, `prev_result` accesses the resolved value of the previously resolved label. The `create_add_node` is a user-defined method that constructs the binary addition AST node. It is important to note that the replay function can be implemented in any language and that it is not in any way bound just to the Ruby programming language. For example, the same replay function can be implemented in the C programming language, as shown in Fig. 23.

4.8 Parsing Reflective Grammars

One of the key reasons for choosing the Earley parser as the basis for constructing the parsing method for an REP language is its flexibility and limited need for grammar preprocessing. In this chapter, we describe how the EVM can be extended to support adaptable grammars. The approach for implement-

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

Figure 22: The replay function for binary addition in the Ruby programming language

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

Figure 23: The replay function for binary addition in the C programming language

ing adaptable grammar support in the EVM is inspired by [28].

4.8.1 Dynamic grammar composition

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

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

Unfortunately, a single global list of active grammar rules is insufficient to correctly parse any context-free grammar because the statement for grammar rule activation may be ambiguous. Thus, in such a situation, a parser must be able to parse the same input with two separate sets of grammar rules: one when the recognised statement activates new grammar rules and another for an ordinary statement. Therefore, the active list of grammar rules must be bound to a specific fiber.

To avoid having to make multiple copies of the active grammar rules, the target language can be divided into *domains*. A domain is a part of a grammar. Each grammar rule is assigned a set of domains. Each fiber has a set of active domains. If the set of rule domains is a subset of active fiber domains, then that grammar rule is considered active within the context of the domain. By manipulating the set of active domains, it is possible to dynamically extend and constrain the current language.

Additionally, this method of grammar division and domain activation can be used to eliminate certain flaws present in traditional parsers: for example, there is no reason that **break** should be a reserved keyword in the C programming language. Because the **break** keyword is meaningless outside of the loop and switch constructs, it should only be recognised as a keyword inside the bodies of such constructs. However, due to lexer and parser limitations, that is not

Table 17: Additional grammar language elements to support reflective grammars

<table>
<tr><th>Element name</th><th>Element syntax</th></tr>
<tr><td>Domain definition</td><td><pre>domain dom1</pre></td></tr>
<tr><td>Grammar rule with domain annotation</td><td><pre>@domains dom1 dom2 dom3
rule name
  stmt1
  ...
  stmtN
end</pre></td></tr>
<tr><td>Domain activation statement</td><td><pre>with_domains dom1 dom2 dom3
  stmt1
  ...
  stmtN
end</pre></td></tr>
</table>

the case. However, using the EVM, it would become possible to dynamically activate the rule for only the **break** keyword inside a looping construct body. Similarly, the **return** keyword (and the grammar rule for it) could be activated only within a function body and so on.

4.8.2 Extensions of the Earley virtual machine grammar language

To enable domain manipulation within the EVM, additional grammar elements are required. They are listed in Table 17:

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

An example of a simplified grammar that uses domains to enable a **break** statement only within the body of a loop statement is provided in Fig. 24. By adding every rule of a language extension to a specific domain, it is possible to enable or disable the entire language extension with a single statement.

```
domain loop

@domains loop
rule statement
  parse "break"
end

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

Figure 24: Example domain usage

4.8.3 Compiling Earley virtual machine grammars with domains

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

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

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

Table 18: Rule for compiling domain activation statements

Element name	Grammar element	Instruction sequence
Domain activation statement	**with_domains** d_1 ... d_n *body* **end**	`i_dom_push_active` `i_dom_enable` d_1 ... `i_dom_enable` d_n $code(body)$ `i_dom_restore` $stack_slot_{dom}$

These instructions can be used to compile the **with_domains** statement, as shown in Table 18. Furthermore, $stack_slot_{dom}$ refers to the stack slot that contains the previous active domain set pushed by `i_dom_push`.

4.8.4 Loading multiple grammar modules in Earley virtual machine

When using the EVM to parse a language, the base variant of that language will most likely be contained in a single compiled grammar module that will be loaded into the EVM during EVM initialisation. Language extensions then could be contained in separate grammar modules that can be both generated and loaded dynamically during parsing.

Loading multiple grammar modules in the EVM is not trivial because each grammar module has its own address space. To support multiple address spaces within the EVM, the instruction pointer can be extended to include the module index. That way each instruction pointer in the EVM that is stored internally (e.g. the ip of a fiber) is a pair $\langle id_{mod}, ip \rangle$, where id_{mod} is the module index and ip is the relative instruction pointer to the start of the module. All existing instructions use relative instruction pointers (e.g. `i_fork`, `i_match_char`, etc.). In practice, for performance reasons, several bits of the instruction pointer can be reserved for storing the module index. Thus, the instruction pointer could remain word-sized.

Additionally, all the grammar rules of any language extension should belong to a corresponding extension domain. That way the language extensions could be enabled dynamically only for the desired scopes with the `i_dom_enable_dyn` instruction. Additional instructions that work with absolute instruction pointers may be added in the future, if necessary, for performance reasons.

4.8.5 Parsing reflective grammars in the Earley virtual machine

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

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

2. Define the extension metalanguage within the base language. The EVM does not provide a specific extension metalanguage because the extension metalanguage should be defined to match the syntax of the base language. However, the extension metalanguage could be designed to be similar to the EVM grammar language. The extension metalanguage should include an extension activation construct to activate the defined language extensions.

3. Implement the compilation of the extension metalanguage node into a grammar module, as described in this chapter. If the extension language matches the EVM grammar language, then the rules for compiling the EVM grammar language elements can be used directly to implement this compilation step.

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

4.9 Earley Virtual Machine Performance Improvements

In this chapter, we describe several EVM optimisations that significantly increase the overall parsing performance both in terms of CPU time and memory usage.

4.9.1 Garbage collection of suspended fibers

The EVM currently creates a state for every input position where other non-terminal rules are invoked with the `i_call` instruction family. This state information is then used to record the execution trace, store the reduction information, and, most importantly, park the suspended fibers so they may be resumed later. All this information over time adds up to a significant amount. However, not all of it is needed for further parsing. There are several important observations to make:

- Most states and fibers will never be needed again after suspension during parsing. Along with the suspended fibers they contain, some states that are unnecessary may be discarded before the parsing process completes.
- Only the reduction instructions access variables from previous states.
- The state index *sid* of a fiber is always equal to or higher than the lowest value *sid* in the fiber queue. In other words, new fibers are always created with monotonically increasing state indices.

Based on these observations, the following optimisations can be performed:

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

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

Such a garbage collector discards all states with the fibers they contain that are not part of any parse rule/active reduction that can be traced back to the starting non-terminal symbol. To reduce the influence of the performance of the garbage collector on the parsing process, the garbage collector could be run every n parsed terminal symbols.

4.9.2 Eliminating dynamic non-terminal call indirection

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

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

If the active domain set for a specific call is known during compile time, then the instruction pointers for target rule entry points and reduction tags can be computed during compile time. Thus, it is no longer necessary to perform dynamic rule lookup and filtering during parse time. Therefore, the dynamic instructions `i_call_dyn` and `i_match_dyn` can be replaced with the corresponding static instructions: `i_call` and `i_match_sym`. Furthermore, `i_call` ip_{target}, n is a new instruction that invokes a non-terminal rule with the entry point ip_{target} and n arguments.

4.9.3 On-demand instruction subset construction

Importance of subset construction

The EVM is based on the Earley parser and therefore inherits some of its flaws. One of the main reasons the Earley parser in its original form is not used for parsing programming languages is its inefficiency. One of the common tasks of parsing programming languages is parsing expressions. Even older programming languages (such as C++) have huge operator hierarchies with many precedence levels. For example, the C++ language has a total of 44 distinct operators:

- 12 arithmetic operators,
- 6 comparison operators,
- 3 logical operators.
- 6 bitwise operators,
- 10 compound operators, and

- 7 member and pointer operators.

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

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

The way the EVM currently operates can be similar to a non-deterministic finite automaton; just like an NFA can be in multiple states at the same time, the EVM can execute multiple fibers at the same time. However, it is well known that any NFA can be converted into a DFA by applying the process known as subset construction. The faster Earley parser [20] or efficient Earley parser with regular right-hand sides [15] are both based on this algorithm. By applying such parsing algorithms to parse C++, it would no longer take 50 distinct fibers (or items in the Earley parser case) to parse a single expression. Instead, all 50 grammar rules could be merged into one optimised rule.

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

Instruction ε-closures

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

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

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

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

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

There are two possible actions:

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

The actions to be executed for each instruction are provided in Table 19. Furthermore, *ip* and *ads* refer to the instruction pointer and active domain set of the current entry, respectively, and $entries(A, ads)$ refers to the set of rule entry points for non-terminal *A* with the currently active domain set *ads*.

Table 19: Rules for computing instruction closures

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

Merging instruction ε-closures

The goal of merging instruction ε-closures is two-fold: the merger of similar instructions to avoid duplicate computation and the elimination of dynamic elements that can reduce parsing performance. Because of the second goal, dynamic instructions like `i_call_dyn` and `i_match_dyn` are replaced with their static counterparts. In general, all instructions are merged based on the *instruction merger key*. If two instructions share the same instruction merger key, then they can be merged into a single instruction. The instruction merger keys can be derived from the rules provided in Table 20.

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

Table 20: Rules for computing instruction merger keys

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

- `i_match_chars` *table* instructions are merged by merging their jump tables. Transitions that share the same character are merged by computing their ε-closure and optimising it. The resulting instruction is a `i_match_chars`.
- `i_match_dyn` and `i_match_syms` instructions are merged into a single `i_match_syms`. The merger process works similarly to the merger of `i_match_chars`. The instructions are merged by merging their jump tables. In the case of `i_match_dyn` (which has no jump table argument), jump tables are computed based on the `i_match_dyn` operands and active domain set. Then, transitions that share the same non-terminal symbol are merged by computing and optimising their ε-closure.
- `i_call` and `i_call_dyn` are merged into a single `i_call_opt` or `i_call` instruction. This is done by computing the ε-closure of the entry points of the target non-terminal. If the optimised instruction sequence for the resulting ε-closure already exists, then a direct call with `i_call` to that instruction sequence is generated. Otherwise, `i_call_opt` *closure* is generated. Moreover, *closure* refers to the target ε-closure. This instruction is used to avoid the subset construction of the entire grammar module. The optimised (subset constructed) version for the closure is generated only upon executing `i_call_opt`, thus making the instruction subset construction process only run on demand.
- `i_reduce` *A* instructions are merged simply based on the reduction non-terminal into a single `i_reduce` instruction. This way, duplicate reduc-

tions with the same non-terminal are eliminated.

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

Once instructions are merged, they can be output to a target grammar module. The resulting instructions are output in a specific order:

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

An example of an optimised (subset constructed) instruction sequence is provided in Table 21. The resulting instruction sequence is longer but more deterministic. For example, at offset 16 of the optimised instruction sequence, prefixes for addition and multiplication have been merged successfully. It will take a single instruction at offset 16 to match the binary operator, at which point parsing diverges based on the matched operator.

4.10 Conclusions

The following conclusions were reached:

- The EVM parser satisfiers all REP language analysis functional requirements. As a result, it is suitable for REP language parsing.
- Terminal symbol matching in EVM is not efficient because at least one instruction must be interpreted and one fiber must be registered in the ETS to parse a single terminal character.
- To further improve EVM performance, additional ambiguity elimination methods must be implemented.
- Information contained in ETS is superfluous and may be computed from other EVM data structures (mainly, the list of suspended tasks). As a re-

Table 21: Subset construction example

Grammar	Compiled grammar	Optimized grammar

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

sult, this information duplication should be eliminated to further improve EVM performance.

5 IMPLEMENTATION OF SCANNERLESS EARLY VIRTUAL MACHINE

5.1 Scannerless Earley Virtual Machine

5.1.1 Flaws of the original Earley virtual machine

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

- **Recognised grammar class**. Different parsing methods can recognise different classes of input languages. For example, LR(0) parsers can only recognise LR(0) grammars. More generalised methods, such as the GLR [29] can recognise a wider class of input languages (all context-free languages in the case of the GLR). However, even then, it is possible that such parsing methods may not be able to recognise all programming languages used in practice because not all programming languages are context-free languages. The size of the recognised grammar class determines how many real-world computer languages can be recognised by this parser.

- **Expressiveness of the grammar language**. Parser development typically starts with the creation of a target language grammar. This grammar is written in a specific grammar description language, which is read by a parser or parser generator, which then is responsible for generating and/or configuring the parser so it can recognise 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 clearer and more compact method to describe operator precedence. The existence of such operator precedence declarations does not permit parsing additional languages but merely allows expressing the already recognisable languages in a more intentional fashion. As a result, the greater expressiveness of the grammar language makes the development of new grammars easier.

- **Performance**. The performance of the parsing method and its imple-

mentation is one of the primary factors determining whether or not such a parser is suitable for parsing real-world computer languages. Generalised 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. Not a single scannerless parser is used to parse any high-profile programming language. For example, `GCC`, `Clang`, and Lua use hand-written recursive descent parsers [6], and the MRI Ruby implementation uses LALR(1) `Bison`, whereas CPython uses a custom bottom-up tokeniser and parser combination.

- **Support for scannerless parsing.** This determines whether or not two grammars can be effortlessly combined. If two grammars can be combined during runtime, then such a parser can be used to parse extensible languages. Additionally, scannerless parsers must provide additional features to eliminate character-level ambiguity.
- **Presented description of errors**. Any parser used in practice should be able to provide informative feedback when a parsing error occurs, so the user of such a parser can correct the errors in the parser input. The more descriptive and informative the error messages are, the less time the user needs to determine why the error occurred and how to fix it.

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

- The original research prototype for the EVM was implemented in the Ruby programming language. Because Ruby is interpreted, any parser written in this language will be orders of magnitude slower due to the overhead of the interpreter. Thus, a new EVM implementation is needed if the performance of the EVM is to be compared to other parsing methods.
- While the EVM was created with scannerless parsing in mind, one key issue will severely limit the performance of the EVM, even if the EVM is implemented in a non-interpreted language. During parsing, the EVM creates a state for each terminal input symbol. This means that, to parse the input of length n, $n * size_of(State)$ bytes of memory are needed to represent the parser states. These states will then contain additional

dynamically allocated structures, such as the list of suspended tasks and reductions and the trace.

- The EVM is a scannerless parser; therefore, a method is needed to disambiguate the identifiers from keywords in languages that have such grammar elements. Furthermore, a way to disambiguate operators that consist of more than one character is needed (e.g. the 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 the resulting parse forest [4], 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.

- Regarding trace simplification, in the current EVM version, the EVM records previous parse positions in a set called trace. This trace contains a set of complete snapshots of fiber states at various positions during parsing. Consequently, a significant overlap of information is stored in *trace* along with the list of suspended tasks and reductions in each state.

5.1.2 Overview of the internal structure of the scannerless Earley virtual machine

In this section, we provide a description of the internal structure of the scannerless EVM (SEVM). The SEVM is a further modification of the EVM that attempts to improve its performance and extend the parser to recognise real-world computer languages.

The SEVM consists of the following primary components:

- The **grammar compiler** translates the textual representation of input grammar into a medium-level intermediate representation (MIR). It also detects any syntax or semantic errors of the input grammar.

- The **optimiser** is responsible for merging grammar rules in MIR form. The optimiser takes a list of grammar rules to be merged in MIR form and produces a combined MIR that implements all of the merged grammar rules, but with their prefixes merged.

- The **resolver** is responsible for invoking the optimiser and translating the resulting MIR into machine code.
- The **runtime** is responsible for coordinating the execution of the parser.

The SEVM consists of the following data structures:

- The **MIR tree** is an AST with intermediate language representation of SEVM grammars.
- The **chart** is the primary data structure of the parser runtime. It closely corresponds to the EVM state list. The chart is a sparse index map from the input positions to the **chart entries**.
- The **chart entry** stores all information about the parsing progress at a specific input position. Each chart entry contains the following: a reduction list *reductions*, a list of suspended tasks *suspended*, a list of currently active tasks *running*, and an activity indicator *queued*.
- The **reduction** component contains information about a single reduction: *kind*, *reduce_id*, *length*, and *tree_id*. The reduction *kind* determines whether the current reduction is an *accept* or *reject* reduction. This information is used to implement negative reductions. The reduction index *reduce_id* determines the non-terminal symbol associated with the reduction. The reduction *length* indicates the reduction length in bytes. Finally, *tree_id* stores the index of the resulting parse node.
- The **task** directly corresponds to the 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*, and *grammar_id*. Semantically, a task can be viewed as a function closure in other programming languages. The *state_id* determines the current state of the task. This value is used to implement task suspension and resumption. In addition, *origin* is the index of the chart entry in which this task was initially created. In other words, it represents the starting position of the non-terminal that this task will parse. Moreover, *position* indicates the current parsing position. It is an offset from the beginning of the parse input. The *tree_id* is the node index of the partially constructed parse tree so far, and *grammar_id* stores the active grammar index.

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

- The **resume** component 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 the 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 Section 5.5 for more about eliminating exponential complexity.

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

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

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

- The **call specifier** represents a set of grammar rules that are meant to be invoked during parsing. The optimiser uses the call specifier and the grammar MIR as inputs to produce an optimised MIR in which multiple

rules are merged. Internally, the call specifier is a sequence of *match_id* and minimum precedence *min_prec* value pairs.

- The **grammar** component stores a mapping between the reduce and match indices. Each grammar has a unique index.
- The **parse tree** stores the automatically constructed parse tree during parsing. Section 5.6 details how parse trees are encoded.
- The **call stack** is a stack of chart indices that represents the call stack of the parser.
- The **DFA** is a data structure that encodes a deterministic finite automata, which is used to parse non-ambiguous intervals of input languages.

5.2 Improving Grammar Expressiveness

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

5.2.1 Abstract grammar rules

Abstract grammar rules are a new type of grammar rule that has several purposes:

- They provide an alternative method to declare production rules, such as $Z = A|B|C$.
- They provide an extension point for extending grammars. The original EVM grammar language provided no grammar construct to specify extension points. The EVM only provided low-level infrastructure to implement such extension points but provided no metalanguage at the grammar level to specify such extension points.

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

```
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"; }
```

Figure 25: Abstract grammar rule example

calls itself via the abstract rule to determine whether this rule should be part of the call.

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

Figure 25 shows an example of grammar that uses an abstract grammar rule to implement an expression hierarchy, which contains $+$ and $*$ operators with the appropriate precedence.

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

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

- Increased performance. Abstract grammar rules do not perform reductions and are matched directly against the callee *match_ids*.
- Rule precedence. Abstract rules provide a simpler method to specify operator hierarchies.

5.2.2 Named precedence groups

The named precedence group is a grammar feature closely related to abstract rules. Named precedence groups provide a method to call an abstract rule with a custom precedence value. Consider the ANSI C grammar fragment provided in Fig. 26.

The expressions that have the highest precedence in the ANSI C language are *primary expressions*. Below them are *postfix expressions* with slightly reduced precedence. *Unary expressions* have even lower precedence followed by *cast expressions*, and so on. 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 the ANSI C case, `unary_expression` refers to `cast_expression`, which has lower precedence. Similar situations can be observed in the grouping expression of `primary_expression`, which refers to `expression`, which is the top of the expression hierarchy.

To be able to represent such expression hierarchies with abstract syntax rules, there must 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 a specific precedence level (value) of an abstract grammar rule.

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

The grammar fragment in Fig. 26 may be rewritten in `north` as shown in Fig. 27. In `north`, the ANSI C expression is an abstract grammar rule. Different precedence levels are just named precedence groups (`primary_expression`, `postfix_expression`, `unary_expression`, and `cast_expression`).

There are several types of calls in `north`:

- Concrete rule calls. These are in the form of a `unary_operator`, where the `unary_operator` refers to a concrete rule.
- Abstract rule calls (non-associative). These are in the form of an `expression` , where the `expression` refers to an abstract rule. If the call is directly recursive from the callee with precedence *prec*, then the same abstract

```
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 26: A simplified fragment of C99 grammar

```
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 27: A fragment of C99 grammar rewritten in North

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 the form of `expression!`. They are statically ensured to be directly recursive. If the callee has the precedence value *prec*, then an abstract rule with the same precedence value *prec* is invoked.
- Abstract rule calls with explicit precedence value. These are in the form of `expression!prec`, where *prec* is an integer value. In such calls, the callee precedence level (if it exists) is ignored, and an abstract rule with precedence *prec* is invoked.
- Named precedence group calls. These are in the form of `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 of the referenced named group with the appropriate precedence level.

Because it is now 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 the existing precedence levels with the `part_of` attribute, or
- adding entirely new levels (that may exist 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 maximise the extensibility of the implemented grammar.

5.2.3 Dominating terminals

Multi-line comments in the ANSI C programming language start with the characters `/*` and terminate with `*/`. In North, such comments may be parsed with a rule shown in Fig. 28. However, such a simple rule is not entirely correct. The comment terminator `*/` will be ambiguously matched both as a comment terminator and as a comment body, forking the rest of the input into

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

Figure 28: A `north` grammar rule for parsing ANSI C multi-line comments (Attempt 1)

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

Figure 29: A `north` grammar rule for parsing ANSI C multi-line comments (Attempt 2)

two distinct paths: one where the comment never terminates and another where `*/` was interpreted as a comment terminator.

To avoid this ambiguity, the rule may be redefined as shown in Fig. 29. In this case, the character sequence `*/` is excluded from the comment body by first allowing the comment body to only contain non-* characters (`r"^*"`), and then requiring that character * must not be followed by a slash (`*`, `r"^/"`). Such a rule correctly and unambiguously parses C comments; however, it is not as clear as the initial rule shown in Fig. 28.

It would be ideal if a method existed to specify that the slash in the comment terminator `*/` would take precedence over the one possibly found in the comment body. This would enable retaining the correct and non-ambiguous semantics of the grammar rule in Fig. 29 while keeping the simpler definition in Fig. 28.

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

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

Figure 30: A `north` grammar rule for parsing ANSI C multi-line comments (Attempt 3)

```
#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
```

Figure 31: Unoptimised MIR for the ANSI C multi-line comment rule

in Fig. 30.

The primary reason for implementing dominating terminals is that they can simplify the definition of various grammar rules without any reduction of parsing performance. Such behaviour may be implemented statically in the SEVM optimiser. The rule shown in Fig. 30 may be translated to an unoptimised MIR graph, as shown in Fig. 31.

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

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

By modifying the SEVM optimiser implementation and annotating instruction `#6` with the domination flag `DOM` (that was added as a result of the `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 only the instruction nodes with the highest priority, which becomes `[DOM #7]` after filtering. Such an implementation of dominating terminals is not only simple and effective but also enables using dominating symbols to disambiguate tokens at the character level, as described in Section 5.3.6.

5.3 Ambiguity Elimination

Even though the 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 [4]to 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 performance on invalid parse paths.
- It increases the overall parser complexity because additional code is needed to perform ambiguity elimination at the parse-tree level.

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

5.3.1 Negative reductions

Negative reductions are an adaptation of the scannerless GLR's reject reductions [9] for the SEVM. In the scannerless GLR family of parsers, the reject reductions/productions are used to disambiguate the reserved keywords from the identifiers.

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

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 second condition implements reduction priorities: if a reduction with higher priority already exists, then the new, lower priority reduction is rejected. For this

Table 22: Reduction kind values

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

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

Figure 32: A grammar rule that defines an identifier followed by a space

approach to work, all code that implements higher priority reductions must be executed first; otherwise, it is possible for lower reductions to 'slip through'. If this does 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.

To compare reduction kinds, each reduction kind is assigned a unique integer value (see Table 22). Then, the reduction kinds are compared using these integer values.

From the user's perspective, negative reductions can be defined in grammars with the `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 Fig. 32 defines a rule for parsing identifiers, which may be composed of lowercase or uppercase 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 the keyword `if` (see Fig. 33 for the MIR of the same grammar rule). By adding more complex grammar expressions to the `reject` statement, it is possible to disambiguate several keywords or even more complex grammar expressions from identifiers.

```
#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
```

Figure 33: Unoptimised medium-level intermediate representation for a grammar rule that defines an identifier

5.3.2 Strict execution ordering in scannerless Earley virtual machine runtime

The EVM, much like the original Earley parser [8], performs mostly breadth-first searches (with the exception of when fiber priorities are involved, which are used primarily to implement a regular lookahead). Other than this, the rest of the execution of the parser is unordered: the `i_fork` instruction for creating duplicate fibers queues the fiber for execution but in arbitrary order. The `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 the EVM, it is no longer suitable for the SEVM. The SEVM must ensure that the reductions with higher priority execute first to avoid reduction slips. To that end, the entire execution model for the SEVM must be shifted to depth-first execution:

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

Internally, this is implemented using a two-layer stack:

1. The 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 the appropriate secondary stack. Then, the index of that chart entry is added to the top of the primary call stack if the index does not already exist in the primary stack. To avoid having to perform a linear search in the primary stack to determine whether 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 comprises the following steps:

1. Locate the currently active chart entry by retrieving its chart index from the top of the primary call stack. If the primary stack is empty, then the parser terminates.

2. If the secondary stack is empty, then attempt to populate it by *failing* the current entry (see Section 5.3.3). If failing yields no new tasks, then remove the top element from the 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 simulates how the call stacks work in traditional imperative programming languages but also adds the 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 re-

suming 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 second one causes the current task (caller) to be suspended, effectively yielding execution control to the callee. If the callee creates new subtasks (e.g. 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 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 the 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 the `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 the `NORMAL` reductions, thus fulfilling the strict execution ordering requirement for negative reduction implementation.

5.3.3 Negative matches

When applied to rule calls, the strict execution ordering 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 are fully complete, it is possible to determine whether a particular non-terminal *failed* to match.

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

In the 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:

- All active tasks from the current chart entry have been completed (because the secondary stack is empty).
- The current chart entry is active (its index is stored at the top of the 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 (its negative match specifier is not empty), then it goes to the next step. Otherwise, it discards the currently suspended task because all of its subtasks have failed. Then, it goes to Step 1.
3. The last entry MS of the negative match specifier of task T is selected.
4. The last entry MS is matched against the list of all reductions of CE. If at least one positive match exists, the suspended task T has been resumed at least once and a negative match cannot be performed. As a result, MS is removed from the negative match specifier of T. Then, it continues to Step 2; otherwise, it proceeds to the next step.
5. If no positive match for MS was found, 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 the secondary stack of E. Further chart entry failure is aborted.

In essence, during *chart entry failure*, each suspended task from the 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 continues until at least one task is resumed or the list of suspended tasks in the current chart entry is 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. In addition, the order of the resume conditions in the negative match specifiers is also important. Negative matches in the SEVM/`north` implementation are not directly accessible to the user, but they are used to implement greedy non-terminal repetition operators.

5.3.4 Greedy non-terminal repetition

Greedy repetition in the SEVM is accessible via `parse_g` statements, which are similar to regular `parse` statements, but certain operations within the provided grammar expression are replaced with greedy equivalents. Greedy non-terminal repetition is implemented using negative matches. The 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 a child of one of the repetition operators (`?`, `*`, or `+`), then the call is compiled differently. The `CtlMatchSym` instruction now contains the callees' *match_id* in both the positive and negative parts of the 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 it 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 the SEVM. Thus, greedy non-terminal repetitions should be used sparingly to avoid interfering with the optimiser's subset construction.

5.3.5 Strict execution ordering in the scannerless Earley virtual machine optimiser

So far, we have 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. The SEVM relies heavily on its optimiser, which can merge multiple grammar rules by performing a variation of subset construction on MIR graphs (the algorithm inspired by the efficient Earley parser [15]). In this chapter, the description is given regarding how the strict execution ordering is preserved during the optimizer's subset construction.

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

1. Add the instruction pointers to be merged into an initial set S_I.
2. Add this set into resolution queue Q.
3. Remove one set S_0 from the Q.
4. Find the 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 its entry in subset construction cache C.
6. Store the mapping S_1 to ip_{end} in 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 the 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 [21], the one used for the EVM uses sets to represent instruction ε-closures and a queue to control the order of individual subset construction steps.

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

Rules for constructing ε-closures in the SEVM are given in Table 23. 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 is visited according to the rules provided in Table 23.

Table 23: Rules for computing scannerless Earley virtual machine 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`

  - 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 its implementations are visited.

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

Once an ε-closure is obtained, its instructions are merged much like in the original EVM. One key difference in the SEVM subset construction is that the initial merge sequence may only contain other concrete rules. In other words, during the 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 optimised rules are entirely independent and isolated from any other code.

The primary advantage of this is that the calls that start at the 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 the original EVM, the generated rule suffixes were reused possibly several times across the entire grammar. In the SEVM, the reuse may only

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

Figure 34: A grammar for parsing identifiers and keywords

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

Figure 35: A modified grammar for parsing identifiers and keywords

happen internally within one generated rule. In other words, if the optimiser constructs a merged rule for parsing `A | B` and later for `A | C`, then no code between these two generated rules will be shared, whereas the 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 DFAs are cached and de-duplicated, as described Section in 5.4.3.

5.3.6 Token-level ambiguity elimination

An unexpected side effect of implementing dominating terminals is that these 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, and 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.

Consider the grammar shown in Fig. 34. It defines two grammar rules: one for parsing identifiers and another for parsing the reserved keyword `self`, both of which must be followed by a space. If these non-terminals are used in a grammar expression, such as `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. 35. In this case, the terminating whitespace symbol (in practice an alphanumerical boundary symbol is typically used instead) is changed to be dominating with the `dom_g` specifier. As a result, when the `ident | kw_self` expression is encountered, the `ident` and `kw_self` rules are merged. The subset construction continues until the terminating symbol is encountered, at which point the lower

```
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, ...);
}
```

Figure 36: A grammar that uses token groups to disambiguate keywords from identifiers

Table 24: 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

priority (non-dominating) terminating symbol for `ident` is filtered-out, allowing only `kw_self` reduction to occur, thus eliminating the ambiguity.

This scenario 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, the SEVM can be extended to enforce this condition.

For this reason, the `#[token_group]` attribute is introduced in the `north` implementation, 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 the `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 it to effectively disambiguate keywords from identifiers (see Fig. 36).

Comparing this approach to negative reductions reveals significant performance gains for parsing reserved keywords. Table 24 shows the performance

cost (in terms of total reductions needed) to recognise disambiguated keywords and identifiers for both of the described approaches. Disambiguating keywords from identifiers with negative reductions requires three separate reductions to be performed:

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

When using token groups (in combination with dominating terminals), only one reduction is needed. Another positive effect of token groups is that it results in a significantly lower amount of generated code for the following reasons:

- No separate parse path for matching keywords and performing negative reductions is needed.
- Token group disambiguation can happen as part of the DFA extraction process (see Section 5.4.3 for more), which reuses matching transition tables across different rules.
- Replacing all direct calls to individual keyword rules into corresponding token groups results in a lower number of unique call specifiers, which means that fewer optimised 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 recognising every keyword defined in a token group).

5.4 Parser Optimisations

5.4.1 Profiling the Earley virtual machine

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

- Built-in performance counters. During various steps of the `north` execution, execution times for the most important components are measured

and stored. This information is then optionally displayed after the execution to the user.

- The `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 more slowly, but the additional instrumentation allows it to obtain detailed metrics about the entire process of the program execution.
- The `massif` heap profiler. This is a tool that allows it to measure and observe the changes in overall memory usage.
- The `bench_parsers` tool. This was developed as part of the `north` implementation and allows it to compare the performance of different parser implementations with great accuracy.

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

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

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

5.4.2 Just-in-time grammar compilation

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

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

- The MIR instructions are organised into basic blocks. Each basic block contains zero or more statement instructions and must terminate exactly by one control instruction. All operations that affect the flow of the execution are control instructions. This approach somewhat mimics the LLVM design, where the instructions are also organised into basic blocks.
- The MIR rules comprise basic blocks instead of instructions. This matches the LLVM functions, which comprise LLVM basic blocks.

Each task is compiled into a single native function, which takes the parser context and a pointer to the current task as parameters. This function is referred to as the `resume` or task resumption function. The `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 the execution to the appropriate state based on the task $state_id$ value. The $state_id$ values correspond to the matching MIR basic block indices to ease the debugging process. Each SEVM basic block is translated by translating the individual instructions of that basic block directly into the LLVM IR. Some MIR instructions can be translated into several LLVM IR instructions or even several LLVM IR basic blocks.

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

The `CtlMatchChar` instructions are compiled into several LLVM instructions:

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

a conditional jump is made as a result.

The `CtlMatchClass` instruction is compiled similarly: the first four steps are the same as in `CtlMatchChar`, but the input character is compared using an unrolled binary search: each bound of search space is compared with a pair of `icmp` and `br` instructions. The `CtlReduce` instruction is translated into a call to an appropriate runtime function and unconditional jump to a target location. `CtlMatchSym` is translated into a call instruction to an appropriate runtime function, which takes ownership of the current task and (potentially) adds it to the list of suspended tasks, and the `ret` instruction, which stops the current task. `CtlStop` is translated into a single `ret`, which terminates the current task.

5.4.3 Deterministic finite automata extraction

Terminal symbol matching shortcomings

In the current version of SEVM, the terminals are matched with the `CtlMatchChar` and `CtlMatchClass` instructions. `CtlMatchChar` can match a single input character against another character, whereas `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 are replaced with the `CtlMatchClass` during the subset construction, which is later translated into LLVM IR. The compiled `CtlMatchClass` performs a binary search to match the input character against several possible alternatives. As a result, the resulting LLVM IR code contains at least the following:

- three basic blocks;
- three comparison instructions: one to test for the end-of-stream, one to test the lower bound, and one to test the upper bound;
- three conditional jumps;
- one addition that is used to increase the position value of the current task;
- two memory loads used to load the current position and the character at the current position; and
- one memory store used to store the updated position of the current task.

This means that matching a single input character, when translating the SEVM MIR to LLVM IR requires a significant number 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 characters, and each operator contains 1.5 characters. All keywords and operators occupy 250 characters when concatenated. 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 optimised rule. Because of this, each distinct operator precedence level would be optimised at least once, each time including every keyword of the grammar, resulting in massive amounts of generated code.

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

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

All of these issues described in this chapter can be solved (to an extent) by comparing the current method for matching terminals in the SEVM with traditional lexers. During subset construction, the SEVM optimiser essentially constructs an embedded lexer each time a terminal symbol (or terminal symbol sequence) is to be matched. By isolating these deterministic fragments of

```
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 37: A `north` grammar for matching three keywords

sequences of instructions, it would be possible to extract them and to perform terminal symbol matching in a lexer-like environment, isolated from the rest of the virtual machine. We call this approach of separating terminal matching *deterministic finite automata extraction* (*DFA extraction*).

Simple deterministic finite automata extraction

Consider the grammar shown in Fig. 37. It defines three keywords: `self` , `static`, and `struct`. During subset construction, shared prefixes of these keywords are merged, and the MIR shown in Fig. 38 is produced. This MIR may also be visualised as a deterministic finite automaton, as shown in Fig. 39, which captures the essence of the DFA extraction method. The segments of the deterministic source MIR are extracted into a separate DFA, which is 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 the success or failure of the DFA match result.

An optimised MIR for the abstract rule `kw` with DFA extraction enabled is shown in Fig. 40, and the corresponding extracted DFA is shown in Fig. 41. The `CtlExecDFA` instruction takes two operands: the DFA to be executed and the transition table that pairs the result of the DFA with the target *state_id* of the task. Note the significant reduction of basic blocks in this version of the optimised MIR.

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

It is also important to note that the states in the SEVM DFA are classified

```
 #0: CtlMatchClass 's' => #1
 #1: CtlMatchClass 'e' => #2, 't' => #7
 #2: CtlMatchClass 'l' => #3
 #3: CtlMatchClass 'f' => #4
 #4: CtlMatchClass ' ' => #5
 #5: StmtRewind 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: StmtRewind 1
     StmtReduce REDUCE_ID(:kw_static), NORMAL
     CtlStop
#14: CtlMatchClass 'u' => #15
#15: CtlMatchClass 'c' => #16
#16: CtlMatchClass 't' => #17
#17: CtlMatchClass ' ' => #18
#18: StmtRewind 1
     StmtReduce REDUCE_ID(:kw_struct), NORMAL
     CtlStop
```

Figure 38: An optimised medium-level intermediate representation for matching three keywords

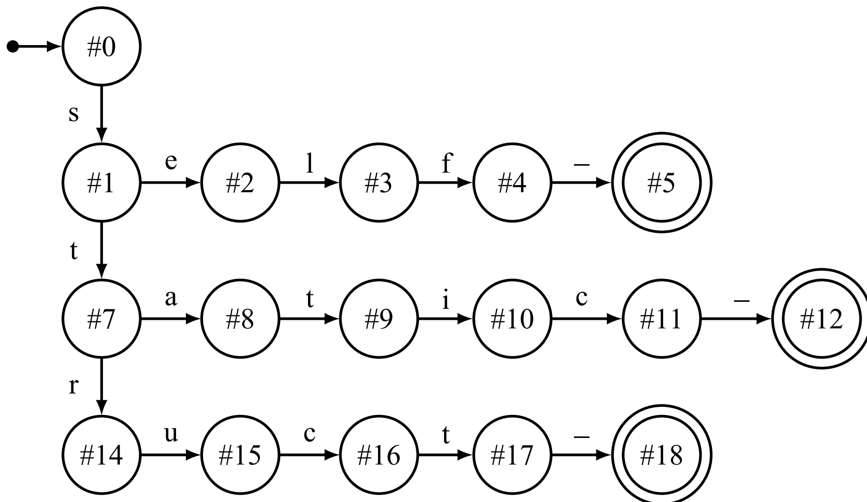

Figure 39: Traditional deterministic finite automata for matching three keywords

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

Figure 40: Optimised medium-level intermediate representation for matching three keywords (with the deterministic finite automata extraction enabled)

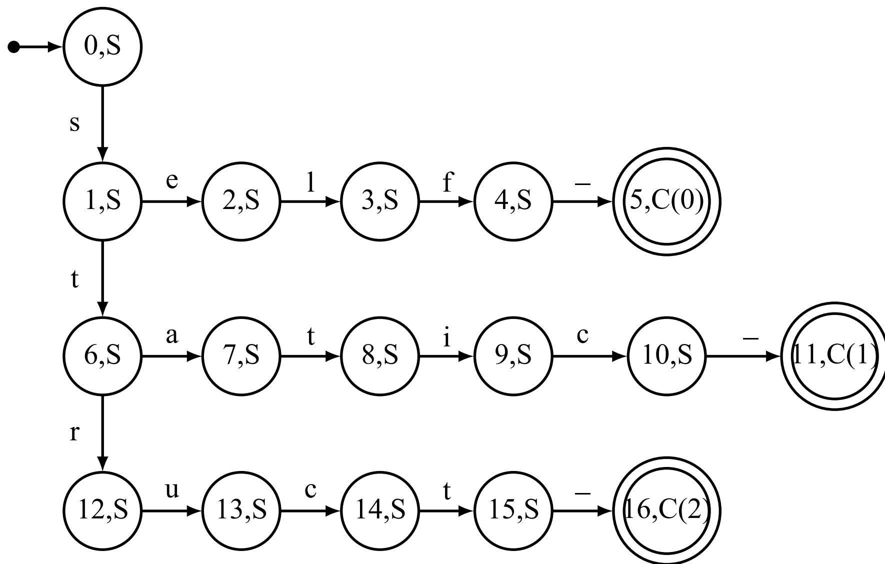

Figure 41: Scannerless Earley virtual machine deterministic finite automata for matching three keywords

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. Typically, each DFA contains exactly one fail state, which is reachable from all other states with unexpected terminals. They are not shown in any of the DFA visualisations because there would be an edge from each *shift* state to the fail state with all other characters from the ASCII range 0 to 255.
- *Complete* (C) states terminate the DFA simulation with a given result. The result is a number that is used in the MIR to transfer the control of the execution.
- *Lookahead* (L) states are used to implement a lookahead (see Section 5.4.3 for more information).

Furthermore, shift state transition tables are split into two parts: a transition index table and a transition state table. The transition index table stores the indices of the transition state table, which stores the actual destination state indices. This two-layer transition-table approach allows it to de-duplicate and reuse transition index tables. All transition index tables have 256 entries (1 byte each), where one entry is reserved for each possible input character. The size of the transition state table is variable and corresponds to the number of unique transition destinations from a specific state. This significantly reduces the size of the generated DFAs because the largest parts of each DFA can be reused. The largest DFA used to parse the Rust language is composed of 237 distinct states, 136 of which are shift states. Of these, ≈95% are reused in at least one other DFA.

Dominating terminals in extracted deterministic finite automata

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

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

Figure 42: A `north` grammar for parsing `.` and `...` operators

Greedy tokens

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

The way the extracted DFA works already resembles traditional lexers. By extending this analogy further, we can implement *greedy tokens*. In the case in which multiple token matches are available (such as `&` and `&&`), we select the longest. In the SEVM, this can be done by adding an additional indicator to the `CtlMatchChar` and `CtlMatchClass` instructions to specify the longest match preference. At the grammar level, the `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 the ε-closure when building DFAs. If all members of the constructed ε-closure are `CtlMatchChar` or `CtlMatchClass` instructions, then they are merged into a single shift state in the DFA. If additional instructions (such as `StmtCallRuleDyn`, `StmtReduce`, or `CtlReduce`) are used, then a complete state is generated instead, which hands the execution control back to the SEVM, which will split the execution (typically) into two different paths: one task with another DFA that performs character matching and one that contains other instructions.

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

An example grammar for parsing and disambiguating the operators `.` and

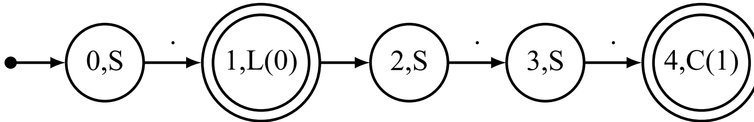

Figure 43: SEVM DFA for the triple dot grammar

. . . is shown in Fig. 42. During subset construction, terminal symbol matching will be extracted into the DFA shown in Fig. 43. 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.

Several levels of lookahead states can exist, allowing it to disambiguate complex tokens. For example, greedy tokens are used in the grammar of the Rust programming language to disambiguate *all* of Rust's tokens:

- Raw string literals `r"text"` are disambiguated from identifier `r` and the text literal `"text"` sequence.
- Operators of varying lengths are disambiguated (`.`, `..`, `...`, `..=`, `=`, etc.).
- In combination with dominating terminals, base-16 integer literals, such as `0x1234ABCD`, are disambiguated from base-10 integers with the suffix (`10i32`).

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

5.4.4 Partially incorporated reductions

Reduction incorporated parsers

The LR family of parsers [18] uses a stack to track the execution of the overall parsing process. The stack contains state indices that 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, the reduce action is computationally more expensive.

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

1. The top N elements from the stack are removed.
2. The newly exposed top element is used to determine the current parser state.
3. The transition table, current parser state, and reduced non-terminal are 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 considered bound by the amount of *stack activity* needed to parse the input. As such, numerous approaches to reduce the overall stack activity during parsing have been used to increase parsing throughput. Typically, left recursion is favoured over right recursion because it leads to lower stack growth.

A more involved and recent approach for reducing stack activity is using reduction incorporated parsers [23]. At the cost of significantly increasing the number of parser states (and thus the transition table), it is possible to record the target state index as part of the 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 the reduction sequence.

The cost of a reduction in the scannerless Earley virtual machine

Just like in LR parsers, reductions in the SEVM are computationally 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.

```
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; }
```

Figure 44: A simple `north` grammar

3. The new reduction is added to the reduction list of the current chart entry.

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

Reduction incorporation in SEVM

Optimised rules in the SEVM are entirely defined by a call specifier and the currently used grammar. Consider the grammar shown in Fig. 44. When optimising a main rule, the direct call to rule `ABCD` is replaced with a dynamic call with call specifier `[REDUCE_ID(:ABCD), 0]`. Then, the optimiser uses this call specifier to drive the subset construction and generate the optimised version of `ABCD`.

Because `ABCD` starts with both `AB` and `CD` rules, both of these rules are merged into an optimised version of `ABCD`. Continuing this process recursively, the optimiser merges `ABCD`, `AB`, `A`, `B`, `CD`, `C`, and `D` rules in this order. Eventually, the MIR code, which is shown in Fig. 45, is produced (to make the MIR more readable, DFA extraction was disabled).

Consider the following step sequence, which is executed for parsing character `a`:

1. Task 0 starts the execution in basic block `#0`.

2. Task 0 is forked (queued) into Task 1 with State `#2`.

3. Task 0 is suspended in `#1`.

4. Task 1 executes `#2` and matches character `a`, which transfers control to

```
#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
```

Figure 45: Optimised medium-level intermediate representation for the ABCD grammar rule

#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.

Tasks 2 and 3 were created only to perform a single reduction, after which both were terminated. Another important observation 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 a reduction that occurs within an optimised MIR but is not a part of the call specifier. In the current example, reductions for A, B, C, D, AB, and CD are internal. Reductions that are part of the call specifier are

```
#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
```

Figure 46: Optimised medium-level intermediate representation for the `ABCD` grammar rule (with partial reduction incorporation)

called *external reductions* because the effect of the reduction will be transferred beyond the current optimised rule.

All control transfers for internal reductions during the reduction process may be resolved statically. By definition, the effect of internal reductions does not extend beyond the current rule. The rule that performs an internal reduction must have also been invoked from the same optimised (merged) rule. Furthermore, only rules that are part of the rule prefix are merged into an optimised rule. Because of this, the `StmtReduce` of internal reductions will always match the match specifier of the `CtlMatchSym` instruction, which will always be located at the start of the optimised MIR.

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

Figure 46 shows the optimised MIR for `ABCD`, but with partial reduction incorporation enabled. With this MIR, the 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 four instances). Furthermore, calls that are part of the prefix no longer require `CtlMatchSym` because the control transfer of internal reductions is handled directly using `CtlReduceShort`. As a result, the reduction incorporated version of `ABCD` performs three fewer reductions and one fewer task suspension. This optimisation also yields significant performance gains in real-world programming languages, as shown in Section 6.6.3.

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

5.4.5 Garbage collection

The purpose of the garbage collector in the SEVM, just like the EVM, is to remove information that is no longer needed from the memory so it may be reused again. Because of the changes in the SEVM structure, the original garbage collector of the EVM is no longer suitable. The memory in the SEVM is freed up by removing potentially unneeded chart entries from the parser chart. The condition for removing entries from the chart is based on a heuristic and thus may remove entries that may still be needed later during parsing. Ideally, such a situation would not occur often, and if it did, these chart entries would have to be recreated by reparsing fragments of the 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 thus must not be freed.
- Parsing is typically done sequentially with relatively few significant jumps due to ambiguities or backtracking.
- Ambiguities and backtracking are typically localised.

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 do

not contain any active tasks are marked for removal during garbage collection.

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

The desired GC_{iter} is chosen by manually inspecting the parse times of the sample inputs and setting it to a value higher than GC_{min} (typically $3 * GC_{min}$), where 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 that partially disables the garbage collector. In this mode, the garbage collector only marks them as removed instead of removing those entries. If an entry with a 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 a strategy of garbage collection may not be optimal (and may lead to significant slowdowns in worst-case ambiguity scenarios); however, it works well when used with real-world programming language grammars: the heap usage and processor time profiling reveal that the parser runtime uses only minor amounts of memory, while taking an insignificant amount of time to execute (less than 5% of the total execution time with ANSI C and Rust grammar tests). With the garbage collector enabled, the parse tree becomes the largest memory consumer in `north`, followed by the index map of the chart.

5.5 Avoiding Exponential Complexity

The original EVM had one primary way to avoid exponential parsing complexity: the trace. The trace in the EVM was a set stored in each state containing fiber snapshots of the 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.

- The 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, and multiple resumptions of the same task with a 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 fibers was expensive performance-wise because each fiber had to be checked against the 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.

First, it is 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 the shift/reduce and reduce/reduce conflicts of the (G)LR parsers [18]), as they potentially may lead to multiple parse paths. There are four types of conflicts in the SEVM:

- **Reduce/reduce conflict**. These conflicts occur when two reductions of the 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 the reduce/reduce conflicts. They occur when two reductions of the same length but with a different *reduce_id* occur and result in the resumption of the same task in the same *state_id* twice. The existence of a resume/resume conflict indicates that a rule has an ambiguous but fixed-length prefix with a 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 in the same position. Executing both calls would mean that the same input segment is parsed multiple times with the same grammar rules. The callees may perform further calls that may lead to more conflicts and/or infinite hidden recursion.

- **Match/match conflict**. These conflicts occur when the same task is sus-

pended in the same position twice. Accepting both matches may lead to a 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 an 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 so that 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 the *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 using the following observation. Whenever a new task is called, its 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 bitmask, where the bitmask represents the set of concrete rules that were called. By performing the bitwise-or operator between these masks, it is possible to efficiently recreate a 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, match/match 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 match/match conflicts are quite rare and only happen when a task is resumed twice at the same position, but with reductions of two different lengths.

5.6 Parse-tree Construction

The SEVM constructs SPPFs automatically, as described in Section 4.7.1. Automatic parse-tree construction was chosen over manual AST construction

for the following reasons:

- **Less-noisy grammars**. Including AST construction statements and expressions in the source grammars makes them less readable.
- **Universal node format**. Forcing a specific node storage format makes the parser more predictable because each grammar names and constructs the resulting parse tree in the same fashion. This also enables an easier grammar merger because it is now guaranteed that all grammars will use the same node format, thus eliminating any node type-mismatch conflicts.
- **Higher parsing performance**. The automatically constructed SPPFs are designed to use a minimal amount of memory and are laid out sequentially in a heap (the same cannot be said about traditional ASTs, which may contain additional fields needed for further compilation steps). This ensures that the creation of new SPPF nodes is cheap. Furthermore, all constructed SPPF nodes can be erased from memory in one sweep.

In the typical usage of the SEVM, after parsing, the user takes the constructed SPPF, ‘manually’ removes ambiguous nodes (if there are any) using disambiguation filters, and converts the SPPF to AST (in the host language environment). This may contain additional fields needed for further AST processing, such as fields that contain information needed for type-checking or code generation.

5.7 Conclusions

The following conclusions were reached:

- Direct instruction interpreter substitution to an equivalent JIT compiler may result in unexpectedly low parsing performance because many sizeable instructions are needed to encode the instructions used for terminal symbol matching.
- The DFA extraction enables not only reducing the number of generated machine-code instructions but also extending the parsing-method functionality because the extracted DFAs may be augmented with additional functionality, which otherwise would be difficult to implement in the primary virtual machine.

- The SEVM subset construction further increases the size of the optimised grammar rules because it frequently inlines significant portions of source grammars. This may be avoided using DFA extraction with DFA state caching and deduplication.
- Replacing full execution tracing with more precise task deduplication avoids exponential parsing complexity (5.5) and provides yet another parsing performance boost.
- All these changes extend the SEVM functionality and improve its performance without removing any features present in the EVM or imposing new restrictions. As a result, the SEVM satisfies our functional requirements for an REP parser (generalised parsing, scannerless parsing, dynamically extensible grammars, and local grammar extensions) and, with additional optimisations, *should* exhibit acceptable performance, which was the final requirement for the REP language parser.

6 EVALUATION OF THE SCANNERLESS EARLEY VIRTUAL MACHINE

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

6.1 Language Selection

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

- **ANSI C**. It is one of the most widely used programming languages. As such, any parsing algorithm with the goal of parsing programming languages should be able to parse such a language. It is also commonly used for comparing parser performance.
- **Rust**. Rust is a relatively new programming language but one that is quickly gaining popularity. Its grammar is significantly larger compared to ANSI C and 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 with 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, to parse the ANSI C code deterministically, the parsing method needs to perform a 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 generalised methods become more useful. They are capable of parsing this input with both interpretations and can produce a parse forest, which then can be filtered *after* parsing based on semantic predicates. As such, using generalised

parsing methods to parse the C programming language allows the separation of parsing from semantic analysis and thus improves the separation of concerns.

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

In this sense, the Rust programming language is the complete opposite of ANSI C. Its 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.

6.2 Implementation Selection

The following parser implementations are included in this evaluation:

- **North**: it is the implementation of the SEVM described in this work, which was written in the Rust programming language.
- **Bison** with **Flex**: Bison is a Yacc-compatible LALR(1) parser generator. It is perhaps a de-facto LALR(1) parser generator. It has been 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 a wide variety of programming languages. Because Bison works only with tokens (it is not a scannerless parser), a lexer is needed 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 the Earley parser with various optimisations to make it suitable for use in practice. It is also a nonscannerless implementation and thus is used in conjunction with Flex during evaluation.
- **Dparser**: Dparser is a scannerless implementation of the GLR parsing algorithm. It is one of the very few still maintained projects capable of generalised scannerless parsing. Therefore, Dparser is the closest match in this list to North.
- **Syn**: Syn is a parser for the Rust programming language, implemented

with a hand-written recursive descent parser. It is a nonscannerless parsing method but comes with its own lexer. As such, no external lexer is needed. Syn is primarily used as a library for developing language extensions for the Rust programming language.

6.3 Comparison Method

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 A to learn how to use the tool or how to reproduce the results of this evaluation.

Each scenario comprises the following steps:

1. An input file containing the source code to be parsed is read.
2. The system time is accurately measured, 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 ASTs during parsing.
5. The system time is accurately measured, called end_time.
6. Finally, the $end_time - start_time$ 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 in a CSV file, which can be analysed later. 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 resulting irregularities related to input/output caching (either at the hardware level or at the kernel/file system level), dynamic CPU frequency scaling, and others.

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

6.4 Test Environment

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

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

6.5 Test Data

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

1. The file `input_gcc_470k.i` is an ANSI C source file from the Yaep parser benchmark suite. It contains preprocessed source code of the entire GCC 4.0 compiler. The file is 14.8 MB in size and consists of $\sim$475000 lines of code.
2. The file `input_rust_650k.rs` is a 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 the GitHub Rust repository. Minor modifications were performed on 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; thus, not all source files can be simply concatenated and result in a valid Rust source code). These modifications were primarily performed so that the Syn parser would be capable

Table 25: The median time needed to parse sample inputs

Parser	Language	N	IQR	% Outliers	Median
bison	ANSI C	50	0.0008	20.0	0.4974
dparser	ANSI C	50	0.0104	20.0	16.1007
north	ANSI C	50	0.0162	0.0	4.6132
yaep	ANSI C	50	0.0737	0.0	1.7231
north	Rust	50	0.0197	0.0	6.3258
syn	Rust	50	0.0346	0.0	5.5434

of parsing the resulting file without any additional modifications. 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.

6.6 Test Results

6.6.1 Relative performance comparison

The relative performance comparison results of different parser implementations are shown in Table 25. Out of all tested ANSI C parsing methods, Bison was unsurprisingly the fastest. It is a token-based, fully deterministic parsing method that performs no variable-length lookahead or backtracking. Because it is an LALR(1) parser, a limited form of semantic analysis was performed during parsing to disambiguate identifiers from type names. The ANSI C Bison parser only performs recognition and constructs no parse tree or AST as a result. The Yaep parser is slightly less performant than Bison, but it is significantly more general because it is an Earley parser. It still requires the use of a dedicated lexer; however, no semantic analysis was performed during parsing because the Earley parsers can produce ambiguous parse forests to represent different parse paths. `north` is ≈9.3 times slower than Bison, but it is the first parser in the list that is 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.

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, only one additional Rust parser im-

Table 26: The median time needed to parse `input_gcc_470k.i` with and without garbage collection in `north`

Benchmark	N	IQR	% Outliers	Median
ANSI C	10	0.0150	0.0	5.3165
ANSI C (with GC)	10	0.0035	0.0	4.6139

Table 27: The median time needed to parse `input_rust_650k.rs` with and without garbage collection in `north`

Benchmark	N	IQR	% Outliers	Median
Rust	10	0.0069	0.0	6.9651
Rust (with GC)	10	0.0198	0.0	6.3965

plementation exists: the Syn parser, which is a hand-written predictive recursive descent parser. While it is faster than `north` for parsing Rust, it is only faster by a narrow margin.

The amount of time it takes for `north` to optimise just-in-time and otherwise preprocess 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 the SEVM/`north` implementation apart from other parsing algorithms and implementations.

6.6.2 Performance influence of garbage collector

The primary purpose of the garbage collector in the SEVM/`north` implementation is to reduce memory usage of the parser. As described in Section 5.4.5, it works by periodically scanning all of the currently existing chart entries and removing the ones that are believed to no longer be 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, the SEVM runtime must recreate the required chart entries by reparsing the corresponding input fragments.

Initially, it may seem that the garbage collector should reduce the overall parsing performance for the following reasons:

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

To see the actual performance influence of parsing ANSI C and Rust, additional tests were conducted. The ANSI C (`input_gcc_470k.i`) and Rust (`input_rust_650k.rs`) inputs were parsed both with and without a garbage collector. The results of these tests are displayed in Tables 26 and 27.

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

- Enabling the garbage collector allows the reuse of 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, fewer 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.

6.6.3 Performance influence of incorporated reductions

Partial reduction incorporation (described in Section 5.4.4) is a further optimisation made possible by performing an MIR subset construction. In short, whenever a reduction is performed, the 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 consists of several steps that are computationally expensive:

- Iterating through all suspended tasks requires N_{susp} steps, where N_{susp} is the number of suspended tasks in the origin entry.
- To determine whether a suspended task needs to be resumed, its match specifier must be matched against the *reduce_id* of the reduction. This matching is performed using a hash table.

Table 28: The time needed to parse `input_gcc_470k.i` with and without reduction incorporation in `north`

Benchmark	N	IQR	% Outliers	Median
ANSI C	10	0.0054	0.0	6.9844
ANSI C (with RI)	10	0.0060	0.0	4.6619

Table 29: The time needed to parse `input_rust_650k.rs` with and without reduction incorporation in `north`

Benchmark	N	IQR	% Outliers	Median
Rust	10	0.0101	0.0	9.1468
Rust (with RI)	10	0.0076	0.0	6.4659

- Finally, when it is known that a suspended task can be resumed, a copy of it is made in the target state.

To test the effect of reduction incorporation on parsing performance, additional tests were conducted. The files for 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 28 and 29.

Reduction incorporation on average improves the parsing times in both tests by approximately 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 the origin position. Therefore, each call to a rule with partially incorporated reductions results in one fewer task suspension. This means that fewer tasks are suspended overall, 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 optimised MIR grammars become language (*grammar_id*) dependent and can be no longer reused when dynamically switching to other grammars. As such, in workloads where a parser must parse input that is described by several closely related

```
rule_dyn expr();

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

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

Figure 47: Left-recursive `north` grammar test

```
rule_dyn expr();

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

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

Figure 48: Right-recursive `north` grammar test

grammars, it may be desirable to disable partial reduction incorporation.

6.6.4 Performance influence of recursion type

To test the performance influence of the recursion type, two additional synthetic test inputs were created:

- `input_a_1k.txt` contains 1000 characters of `a`, followed by a semicolon and a new line.
- `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, more instances of recursion are used, such as binary expressions of various programming languages.

The inputs are then parsed with grammars shown in Fig. 47 and Fig. 48 both with and without reduction incorporation. The results for parsing `input_a_1k.txt`

Table 30: 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 31: 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

are shown in Table 30. This test scenario triggers quadratic complexity when performing right recursion; therefore, right recursion is, on average, two orders of magnitude slower than left recursion. This is a well-known characteristic of Earley parsers and is inherited by the SEVM/`north` implementation as well. Optimisations to eliminate the quadratic complexity of right recursion in the Earley parser exist [19]; however, they are not implemented in `north`.

The results for parsing `input_5a_10k.txt` are shown in Table 31. In this scenario, the difference in parsing times between left and right recursion is significantly lower because the recursion depth is limited to five 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 the 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, and Rust) are left recursive.

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

6.7 Validity

6.7.1 Internal validity

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

- All benchmarks are executed in Linux at run-level 3. This means that no desktop applications were running in the background while executing the tests. Thus, any potential unwanted influences of the operating system and the environment are minimised.
- 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 and to ensure that the dynamic CPU frequency scaling policy would not influence the results.
- After running all of the scenarios, outlier detection (IQR method) was conducted to ensure the consistency of the data: even though the tests were performed in a fairly isolated environment, it was still possible for the 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, a large number of outliers would suggest the existence of unwanted external influences.
- Other tests that are not performance-dependent tests are deterministic and only depend on the parser's input and grammar. As such, no external influences can interfere with such tests.

6.7.2 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 practice (especially ANSI C). As such, two primary questions regarding the generalisation of results exist:

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

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

It is still possible that some edge cases remain in the 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 the SEVM or its 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 the probability of such performance issues, 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 chart entry. High average values of suspended tasks and reductions indicate the high overall ambiguity of input grammars, while unexpectedly 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 of magnitude smaller would result in insignificant CPU time. As such, no tests with tests of a minor size were conducted. Therefore, the performance of `north` will generalise to other inputs of ANSI C and Rust.

The other question is whether or not the performance of `north` will generalise to other grammars used in practice? 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 these languages can be parsed 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 exceptions to this rule). The languages that do require semantic analysis for parsing can be parsed in `north` or other generalised parsing methods ambiguously and can be filtered after parsing [4]. The ANSI C language and its input can be considered 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 the depth of the overall grammar. While left recursion is more efficient in the SEVM, primarily because left recursion can be partially incorporated and can avoid much of the complex machinery of new reduction handling, right recursion 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 the 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, an important takeaway of these test results is not the exact absolute

performance values but the relative performance of `north` compared to other parsing implementations because the goal is for the SEVM to be a suitable replacement for such parsers. Even if minor performance fluctuations were to occur, they would not significantly affect the overall result of this study. The SEVM is becoming a viable alternative to more traditional parsing methods, even though it still requires some further research and improvements in certain areas.

6.8 Conclusions

We have created an implementation for the SEVM parser called `north`. Then, we implemented ANSI C and Rust grammars for `north`, which then were used for the performance evaluation. We compared `north`'s performance against several other parser implementations and found that a proper SEVM may be used in practice to parse real-world programming languages because it offers sufficiently high performance and grammar flexibility for such tasks.

On the other hand, further research should clear up the following topics:

- Further SEVM optimisations. The SEVM may be further optimised by eliminating external stacks and driving the execution with native recursion, much like in packrat parsers [11].
- Greedy calls and ordered choice. These additional operations should be added to the SEVM to boost its disambiguation capabilities.
- Error reporting. `north` currently implements no error reporting, but this can be done by analysing the contents of the *susp_list* in the final chart entry.
- Error recovery. The SEVM aborts execution upon encountering the first parse error. It should be modified so that the parsing process may continue (by skipping fragments of invalid input). Error recovery algorithms for the Earley parser exist but are not designed for scannerless parsing [1]. Some work on error recovery in SGLR parsers [31] has been done, but it is uncertain how well such a method may translate to the SEVM.

7 GENERAL CONCLUSIONS

The primary conclusion is that the goal of this research has been achieved. The scannerless Earley virtual machine (SEVM) is suitable for parsing reflectively extensible programming (REP) languages because this parsing method satisfies all REP language parsing requirements (supports dynamically changing grammars with local grammar extensions and enables generalised scannerless parsing). Additionally, this parsing method provides flexibility and sufficient performance for practical application even when used to parse non-extensible programming languages.

Additional conclusions include the following:

- The SEVM can more effectively eliminate lexical parsing ambiguities than the SGLR.
- The DFA extraction improves overall parsing performance. It also can be used to augment other parsing methods (such as Earley or SGLR) to enable more efficient scannerless parsing.
- The garbage collector not only reduces the overall memory usage but also improves parsing performance.
- Left-recursion (and left calls) is more efficient than right-recursion (due to subset construction and partial reduction incorporation).
- The LLVM ORC JIT is not suitable for dynamic grammar compilation into machine code because its code generator is too slow. It produces high-quality low-level code even when optimisations are turned off, which is unnecessary for SEVM.

Possible future research directions include the following:

- Automatic error recovery,
- More performant MIR grammar for machine-code compilation (without using LLVM JIT),
- More recursion and stack memory usage in SEVM to reduce the number of dynamic memory allocations needed during parsing, and
- Layout-sensitive programming language parsing support.

APPENDICES

A bench_parsers Utility

The bench_parsers utility is designed to obtain accurate benchmarks of various parsing methods, north included. The bench_parsers may be run with the cargo utility of the 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: a relative parser performance comparison.
- gc: a benchmark for testing the influence of garbage collection to parsing performance.
- ri: a benchmark for testing the influence of partial reduction incorporation to parsing performance.

B north_cli Utility

The north_cli utility enables the testing of the north implementation of the SEVM. Users can inspect the SEVM parsing process, resulting parse tree, and additional metrics by providing an input grammar file and input data file. 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 be parsed and analysed, after which the grammar MIR will be generated, which will then 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 unoptimised MIR which is directly constructed from the input grammar.
- `-o` shows optimised MIR fragments immediately after they are constructed.
- `-r` shows the reduction trace. This allows tracing 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 the number of reductions per chart entry histogram, number of suspended tasks per chart entry histogram, number of reductions, number of duplicate reductions, allocator information, and others. Some of this information may be meaningful only when parsing when the garbage collector is disabled.

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

REFERENCES

[1] Anderson, S., Backhouse, R. (1981). *Locally Least-Cost Error Recovery in Earley's Algorithm*. ACM Trans. Program. Lang. Syst., 318–347.

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

[3] Brabrand, C., Schwartzbach, M. (2007). *The Metafront System: Safe and Extensible Parsing and Transformation*. Science of Comp. Prog., 2–20.

[4] Brand, M., Scheerder, J., Vinju, J., Visser, E. (2002). *Disambiguation Filters for Scannerless Generalized LR Parsers*. Compiler Construction.

[5] Bravenboer, M., Kalleberg, K., Vermaas, R., Visser, E. (2008). *Stratego/XT 0.17. A Language and Toolset for Program Transformation*. Sci. Comput. Program., 52–70.

[6] Burge, W. (1975). *Recursive Programming Techniques*. Addison-Wesley.

[7] Cazzola, W., Vacchi, E. (2014). *On the incremental growth and shrinkage of LR goto-graphs*. Acta Inf., 419–447.

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

[9] Economopoulos, G., Klint, P., Vinju, J. (2009). *Faster Scannerless GLR Parsing*. Springer Berlin Heidelberg, Berlin, Heidelberg.

[10] Erdweg, S., Rendel, T., Kästner, C., Ostermann, K. (2011). *SugarJ: Library-based Syntactic Language Extensibility*. SIGPLAN Not., 391–406.

[11] Ford, B. (2002). *Packrat parsing: a practical linear-time algorithm with backtracking*. M.S. Thesis, Massachusetts Institute of Technology, Cambridge, United States.

[12] Ford, B. (2004). *Parsing Expression Grammars: A Recognition-based Syntactic Foundation*. SIGPLAN Not., 111–122.

[13] Grimm, R. (2004). *Practical Packrat Parsing*. New York University, Computer Science, Tech. Report TR2004-854.

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

[15] Jim, T., Mandelbaum, Y. (2010). *Efficient Earley Parsing with Regular Right-hand Sides*. Electronic Notes in Theoretical Computer Science, 135–148.

[16] Jim, T., Mandelbaum, Y. (2011). *Delayed Semantic Actions in Yakker*. Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications, LDTA ’11, ACM, New York, NY, USA.

[17] Kats, L., Visser, E., Wachsmuth, G. (2010). *Pure and Declarative Syntax Definition: Paradise Lost and Regained*. SIGPLAN Not., 918–932.

[18] Knuth, D. (1965). *On the translation of languages from left to right*. Information and Control, 607–639.

[19] Leo, J. (1991). *A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead*. Theoretical Computer Science, 165–176.

[20] McLean, P., Horspool, R. (1996). *A Faster Earley Parser*. Proceedings of the 6th International Conference on Compiler Construction, CC ’96, Springer-Verlag, London, UK.

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

[22] Reis, L., Bigonha, R., Iorio, V., Amorim, L. (2014). *The Formalization and Implementation of Adaptable Parsing Expression Grammars*. Sci. Comput. Program., 191–210.

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

[24] Scott, E., Johnstone, A. (2006). *Right Nulled GLR Parsers*. ACM Trans. Program. Lang. Syst., 577–618.

[25] Scott, E. (2008). *SPPF-Style Parsing From Earley Recognisers*. Electronic Notes in Theoretical Computer Science, 53–67.

[26] Seaton, C. (2007). *A Programming Language Where the Syntax and Semantics Are Mutable at Runtime*. M.S. Thesis, University of Bristol, Bristol, United Kingdom.

[27] Standish, T. (1975). *Extensibility in Programming Language Design*. SIGPLAN Not., 18–21.

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

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

[30] Vacchi, E., Cazzola, W. (2015). *Neverlang: A framework for feature-oriented language development*. Comp. Lang., Syst. & Struct., 1–40.

[31] Valkering, R. (2007). *Syntax Error Handling in Scannerless Generalized LR Parsers*. M.S. Thesis, University of Amsterdam, Amsterdam, Netherlands.

Audrius Šaikūnas

EXTENSIBLE PARSING WITH EARLEY VIRTUAL MACHINES

Doctoral dissertation

Tehnological Sciences
Informatics Engineering (T 007)

Editor: Brandy Kelly

Vilniaus universiteto leidykla
Saulėtekio al. 9, LT-10222 Vilnius
El. p. info@leidykla.vu.lt,
www.leidykla.vu.lt
Tiražas: 20 egz.