VILNIUS UNIVERSITY

AUDRIUS ŠAIKŪNAS

EXTENSIBLE PARSING WITH EARLEY VIRTUAL MACHINES

Doctoral dissertation
Technological Sciences, Informatics Engineering (07 T)

Vilnius, 2019

The dissertation work was carried out at Vilnius University from 2015 to 2019.

Scientific Supervisor:
prof. habil. dr. Albertas Čaplinskas (Vilnius University, Technological Sciences, Informatics Engineering – 07 T)

VILNIAUS UNIVERSITETAS

AUDRIUS ŠAIKŪNAS

EARLEY VIRTUALIŲ MAŠINŲ PANAUDOJIMAS PLEČIAMAI PROGRAMAVIMO KALBŲ SINTAKSINEI ANALIZEI

Daktaro disertacija
Technologijos mokslai, informatikos inžinerija (07 T)

Vilnius, 2019

Disertacija rengta 2015-2019 metais Vilniaus universitete.

Mokslinis vadovas:
prof. habil. dr. Albertas Čaplinskas (Vilniaus universitetas, technologijos mokslai, informatikos inžinerija – 07 T)

Abstract

Almost every programming language implementation contains a parser: it is the software component that takes program source code and builds a data structure that can be used further 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 grammars are needed to analyse such languages.

After carrying out 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 on creating a new parsing method what would be suitable for parsing such languages.

Earley Virtual Machines (EVM) is the first iteration of such a parsing method: it is a generalized context-free parsing algorithm that can parse dynamically changing grammars. We carefully describe how such an algorithm was constructed. To ensure that EVM can parse languages with acceptable performance, a successor to EVM was created: Scannerless Earley Virtual Machines (SEVM).

SEVM is an enhanced version of EVM with focus on optimizations and parsing performance. Finally, to ensure that SEVM can be used for practical applications, an implementation of SEVM was developed and compared against various existing parser implementations.

Keywords: adaptive grammars, extensible programming language, extensible parsing, Earley parser, generalized parsing, just-in-time compiler, reflective parsing, scannerless parsing.

Contents

List of Figures

List of Tables

Acronyms

APEG Adaptable parsing expression grammars. 27

API Application programming interface. 16

AST Abstract syntax tree. 12, 14

BNF Backus-Naur form. 13

DFA Deterministic finite automaton. 34

ETS Execution trace set. 44

EVM Earley virtual machines. 8, 42

GLR Generalized LR. 22

GSS Graph structured stack. 22

IR Intermediate representation. 14, 15

JIT Just-in-time (compile/compiler). 128

MIR Medium-level intermediate representation. 85

NFA Nondeterministic finite automaton. 34

PEG Parsing expression grammars. 27

REP Reflectively extensible programming (language). 8, 17

RIGLR Reduction incorporated generalized LR. 23

RNGLR Right-nulled generalized LR. 23

SEVM Scannerless Earley virtual machines. 3, 9, 85

SGLR Scannerless generalized LR. 94

SPPF Shared packed parse forest. 31, 60, 121, 128

1 Research context

1.1 Introduction 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. This is becomes even more apparent when considering the current direction of where computing is headed: bigger and more complicated systems, the internet of things, more distributed and parallel systems unlike anything seen in our history before. Even a 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 which are taken for granted today, like structural programming, procedures or even variables were a novelty at one point or another in computing history. However, updating an existing programming language to support new features is often difficult and time consuming endeavour.

C++, which is one of the most popular and widely used programming languages, has seen 4 standards over 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 creation of each of those 4 standards. Every proposal had to be submitted in a specific format and had to be reviewed by a committee composed out of experts from various technology corporations, like Microsoft, Google and Red Hat. To many people this process may appear even daunting and off-putting, which may push potential participants away from developing a future version of this language. While it may appear that it is certainly possible for a user to modify one of several existing open-source C++

compilers and implement personalized changes, however practically it is just infeasible, as often advanced knowledge of poorly documented compilers internals is required to implement the desired changes. Furthermore it is even more difficult to maintain these changes, as compiler fixes and updates are created hourly for a language as large and complex as C++.

This is one of the reasons why new programming languages are created every year: many developers find it just easier to develop a new programming language rather than to adapt an existing one to suit the particular needs. While this inherently is not an issue, as specialized programming languages are often better suited to solve more particular problems, it does present itself with its own 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 5 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 more and more 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 computer language integration is neither convenient, nor error-proof, as errors in an embedded language code can only be detected during runtime and special symbols used in these ”second class” languages often have to be manually escaped.

But what if one language could be properly embedded into another? What if some desired functionality could *just 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 in researching a class of programming languages called *extensible programming languages*. The core idea behind such languages is that language designers and implementers will never be able to conceive all the possible use cases of their programming language. As such, the extensible language or its implementation should provide 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 languages 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 not present in the base programming language. More powerful extensibility methods may even allow definition of a new programming language within an existing one, thus enabling composition of programming languages that was previously impossible.

Unfortunately, the topic of extensible programming languages is fairly new one and as a result, very few extensible programming languages exist. One of the reason for this is the lack of suitable parsing algorithms for such languages.

As a result, the focus of this work is first part of implementing such a language: parsing. It's a process, which gives structure and meaning to otherwise seemingly random sequence of characters. Because extensible languages can change while they are being used, parsing them requires specialized parsing algorithms.

In this thesis Scannerless Earley Virtual Machines (SEVM for short) are presented. It is a new parsing method that is scannerless, can parse all context-free languages and supports dynamically changing (adaptable) grammars all while maintaining acceptable parsing performance. Furthermore, the grammar definition language for SEVM is designed with extensibility in mind and offers constructs that enable to extend and reuse existing grammars without requiring manual modifications to the original grammars. All these features make SEVM as the perfect candidate for parsing extensible languages.

1.2 Problem statement

We wish the following:

- It would be possible to implement a reflectively extensible programming language.
- There was a parsing method that allowed easy grammar extension and reuse.

However, as discovered during this research:

- No parsing method suitable for parsing a reflectively extensible programming language exists, which prevents implementing REP languages.
- Most available parsing methods do not allow reusing and extending grammars. To make matters even worse, grammars are often specified using 2 separate languages: one for tokens and another for the rest of the grammar.

Because of this:

- Very few extensible programming languages exist. Whenever additional language extensions are needed, users typically resort to: 1) unwieldy boilerplate code that is difficult to maintain; 2) preprocessors, which further complicate the development process; 3) creation of new computer languages that attempt to solve the problem in a more direct fashion.
- Grammars typically have to be reimplemented from scratch each time they are needed and the implementations are tightly coupled with the external environment. This restricts how such grammars could be reused.

We propose:

- Creating a new parsing method (possibly based on existing parsing methods) that is both suitable for using in REP languages and that supports reusing/extending grammars.

1.3 Research goal, objectives and tasks

Goal:

- Creation of new parsing method that would be suitable for parsing extensible programming languages.

Objectives:

- Create new parsing method for parsing extensible programming languages.
- Define a grammar definition language for this parser.
- Evaluate the new parsing method.

Tasks:

- Analyse all available parsing algorithms that are generalized, scannerless or support parsing adaptable/extensible/reflective grammars.
- Identify which algorithms could be used as a basis for implementing a REP language parser.

- Construct the main parsing algorithm.
- Define a grammar language for this parser that allows extending and reusing grammars.
- Implement a prototype for this parsing algorithm.
- Identify the flaws and limitations of this parsing method.
- Construct the improved version of the main parsing method.
- Define the improved grammar language for this method.
- Implement the improved parser in such a way that the implementation would be suitable for performance evaluation.
- Implement several sample grammars with this parsing method.
- Implement several sample grammars with other parsing methods.
- Construct representative parse inputs for parser evaluation.
- Test and evaluate the newly created parsing algorithm in relation to other existing parsing methods.

1.4 Scientific contribution of the research

This research contains several important scientific contributions:

- Extensible programming languages is still fairly new and unexplored scientific topic. The SEVM generalized context-free parsing algorithm that supports adaptive grammars can be used to research, implement and test such languages.
- SEVM is a virtual-machine based parsing method. While virtual-machine based parsing methods have existed before, this is the first instance where a virtual-machine is used to parse as complex grammars as C or Rust. In virtual-machine based parsing approaches, grammars internally are represented by a low-level computer language. These grammars can then be subjected to domain-specific optimizations and transformations that would allow to increase parsing expressiveness or performance (for example, by inlining grammar rule call targets).

- The DFA-extraction method that is used to speed-up SEVM grammars *may* be adapted to other parsing methods (in particular: GLR) to enable performance-wise cheaper token-level disambiguation (however, more research in this topic is needed to be certain).

1.5 Practical significance of results

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

- Good parsing performance (as shown in section 6.7.1).
- Generalized context-free parsing. This enables writing grammars in a more concise way, because the grammar developers no longer need to abide by arbitrary parser limitations (such as no left-recursion in recursive descent parsers that makes defining infix and postfix operators needlessly cumbersome).
- Because it is a scannerless parser, the entirety of input grammar can be defined by using a single language (unlike 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 grammar implementation.
- SEVM grammar language allows defining grammars in a modular fashion: base grammars can be extended by adding additional abstract rule implementations (for example, by defining additional statement types and expression types separately from the main grammar). This would enable easier language extension development, as users no longer need to rewrite the entirety of the grammars they are trying to extend. Such grammar definition approach could be used already in compilers that support procedural macros (for example, Rust programming language allows implementing macros in external modules; these modules could use SEVM to parse their input, which then can be transformed to valid Rust code).
- SEVM parser supports dynamically changing grammars. Using SEVM in a new programming language (or existing one) would at very least allow more flexible macro systems, where the syntax of each macro can be defined by the user. More importantly, SEVM is one of the few available parsing methods than can be used to implement extensible programming languages.

- Using virtual machines to represent grammars offers an additional benefit: it is possible to include general purpose computations within grammar bytecode, which enables to manually drive the parsing process using user-written procedural code, thus further extending the recognized grammar class.

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

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

1.6 Defended claims

1. No existing parsing algorithm matches the criteria needed to implement general reflectively extensible programming (REP) language.
2. Earley parser or its derivatives can be extended to support parsing reflectively extensible programming languages.
3. The Scannerless Earley Virtual Machines parser offers acceptable parsing performance for practical use.

The requirements for a 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 SEVM case, the parsing performance for SEVM would be considered acceptable if SEVM can parse inputs at a similar speed (within one order of magnitude) as other generalized context-free parsing methods.

1.7 Approbation and 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.

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.

1.8 Outline

This thesis is split into 7 main chapters:

- In chapter 1 the research context is given: introduction, motivation, research goal, etc.
- In chapter 2 the basic concepts for understanding this work are presented. Furthermore, the term reflectively extensible programming (REP) language is defined.
- In chapter 3 the current state of art is provided: firstly, the requirements for a REP language parser are formulated. Then various existing parsing methods and related tools are analysed to find the closest one capable of parsing a REP language. After an exhaustive search we conclude that no single parsing method satisfies our requirements, however two main candidates (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, called Earley Virtual Machines (EVM) that is based on Earley/Yakker parsers and that satisfies our requirements for parsing REP languages. Furthermore, EVM grammar language for defining new (extensible) languages, EVM optimizations and other considerations are presented in this chapter as well.

- In chapter 5 we use the observations made by testing EVM prototype and the knowledge learned by constructing the original EVM to create a successor to EVM called Scannerless Earley Virtual Machines (SEVM). One of the primary goals of SEVM is practical parsing performance (while maintaining previous requirements for parsing REP languages). To achieve this performance, additional optimizations and changes are implemented (most notably, DFA extraction, token-level disambiguation and grammar JIT) that enable SEVM to achieve this practical performance.
- In chapter 6 we evaluate an implementation of SEVM parser called `north`: the performance influence of various described and implemented optimizations is measured and the SEVM parser implementation is compared with other parsing methods. Additionally, arguments for internal and external validity of the achieved results are given there as well.
- Finally, in chapter 7 general conclusions of this thesis are presented.

2 Basic concepts

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

2.1 Compilers and programming languages

In early days of computers, computer programs were written in machine languages that directly corresponded to 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 got 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 a program written in a human readable text-form and produces the corresponding binary code that then can be executed by computer's 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 assembly instructions when compiled are directly translated into corresponding binary code. Because of this 1 to 1 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 got more abundant, computer programs became larger and more complex and as such it became much more difficult to write and maintain these kind of programs. Additionally, competing computer architectures, each with their own assembly languages, emerged. So in order to make a program that was written in a 2nd 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. Languages from this generation often mimicked mathematical notation and provided features that didn't exist in underlying computer hardware: variables, subroutines, data structures are all abstract elements that had no direct-to hardware correspondents. Instead, they exist 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 more and more advanced problems, new *computer languages* were created to ease the solution of these problems. For example, computer language CSS is used to describe appearance of web pages. Another language called JSON is used structure, transfer and store arbitrary information that can be then manipulated within other programming languages. Yet another language, Markdown is used to describe formatted text. It is not uncommon for a specialised language (such as JSON, XML, etc) or even several of such languages 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 a domain are called *domain specific languages*.

Most of all 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 of all early third generation programming languages were compiled into machine code. However, later on new languages appeared that were compiled into a kind of binary code called *bytecode*, which was then 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: ability to run the same program across different computer architectures/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 instead. A program that executes or *interprets* such programs is called *an interpreter*. Some interpreters still use compilation to bytecode or machine code to execute the source programs, but this process is hidden away from the user. Interpreted programming languages also tend to provide additional liberties (such as dynamic typing) not found in traditional compiled programming languages. This and 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, as interpreted programs tend to be much slower compared to their compiled alternatives.

2.2 Compiler architecture

Over the years many new programming languages have been created. The actual number is difficult to estimate, but 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 emerged. Usually, a compiler of a programming language is split into 4 major parts:

- *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 *tokenization*. A token is essentially a word of a programming language. Common token types include, but are not limited to: identifiers, numeric constants, string constants, operators. Lexers are also responsible for removing comments and whitespace (all characters that do not represent a visible symbol, such as spaces, tabs and etc) from the source code that is being analysed.

- *Parser* takes the sequence of tokens and produces an *abstract syntax tree* or AST. AST, as the name suggests, 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, declaration and so on.

- *Semantic analyser* is primarily used in preparation for code generation: it adds enough information to the AST that it would be suitable for code generation.

Commonly, during semantic analysis variable references and function calls are resolved, types are checked (in statically typed programming languages). Also, this is where semantic errors are detected (such as attempts to use a variable or a function that doesn't exist).

- *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 a input language. While formal grammars are widely in formal language theory to analyse formal languages and algorithms that deal with formal languages, they are not convenient enough to use in practice to define computer languages. 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 existing parsing (and lexing) algorithms during analysis rely on special data which is 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 generally are distinct: lexer tokens typically are defined using regular expressions, whereas parser productions are defined using a variant of BNF language.

Even though most computer languages are analysed using both with 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 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 why distinction between lexer and parser exists: by offloading some of more trivial syntactic analysis steps to lexer, the overall parsing and compiler performance is increased.

However, in more modern times with advent of more powerful computers, new parsing algorithms emerged that no longer require a separate lexer step and instead tokenization is merged into the parser. Such parsers are called *scannerless*. Removal of the lexer reduces overall compiler complexity and increases the variety of possible input grammars, however at a cost of reduced parsing performance.

Most of 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 using rewriting rules of a 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 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 subsequence 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. But to make use of 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 bytecode or machine code. Early compilers were very specialized 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 IR code not only eases design and porting of compilers, but also makes it easier to perform optimizations of compiled code. Currently the most prominent IR language/code generation framework is LLVM toolchain, which provides a custom IR language and libraries that allow optimization of this code, generation of equivalent machine code, debugging tools and 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 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's 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 ends up in sometimes trivial, but repeating and difficult to modify and maintain code that is sometimes referred as *boilerplate* code. In order to avoid having to manually write this boilerplate code, several solutions are used in practise:

- **Specialized/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, so to ease this task, regular expressions are used instead 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.
- **Macros** allow 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 pattern, it replaces (expands) the found code sequence into appropriate body of a macro. This allows to reduce amount of repeating code in the source files, as commonly used patterns can be defined as macros. There ex-

ist multiple variations of macro systems: for example, C programming language performs macro substitution only on a textual level, while Rust programming language allows to define patterns that operate directly on the AST of the compiled program.

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

- **Metaprogramming** is a method that allows to treat computer programs as their data. There exist multiple types of metaprogramming, but in this work we refer to metaprogramming as an ability for 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 is currently being compiled. Many scripting languages provide a function called `eval()`, which allows dynamic evaluation of language code in 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 into the currently running program. This approach of 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 is directly determined by the API, which differs on a compiler to compiler basis.

Then 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 back when procedural programming was a novelty. One of the first definitions of an extensible programming language was provided by Standish [26], who states that an extensible language simply allows users to define new language features. However such a general definition is not that useful, and as such, several new terms have been created to differentiate between languages with varying degree of extensibility.

Extensible syntax programming languages are languages, which allow their syntax to be extended, often by using a specialised grammar language. Languages which 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 former type of the 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 the compile time by providing syntactic and semantic extensions that are mixed in with the regular code.

3 State of art

3.1 Parsing methods

In this chapter we investigate various existing parsing methods to determine if any can be used (or be extended) to parse reflectively extensible programming languages.

3.1.1 Requirements for a REP language parser

It is fairly obvious a REP language requires a specialized parser. Firstly, a REP language parser has to 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 by using 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 practise, however, there are several challenges to using this algorithm:

- **Poor grammar mutation performance.** Many parsing algorithms rely on data that is derived from the original language grammar. For example, LR parsers use transition tables that are generated from 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. But this means that every time the language's grammar is updated, then the whole parser data has to be regenerated. Even trivial syntactic additions to the initial language would result in having to re-analyse the whole input grammar. Furthermore, several successive grammar modifications even in the same source file would result in equal amount of parser data regenerations. This would make adding syntactic extensions to the base language prohibitively expensive and thus would defeat the purpose of using a REP language.

```
1. Let $G_0$ be the initial grammar and $A_0$ the respective parser
   data (e.g., such as transition tables used in LR parsers).
2. Divide input source into $n$ top level blocks $B_0 - B_{n-1}$ (such as
   top level declarations in C/C++).
3. Parse and semantically analyse $B_i$ with current parser data $A_i$.
   If the current block contains a new syntactic extension, then
   produce a new grammar composition $G_{i+1}$ based on $G_i$ and the
   extension. Update the new parser data $A_{i+1}$ based on grammar
   $G_{i+1}$.
4. Parse the subsequent block $B_{i+1}$ using parser data $A_{i+1}$.
5. Repeat steps 2–4 until completion.
```

Figure 1: Naive extensible parsing algorithm

- **Requirement for clear block boundaries.** In order for the naive parsing algorithm to be able to transition from one grammar to the next, it has to 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 needs to be a clear and unambiguous boundary 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, user may wish to enable a specific grammar extension only for the next statement within the initial program. Such 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 requirements arises from the fact that every top level block could be parsed with an updated grammar and therefore internal structure of the original parser that represents the ambiguity would be lost when transitioning from one grammar to the next.

Secondly, 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, introduction of syntactic extensions with lexical ambiguities would mean that all of the ambiguities would have to be propagated towards the parser, which would have to be specially modified to support such ambiguous tokens. This would result in increased lexer and parser performance.

Thirdly, we require for 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. Generalized parsing algorithms, such as GLR and Earley's used to be just too slow to be used practically. However, with both improvements to computer hardware and further refinement parsing algorithms, we believe that 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 shouldn't be forced to understand inner workings of the used parsing algorithm just so they could write a syntactic extension.

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

1. Support for dynamically changing grammars.

2. Scannerless parsing.

3. Unrestricted context-free grammar support.

4. Support for local (temporary) grammar extensions.

5. Reasonable performance.

3.1.2 LR(k) parsers

LR(k) is a family of table-based, bottom-up parsers. It's one of earlier parsing algorithms that is indirectly still widely used even today. It was first described in [17]. 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. There are 3 possible actions that can be taken:

- **Shift** action, denoted by $S(n)$, indicates that the current symbol a must be pushed on to the stack, a new symbol must be read from the input and that the parser must move to state n.

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

Figure 2: LR(0) parser algorithm

- **Reduce** action, denoted by $R(r)$, indicates that the reduction based on grammar rule r must be performed. If the rule is denoted by $lhs \leftarrow rhs$, then 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 goto table for state s.

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

The tables AT_s and GT_s used in the parsing algorithm can be constructed from DFA built from initial grammar. There exists an algorithm that allows do dynamically grow or shrink these tables, described in [6], 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 making a decision on which action to take. LR(0) parsers perform no lookahead and select the action to be taken immediately based on the current input symbol. This makes LR(0) in most cases practically not applicable as the algorithm cannot distinguish input `x + y` from `x` (when + is right-associative), because in order to determine which action to take (reduce `x` as an expression or attempt to read next symbol) the parser has to lookahead a single 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 practise LR(1) or LALR(1) parsers are used instead. LALR(1) is a modified version of LR(1) that accepts 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, there are plenty of languages that 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)` as there is no simple way to reject identifiers that overlap with reserved keywords. As a result of this, 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 GLR-family parsers

Generalized LR (or GLR) parser is an extension to LR parser that allows parsing most non-nullable context-free languages. It was first described by Masaru Tomita in [28] and intended to be used for natural language parsing, but since then it was also adapted and used to parse computer languages.

GLR parsers share most of the key ideas with LR parsers: they are still table based, which is used to select an action to be performed based on current input symbol, tables also contain shift, reduce and accept actions, 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 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 regular stack to represent alternative parse paths.

Because of this change, the parser no longer requires 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 algorithm from terminating when the initial grammar contains hidden left-recursion. This was dis-

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

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

covered in [24] and a modification of GLR was proposed (called RNGLR) that correctly handles 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 incremental LR table generation approach that was described in [6] is no longer directly applicable to RNGLR parsing. So, in order to support mutable grammars that are required for REP language parsing, the algorithm for incremental LR table generation would need to be modified first to allow dynamic generation of RN parse tables.

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

3.1.4 Recursive decent parser

Recursive descent parser [5] 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 input upon encountering them.

See figure 3 for an example grammar and 4 for the corresponding recursive decent parser implementation. The function `accept(a)` used by the parser consumes the next non-terminal symbol if it matches the provided symbol *a*. The function `expect(b)` consumes the next input symbol and fails if it doesn't match the provided symbol *b*.

Because of the simplistic nature of the implementation, the recursive descent parsers are often implemented manually by hand, without resorting to using a parser generator. The provided parser example is called *predictive* recursive decent parser, because it does not use backtracking and thus it executes in linear time. However, in cases of more complex grammars the use of backtracking may be necessary and would result

```ruby
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 `accept()` and `expect()` functions

in 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 decent parser to implement mutable grammars. This can be achieved by using single parsing function with three arguments: first for the current grammar, the second for the current parsing position and the third to indicate the non-terminal symbol that has to 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 would start by calling the function with the initial grammar, initial grammar's symbol and position 0. Then this function would call itself to parse other non-terminals from the current grammar, while consuming all terminal symbols. Failure to consume an expected terminal symbol would lead to backtracking.

Such 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 operate directly with characters from the initial source code, thus fulfilling 2 out of 4 requirements for a REP language parser. Unfortunately, that's where advantages of the recursive decent 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 doesn't support left recursion: attempting to use left-recursion in grammars would cause the corresponding parse functions to infinitely recurse themselves, 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 always has to 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 decent parser even with mutable grammar support is not applicable for REP language implementation, as it doesn't provide reasonable performance and restricts the allowed class of grammars to 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 out 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: 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, backtracking recursive decent parser tries each production in succession, essentially brute-forcing the possible solution, Earley parser tries to mix those two approaches together.

However, it has been noted that using context-free grammars to define computer languages may not be the most intuitive way to do it. So a new grammar definition formalism was created called parsing expression grammars or PEGs [11]. PEGs eliminate the source of context-free grammar ambiguity by replacing the choice operator `|` with *ordered choice operator* $\backslash$. That means that in the production $A \leftarrow B \backslash C \backslash D$, 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 decent parser works. As a result, backtracking recursive decent parsers can be used to parse all PEGs. However, there is a modification of the recursive descent parser called Packrat parser that allows parsing PEGs in linear time, which is one of the reasons of why PEGs became so popular in recent years.

The Packrat parser [10] is a modification of the recursive decent parser that memoizes 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 why the algorithm runs in linear time.

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

- No left-recursion support. Because Packrat parser is essentially a recursive descent parser, left recursive rules (including indirect or transitive left-recursion) would cause it to recurse infinitely. This restriction requires that some

- High memory usage. Because Packrat parser has to memoize all intermediate parsing results, this causes fairly high memory usage. There do exist some variations of the parser that attempt to optimize memory management of the Packrat parser, such as [12].

- No ambiguity support. PEGs by definition are unambiguous and it's impossible to represent ambiguous languages using them. As a result, parsing a language such as C++ using just PEGs alone 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 union we get different results. If we 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 get 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 get matched. This issue is explored in more detail in [16]. That means, that the extension designer has to be aware of all existing definitions of the target non-terminal and upon extension has to correctly specify the order in which existing and new non-terminal production rules are to be applied. Failure to do so may result in breakage of base language grammar.

Because of these issues, we find PEGs and Packrat parser insufficient for REP language parser implementation.

```
$G_0$:
  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 PEGs that define conditional expressions and their unions

3.1.6 Adaptable Parsing Expression Grammars (APEG)

Adaptable Parsing Expression Grammars (APEG) [21] is an extension to 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, APEG contains additional extensions that enable specification of context-dependent constraints:

- Binding expressions that allow saving context-dependant information during parse.
- Update expressions that allow updating existing attribute bindings.
- Constrain expressions that use other attribute bindings to restrict specific parse paths.

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

While APEG parser presents a viable option for implementing a REP language, APEG model is still based on original PEGs, and thus inherits most of it’s restrictions, namely:

- No left recursion support.
- No support for ambiguous languages.
- 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 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 hasn’t been parsed yet and a set of candidates, which are remainders (tails) of production rules. The parsing process is divided into challenge rounds, during each the most lexically specific candidate is selected and then it is 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 becomes 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 parsing error.

This parsing method also prohibits ambiguities, which are resolved whenever new productions are added. If an ambiguity is found during declaration of a new production that depends on exiting one, an error is generated forcing the user to adjust the newly defined grammar. While this way of handling ambiguities is convenient for defining fully deterministic languages, not all languages (and thus possible language extensions) that are used in practise are context-free and deterministic. As a result, we inability to support ambiguous grammars as 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 introduce special form of lookahead called traps, which are then used to restrict parsing of operator `&`, by ensuring that it is not followed by an additional `&` terminal symbol. This issue becomes even more relevant, when considering the scenario, when lower-precedence operator `&&` is added in an extension. Then, to make sure that this operator parsers correctly, the original rule for binary operator `&` would have to be modified with an appropriate trap.

Because of lack of ambiguous language parsing support, requirement for 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 [7] is a top-down chart parser. The original algorithm can parse all non-nullable context-free grammars. There also exists a modified version of the Earley parser that 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 that are used for parsing. Unlike most other parsing algorithms, these productions aren't preprocessed in any way before parsing.

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 form $(X \rightarrow \alpha \bullet \beta, j)$. Each item contains a production rule, the current parsing position within that rule (represented by $\bullet$) and origin state j. Initially, $S(0)$ contains only the starting production $(S \rightarrow \bullet\alpha, 0)$. After executing the Earley's algorithm (see fig. 6), the chart S has enough information to construct the parse tree for the provided input.

The algorithm is split into 3 logical steps that are repeated in sequence for every input symbol:

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

```
for all input symbols $a$ do
  for all $(X \rightarrow \alpha \bullet Y\beta, j) \in S(k)$ do              ▷ Prediction
    for all $(Y \rightarrow \gamma) \in G$ do
      add $(Y \rightarrow \bullet\gamma, k)$ to S(k)
    end for
  end for
  for all $(X \rightarrow \alpha \bullet a\beta, j) \in S(k)$ do              ▷ Scanning
    add $(X \rightarrow \alpha a \bullet \beta, j)$ to S(k + 1)
  end for
  for all $(X \rightarrow \gamma\bullet, j) \in S(k)$ do                      ▷ Completion
    for all $(Y \rightarrow \alpha \bullet X\beta, i) \in S(j)$ do
      add $(Y \rightarrow \alpha X \bullet \beta, i)$ to S(k)
    end for
  end for
end for
```

Figure 6: Earley parser algorithm

- **Scanning** step finds all items in the current state in 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 terminal symbols are matched with the appropriate production parts.

- **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 parsed language mid-parse. This property of the Earley parser makes it very suitable for implementing a REP language parser. However, this flexibility comes at a cost of low overall parsing performance, because the grammar and the production rules it contains have to be traversed by the algorithm many times during parsing.

The Earley's algorithm makes no assumptions about the nature of the input symbols. These symbols can be both characters of the original language, or lexer's tokens. However, when using Earley's parser as a scannerless parser, the performance considerations of using unprocessed grammar productions within the prediction step become even more important, as defining language's 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.

```
for all input symbols $a$ do
  for all $(X \to \alpha \bullet Y\beta, j, G) \in S(k)$ do                ▷ Prediction
    for all $(Y \to \gamma) \in G$ do
      add $(Y \to \bullet\gamma, k, G)$ to S(k)
    end for
  end for
  for all $(X \to \alpha \bullet a\beta, j, G) \in S(k)$ do                 ▷ Scanning
    add $(X \to \alpha a \bullet \beta, j, G)$ to S(k + 1)
  end for
  for all $(X \to \gamma\bullet, j, G_0) \in S(k)$ do                       ▷ Completion
    for all $(Y \to \alpha \bullet X\beta, i, G) \in S(j)$ do
      add $(Y \to \alpha X \bullet \beta, i, G)$ to S(k)
    end for
  end for
end for
```

Figure 7: Modified Earley's algorithm

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

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

Additionally, because the Earley's 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 GLR parsers, a single AST is not sufficient to express the structure of an ambiguous parse and that more sophisticated data structures (like SPPFs) are required. However, the original paper in which the parser was first described doesn't address this issue in enough detail. This discrepancy was first observed by [22], where a modified version of the algorithm is provided, which produces SPPF for ambiguous parses.

3.1.9 Parsing reflective grammars

The idea that Earley's parsing algorithm can be extended to support mutable grammars was noticed by [27]. The paper's authors present a modified version of Earley's recogniser, which supports parsing *reflective grammars*.

Reflective grammars are a type of grammars, which can modify themselves by adding additional productions mid-parse from recognised symbols within the parsed input. This is achieved by adding the following modifications to the original parsing algorithm:

- State items now have an additional element that represents the current grammar. If the original Earley parser uses items in form $(X \to \alpha \bullet \beta, j)$, then the modified recogniser uses items in 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 within the initial language syntax extensions can be placed. Fig. 8 shows an example of using this non-terminal to introduce a binary addition operator to the initial language.

Some important observations:

- The parsing algorithm handles even cases where the user defined grammar overlaps with extension grammar. In this case ambiguities may arise, but the parsing algorithm would continue to work correctly.
- Even though the modified algorithm based on it's 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 their invocation.
- The modified algorithm performance-wise is almost identical 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 Earley parser just as unusable as the original in any practical environment.

3.1.10 Efficient Earley parsing

The primary reason for not using the original Earley parser in practise it it's 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.

```
(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 appropriate operator precedence

```
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

10 have to be added to the current state. In other words, the whole expression hierarchy has to be expanded every time there is a possibility to encounter an expression in the current input position. The similar situation arises when attempting to parse statements as well. It is not uncommon for a programming language to have more than 20 different operators, productions for each of which would have to be expanded every time an expression may begin.

When using Earley parser for scannerless parser, the performance decrease would be even more grim. Because the parser doesn't have any mechanism to perform lookahead, when parsing an identifier which is part of a larger expression, it has to prepared both to continue parsing the current identifier, and to attempt parse the operator that comes after the identifier ends. As a result on every parsed character of a identifier or numeric constant, it has to reduce the current identifier (or a constant) to an expression and to advance all the previous productions that depend on that expression. In 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 it's performance. The first one [2] makes an observation that Earley sets closely correspond to LR(0) DFAs. By using DFAs computed out of grammar productions instead raw grammar productions to perform recognition, the parser no longer needs to traverse the entire expression/statement hierarchy when encountering such non-terminals. As a result, Earley item now contains (q, j), where q is the state number of the corresponding

DFA node and j is origin state, or, in other words, the position in which parsing of the current non-terminal began.

While such optimization massively boosts 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 for generating these DFAs on-demand would have to be applied if the modified parser were to be used for REP language parsing.

3.1.11 Yakker parser

Another parsing algorithm that attempts to make Earley parsing more efficient is described in [13]. Authors of this paper make an observation that each production rule can be represented by a non-deterministic finite automaton (NFA) like 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 lookup productions of a specific non-terminal in the prediction step (see fig. 12). Because now the parser is represented by a single NFA, it is possible to optimize it by performing specialized minimization, 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 optimization, all items in fig. 10 would be merged into a single state, thus solving the previously described problem of having to traverse entire expression hierarchy each time a possible expression is encountered.

At this point the optimized DFA resembles the LR(0) DFA used by optimized Earley parser described in 3.1.10. The same authors then use this new parsing algorithm as a basis for the Yakker parser [15], which introduces new features not present in original or modified Earley parser, the primary of which is the ability to recognise data-dependent grammars. In order to support such grammars, new grammar definition primitives are introduced:

- Attribute bindings in form $x = e$, where x is a variable and e is an expression.
- Non-terminal symbol invocations with bindings in form $x = A(e)$, where A is a non-terminal symbol. This grammar construct allows to not only parse non-terminal symbols by supplying them arguments, but also to store the result of a parse in a variable that may be later used to form a semantic data-dependant constraint.

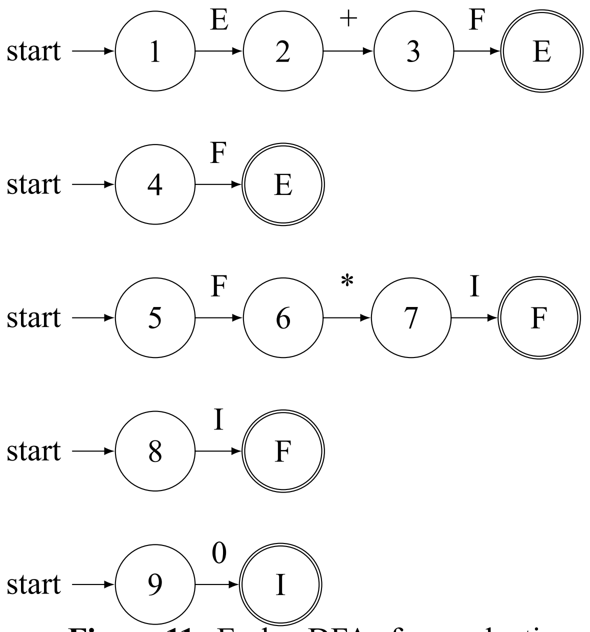

Figure 11: Earley DFAs for production rules from the expression grammar

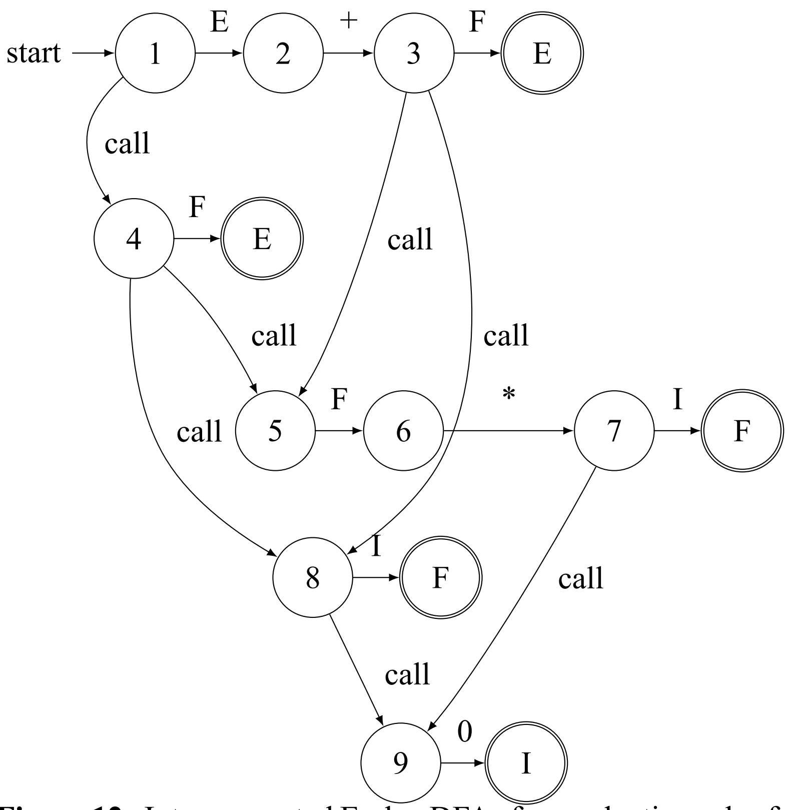

Figure 12: Interconnected Earley DFAs for production rules from the expression grammar

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

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

- Constraints in form $[e]$, which can be considered as ε symbols, which get accepted only if expression e is true.

To support such grammar primitives, corresponding additional Yakker automaton nodes are introduced. Additionally, Earley item is extended to hold environment E with stores all local variable bindings, which results in items in form (q, j, E).

An example Yakker grammar that describes fixed-length digits is provided in fig. 13.

Yakker is the most general and flexible out of all analysed parsing algorithms. It supports regular right hand sides (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-dependant constraints. Using data-dependant constraints allows it to parse even wider class of grammars: for example, the parser may be used to recognise well-formed XML files without using external automatons to match opening and closing tags of this language.

A common task that is performed during parsing is AST construction. In case of LR/GLR parsers, AST nodes for some parsed input are constructed either automatically during reduction execution, or manually, by invoking a user-defined semantic action during reduction process. In case of Earley or Yakker parsers, this can be done in the same way during completion step. However, in case of ambiguous grammars or no lookahead, many intermediate parse results might be constructed and then immediately discarded after entering 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 [14], which can be executed to construct the AST after successfully parsing some of or the whole input, thus eliminating the unnecessary construction of invalid AST nodes.

This parsing algorithm fulfils all the requirements for a REP language parser except one: mutable grammar support. To support mutable grammars, the automatons of Yakker would have to be generated incrementally. Furthermore, new grammar constructs would have to be introduced similar to the ones described in [27] to support parsing reflective grammars. Even despite the required effort to implement such functionality, Yakker parser is a good candidate for implementing REP language parser, be-

```
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

cause the provided feature-set, generality and parsing performance combination cannot be matched by any other parsing algorithm.

3.2 Related tools and languages

3.2.1 Katahdin

Katahdin [25] is one of the very few REP languages that exist. It is dynamically typed language that allows mutation of base language's syntax and semantics. The dynamic nature of the language also allows 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 out of two elements: syntax definition and and evaluation rules, which are both placed within a class that represents the AST node for the newly defined construct. Syntactic extensions are defined using parsing expression grammars, which are later used by a backtracking recursive descent parser. Evaluation rules are defined as series of methods with direct access to the parse tree that allow interpretation of the current AST node. For example, all expressions in the base language and in it's 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 for implementing a simple extension

```
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 unary suffix increment operator ++

is provided in fig. 15.

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

- Katahdin uses PEGs to define syntax for new language constructs. This results in no support for left-recursion, no local ambiguity support, no true language union support. The limitations of PEGs in relation to REP languages are explored in more detail in section 3.1.5
- The choice of recursive decent parser is questionable as well. Recursive decent parser with backtracking, while easy to implement, is notorious for exhibiting poor performance, because backtracking may to exponential time to execute.
- Syntax extensions are global. It is not possible to activate an extension for a selected scope only, like it can be done by using other parsing methods, such as the one described in section 3.1.9.
- Katahdin is fully dynamically typed. Language's author claims that this choice allows the language to support both dynamically and statically typed extensions, because dynamically typed extensions provide more generality. However it 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 on the key motivators for using a REP language - compile-time error checking that ensures that two code fragments from two different languages integrate correctly.

- Poor performance. Because Katahdin is a dynamic programming language, it's libraries are provided in a textual (non-binary) format. That means that each time a language library is imported, it has to be parsed with a fairly inefficient parsing method. After imported libraries are parsed and the constructs within them are evaluated, only then the actual user program may begin execution. With the current Katahdin's implementation it takes several seconds just to parse the standard library. In addition, the performance of user programs even after parsing them is low due to the fact that Katahdin uses AST interpretation to execute it's 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 [9] 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 sugar libraries is a novel concept, a less formal variation of sugar libraries was also implemented previously in Katahdin.

Unlike Katahdin, which uses interpretation to execute it's programs, SugarJ transforms all of it's programs into Java code than then can 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 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 [4] is a language transformation framework that allows grammar definition using context-free grammars. It also provides 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 [8].

```
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 SugarJ extension that implements unary suffix increment operator ++

2. **Desugaring**. This is the transformation step, where all sugar AST nodes within the previously parsed declaration are replaced to appropriate SugarJ nodes based 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, import statements are used load external sugar libraries, while sugar declarations get passed to Stratego.

4. **Adaptation**. During this step new sugar declarations are merged with current desugaring rules. In the same fashion newly defined production rules get composed with 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 creation of extensions that only work in specific scopes. Furthermore, the performance of such parsing method is not ideal (see section 3.1.1 for more details). As such, there is room for improvements regarding syntactic extensibility.

Also, because SugarJ supports only syntax extensions, it is not a true 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 the SugarJ's parsing method.

3.2.3 Neverlang

Neverlang [29] 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 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 incrementally generated LALR(1) parser to parse the target language, which means that it doesn't support scannerless parsers and arbitrary context-free grammars, as such it doesn't meed out 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 in regards to using slices in a REP language compiler.

3.3 Conclusions

In this chapter we defined requirements for a REP language parser and investigated various parsing algorithms in regards to these requirements.

We found no single parsing algorithm that fully satisfies our requirements, but several were almost satisfactory. The closest algorithm to meet our criteria is Yakker, which we believe 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 Earley Virtual Machines

4.1 Earley Virtual Machines

4.1.1 Introduction to Earley Virtual Machines

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

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

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

- **Source grammars** are parser grammars in plain text format. These grammars are written by the user of the parser and describe the parsed language in terms of grammar rules. Additionally, source grammars may contain the abstract syntax tree construction instructions, which allow to control the process of AST construction in fine detail.
- **Compiled grammars** or **grammar modules** are internal representation of source grammars. As the name implies, compiled grammars are compiled from source grammars. Compiled grammars contain sequence of low-level instructions that drive the parsing process.
- The **interpreter** is one of the primary elements of EVM. It *interprets* or executes the instructions contained in one or more grammar modules. As a result, an abstract syntax tree is constructed based on the parse input. The process of

interpreting compiled grammars is synonymous to parsing the input data in the context of EVM.

- **States**. EVM state is an internal structure utilized by the interpreter that tracks execution of the interpreter. These EVM states have a close resemblance to the Earley parser states. There may exist one EVM state per each terminal symbol.
- **Fibers**. EVM fibers have a close relationship to Earley parser items. Each fiber represents a task of grammar rule execution. A fiber may be thought of as a thread of a general purpose programming language in which one grammar rule is executed.
- The **fiber queue** is a queue of fibers that are ready for execution. The interpreter works by removing the first fiber from the queue and keeps executing it until it yields. At which point the next fiber is removed from the queue and execution of it commences. Empty fiber queue indicates a parse error.

4.1.2 EVM grammars

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

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

A *grammar expression* is defined recursively as:

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

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

4.1.3 EVM states

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

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

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

whose starting positions, ending positions and non-terminal symbols match. This avoids the exponential complexity explosion in case of ambiguous grammars/inputs.

4.1.4 EVM fibers

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

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

4.1.5 EVM interpreter

Parsing terminal symbols

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

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

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

Table 1: Terminal symbol sequence parsing example

Grammar rule	Instruction sequence

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

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

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

Parsing non-terminal symbols

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

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

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

Table 2: Non-terminal symbol sequence parsing example

Grammar rule	Instruction sequence

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

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

- `i_stop` instruction discards the current fiber.

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

Resuming suspended fibers

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

1. The suspended thread is duplicated.

2. ip of the copy is set to target instruction pointer, which is retrieved from $symbol\_map$.

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

Table 3: Basic source grammar compilation rules

Grammar element	**Instruction sequence**

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

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

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

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

4.2 Compiling basic EVM grammars

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

4.3 General purpose computation in EVM

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

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

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

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

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 recursive descent and Packrat family of parsers [10]. It allows the usage of regular operators in right hand sides of production rules. This simplifies the definition of new grammars, as repeated and optional grammar elements no longer need to be expressed solely via alternation and recursion.

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

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

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

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

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

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 gets a separate non-terminal symbol, under which

Table 4: Regular operator compilation rules

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

```
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

operators with that precedence level are defined. While such operator with precedence definition method is simple and easy to understand, it quickly becomes cumbersome when dealing with real-world programming languages, such as C++, Ruby and similar, which often have over 15 different levels of operator precedences.

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

Because definition of operators (either unary, or binary) is such a fundamental task when defining new grammars for programming languages, newer parser generators and language translation frameworks often allow specifying the precedences of operators directly either by assigning each operator a numeric precedence value, or using operator definition order to infer the precedence of each operator [8]. As such, it would be beneficial for EVM to support specification of operator precedence levels natively, especially because one of the goals of EVM is to support adaptable grammars that can be extended dynamically during runtime.

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

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

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

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

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

4.4.3 Specifying operator associativity

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

```
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 5: Updated non-terminal symbol compilation rules

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

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

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

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

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

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

Table 6: Fixed length lookahead example

Grammar rule	Instruction sequence

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

```
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

4.5 Parsing with regular lookahead

4.5.1 Fixed length lookahead

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

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

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

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

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

An example usage positive lookahead operator is provided in table 6. Figure 19 shows an example where positive lookahead can be used in a real-world scenario when defining identifiers.

Table 7: Fixed length lookahead compilation rules

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

Table 8: Variable length lookahead compilation rules

Grammar element	**Instruction sequence**

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

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

4.5.2 Variable length lookahead

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

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

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

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

4.6 Parsing with data dependant constraints

4.6.1 EVM grammar language

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

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

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

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

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

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

- `i_call_dyn` instruction needs to be extended to include the argument number to copy to the callee. The copied arguments are discarded from the caller's stack frame after the call is complete.

- `i_reduce_r` (*reduce and return*) instruction needs to be created to allow returning values from the callee. It behaves exactly the same as `i_reduce`, but also pops a value from the current fiber's frame and stores it in *reductions* entry of state S_{origin}. This value can be accessed later by `i_match_dyn_r` instruction.

- `i_match_dyn_r` instruction behaves exactly the same as `i_match_dyn`, but also pushes the return value of the callee to the current fiber's stack.

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

4.6.2 Matching input against dynamic content

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

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

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

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

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

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

Table 9: Extended grammar language elements and their compilation rules

Element name	Syntax	Instruction sequence
Grammar rule	**rule** $sym(arg_1,...,arg_n)$ $stmt_1$ ... $stmt_n$ **end**	$code(stmt_1)$ ... $code(stmt_n)$ `i_reduce` sym, 0 `i_stop`
Block statement	$stmt_1$ ... $stmt_n$	$code(stmt_1)$ ... $code(stmt_n)$

If statement	**if** $cond$
	$body$
	end
		$code(cond)$
		`i_bz` l_{end}
		$code(body)$
		l_{end}:

Parse statement	**parse** $grammar\_expr$	$code(grammar\_expr)$
Return statement	**return** $expr$	$code(expr)$ `i_reduce_r` sym, 0 `i_stop`

While statement	**while** $cond$
	$body$
	end
		l_{start}: $code(cond)$
		`i_bz` l_{end}
		$code(body)$
		`i_br` l_{start}
		l_{end}:

Variable declaration statement	**var** $v = expr$	$code(expr)$
Integer addition expression	$e_1 + e_2$	$code(e_1)$ $code(e_2)$ `i_int_add`
Integer constant expression	$value$	`i_push_int` $value$
Variable read expression	v	`i_peek` $stack\_slot_v$
Variable write expression:	$v = e$	$code(e)$ `i_poke` $stack\_slot_v$
Parameterized non-terminal grammar expression	$A(arg_1,arg_2,...,arg_n)$	$code(arg_1)$ $code(arg_2)$ ... $code(arg_n)$ `i_call_dyn` A, $prec_{min}$, n `i_match_dyn` A, $prec_{min}$

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

Table 10: Parsing fixed length fields

Grammar rule	Instruction sequence

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

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

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

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

4.7 Abstract syntax tree construction

4.7.1 Automatic AST construction

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

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

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

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

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

It is important to note, that EVM is capable of parsing ambiguous grammars, in which case the AST size may grow exponentially. To avoid this, shared packed parse forests (or SPPFs for short) can be used [22]. In SPPFs subtrees that refer to alternative parse paths are packed into a single ambiguous node.

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

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

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

- If a reduction is *unique* (i.e. there are no other reductions that share the same source interval and the same non-terminal symbol), then a normal child node is constructed. Then it is assigned a unique index and this index is stored within *reductions* entry in appropriate state.
- If a reduction is *non-unique* (or ambiguous), then a normal child node is created and it is assigned a unique index. However, this time the existing node is converted to ambiguous packed node, and the newly created node is added as it's child.

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

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

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

- Inclusion of undesirable AST child nodes. EVM is primarily a scannerless parser, and as such will be used to parse whitespace. It is not uncommon to define a non-terminal symbol for recognising whitespace and then using that within other grammar rules. As such, during automatic AST construction, nodes that represent whitespace will be added to the resulting AST, possibly unnecessarily increasing the overall size of AST and littering it with nodes that carry no semantic information.
- Rigid and inflexible AST node type. Every normal node of the AST currently shares the same type and thus the same structure. Such behaviour however may not be desirable, as different non-terminal symbols represent different language elements with unique behaviours. Furthermore, it is common to use the AST to store semantic information when performing semantic analysis of the AST during later stages of compilation. Current node model has no space reserved for such semantic information and changing the node type would require changing the

internals of EVM itself. The most flexible way to use the parsed result would be to convert the EVM AST to possibly polymorphic user-defined AST type that includes all the necessary fields and behaviours to perform semantic analysis.

4.7.2 Manual AST construction

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

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

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

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

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

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

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

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

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

Grammar rule	**Instruction sequence**

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

node construction could be avoided at a cost of having to manually specify (both within the grammars and possibly within the host environment) of how to construct the AST.

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

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

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

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

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

4.7.3 Delayed semantic actions

The arguments for delayed semantic actions

Delayed semantic actions [14] is an attempt to avoid too eager computation within non-terminal rules that may not contribute to the parsing result in the Yakker parser [15]. In this section we present an adaptation of delayed semantic actions for EVM.

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

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

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

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

Constructing execution history labels

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

Several different label types are required:

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

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

Furthermore, additional instructions and existing instruction changes are required:

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

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

Compilation of grammars that use delayed semantic actions

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

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

Table 13: Rules for compiling grammars with delayed semantic actions

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

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

Replaying labels

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

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

The label playback process consists of several steps:

1. **Collection**. During the collection step, labels for a single non-terminal symbol are collected into an array (essentially flattening a linked list of labels into array). The first label in the resulting array is always the normal (non-ambiguous) reduction label that contains the unique reduction tag. The rest of the labels are added to

the array in the order they were constructed. Call labels are added to the resulting array without traversing the callee labels.

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

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

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

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

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

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

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

The replay function for the rule, implemented in Ruby programming language is shown in fig. 22. The method `each_action` iterates over the collected labels (starting from the 2nd label). `prev_result` accesses the resolved value of the previous resolved

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

Grammar rule	Instruction sequence

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

```
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 Ruby programming language

label. `create_add_node` is a user defined method that constructs the binary addition AST node. It is important to note, that the replay function can be implemented in any language and it is not in any way bound just to Ruby programming language. For example, the same replay function can be implemented in C programming language, as shown in fig. 23.

4.8 Parsing reflective grammars

One of the key reasons for choosing Earley parser as basis for constructing the parsing method for a REP language is its flexibility and limited need for grammar preprocessing. In this chapter we describe how EVM can be extended to support adaptable grammars. The approach for implementing adaptable grammar support in EVM is inspired by [27].

```
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 C programming language

4.8.1 Dynamic grammar composition

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

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

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

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

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

4.8.2 Extensions of EVM grammar language

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

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

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

4.8.3 Compiling EVM grammars with domains

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

Table 15: Additional grammar language elements to support reflective grammars

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

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

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

```
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

block, the only those domains should be disabled, which have been previously enabled within the same block.

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

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

Table 16: Rule for compiling domain activation statement

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

- `i_dom_restore` n restores the active domain set by retrieving it from stack slot n of the current fiber.

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

4.8.4 Loading multiple grammar modules in EVM

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

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

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

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

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

4.8.5 Parsing reflective grammars in EVM

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

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

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

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

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

4.9 EVM performance improvements

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

4.9.1 Garbage collection of suspended fibers

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

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

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

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

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

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

4.10 Eliminating dynamic non-terminal call indirection

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

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

If the 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. As such, it becomes no longer necessary to perform dynamic rule lookup and filtering during parse time. Therefore dynamic instructions `i_call_dyn` and `i_match_dyn` can be replaced into corresponding static ones: `i_call` and `i_match_sym`.

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

4.10.1 On-demand instruction subset construction

Importance of subset construction

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

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

- 12 arithmetic operators.
- 6 comparison operators.
- 3 logical operators.
- 6 bitwise operators.
- 10 compound operators.
- 7 member and pointer operators.

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

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

The way EVM currently operates can be similar to a non-deterministic finite automaton: just like a NFA can be in multiple states at the same time, so does EVM can execute multiple fibers at the same time. But it is well known that any NFA can be converted into DFA by applying the process known as subset construction. The Faster Earley Parser [19] or Efficient Earley Parser with Regular Right-hand Sides [13] are both based on this algorithm. By applying such parsing algorithms to parse C++, it would no longer take 50 distinct fibers (or items in Earley parser case) to parse a single expression: all 50 grammar rules could me merged into 1 optimized rule.

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

Instruction ε-closures

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

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

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

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

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

There are two possible actions:

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

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

Merging instruction ε-closures

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

Because of the second goal, dynamic instructions like `i_call_dyn` and `i_match_dyn` are replaced with their static counterparts. In general, all instructions are merged based on *instruction merger key*. If two instructions share the same instruction merger key

Table 17: Rules for computing instruction closures

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

Table 18: Rules for computing instruction merger keys

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

then they can be merged into a single instruction. The instruction merger keys can be derived from rules provided in table 18.

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

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

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

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

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

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

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

1. Unordered instructions: `i_call` and `i_reduce`.

2. $n-1$ `i_fork` instructions for the following n ordered instructions.

3. n ordered instructions.

Table 19: Subset construction example

Source grammar	**Compiled grammar**	**Optimized grammar**

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

4. `i_stop` instruction if $n = 0$.

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

4.11 Conclusions

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

- Can parse arbitrary context-free grammars.

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

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

5 Implementation of Scannerless EVM

5.1 Scannerless EVM

5.1.1 Flaws of the original EVM

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

- **Recognized grammar class**. Different parsing methods can recognize different classes of input languages. For example, LR(0) parsers can only recognize LR(0) grammars. More generalized methods, such as GLR [28] can recognize wider class of input languages (all context-free languages in the case of GLR). However, even then it is possible that such parsing method may not be able to recognize all programming languages used in practise, as not all programming languages are context-free languages. The size of recognized grammar class determines how many real-world computer languages can be recognized by this parser.
- **Expressiveness of grammar language**. Parser development typically starts with creation of target language grammar. This grammar is written in a specific grammar description language, which is then read by parser or parser generator, which then is responsible for generating and/or configuring the parser so it then can recognize the target language. These grammar description languages often provide additional features beyond just production rules to express the target grammar in a more clear and concise fashion. For example, the `bison` parser generator supports operator precedence declarations, which provide a more clear and compact way to describe operator precedence. The existence of such operator precedence declarations does not allow to parse additional languages, but merely allows to express the already recognizable languages in a more intentional fashion. As a result, the greater expressiveness of the grammar language makes development

of new grammars easier.

- **Performance**. The performance of the parsing method and it's implementation is one of the primary factors determining whether or not such parser is suitable for parsing real-world computer languages. Generalized parsing methods have existed for decades, however even today they are not widely used due to their lacklustre performance. The same is even more true for scannerless parsing methods, as not a single scannerless parser is used to parse any high-profile programming language (both `gcc`, `clang` and Lua use hand-written recursive descent parsers [5], MRI Ruby implementation uses LALR(1) `bison`, CPython uses a custom bottom-up tokenizer and parser combination, etc).

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

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

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

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

- While EVM was created with scannerless parsing in mind, there is still one key issue that will severely limit the performance of EVM, even if EVM was implemented in a non-interpreted language: during parsing EVM creates a state for each terminal input symbol. This means that to parse input of length n,

$n * size\_of(State)$ bytes of memory is needed just to represent parser states. These states will then contain additional dynamically allocating structures, such as the list of suspended tasks, reductions and the trace.

- Because EVM is a scannerless parser, there needs to be a way to disambiguate identifiers from keywords in languages that have such grammar elements. Furthermore, there needs to be a way to disambiguate operators that consist of more than one character (for example, logical operator `&&` in C may be incorrectly interpreted as a pair of `&` operators). While this disambiguation can be performed post-parse by eliminating invalid parse paths in resulting parse forest [31], both ambiguous parsing and invalid parse elimination would incur additional performance costs. As such, simple character-level ambiguities should be resolved as early as possible during parsing to avoid ”useless” work that yields invalid parse trees.

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

5.1.2 Overview of the internal structure of SEVM

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

SEVM consists of the following primary components:

- **Grammar compiler** translates textual representation of input grammar to medium-level intermediate language (or MIR for short). It also detects any syntax or semantic errors of the input grammar.

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

- **Resolver** is responsible for invoking optimizer and translating the resulting MIR into machine-code.
- **Runtime** is responsible for coordinating the execution of parser.

SEVM consists of the following data structures:

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

with reductions. Each suspended task contains: *task*, *resumes*, *pos_match*, *neg_match*. A *task* is a copy of suspended task data. *resumes* records the occurrences each time this specific tasks is resumed. *pos_match* represents positive match conditions for resuming this task. *neg_match* represents negative match conditions for resuming this task. Both *pos_match* and *neg_match* are referred as match specifiers.

- **Resume** stores information about a single occurrence of task resumption: the index of reduction that woke the task (*reduce_id*), the length of that reduction, and parse-tree node index that was appended to the newly awakened task. This information is used to eliminate some duplicate parse paths that may lead to exponential complexity. See chapter 5.5 for more about eliminating exponential complexity.

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

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

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

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

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

5.2 Improving grammar expressiveness

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

5.2.1 Abstract grammar rules

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

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

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

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

```
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

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

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

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

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

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

5.2.2 Named precedence groups

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

The expressions that have the highest precedence in ANSI C language are *primary expressions*. Below them are *postfix expressions* with slightly reduced precedence. Even lower precedence have *unary expressions* and then *cast expressions*, etc. In expression hierarchies with precedence, rules that represent expressions with lower

```
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

precedence only refer to expressions with higher precedence. This, however, is not always true: in ANSI C case, `unary_expression` refers to `cast_expression` which has lower precedence. Similar situation can be observed in grouping expression of `primary_expression`, which refers to `expression`, which is the top of the expression hierarchy.

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

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

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

```
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

`cast_expression`).

There are several types of calls in `north`:

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

```
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)

- Named precedence group calls. These are in form `cast_expression`, where `cast_expression` refers to a named precedence group. In this case, the abstract rule is invoked that is provided in the definition on the referenced named group with the appropriate precedence level.

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

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

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

5.2.3 Dominating terminals

Multi-line comments in ANSI C programming language start with characters `/*` and terminate with `*/`. In `north` such comments may be parsed with a rule shown in fig. 28. However such simple rule is not entirely correct: the comment terminator `*/` will be ambiguously matched both as comment terminator and as comment body, forking the rest of the input into two distinct path: one where comment never terminated, and another where `*/` was interpreted as comment terminator.

To avoid this ambiguity, the rule may be redefined as shown in fig. 29. In this case the character sequence `*/` is excluded from comment body by firstly allowing the comment body only to contain non-`*` characters (`r"^*"`), and then requiring that character

```
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: Unoptimized MIR for the ANSI-C multi-line comment rule

* must not be followed by a slash (`("*", r"^/")`). Such rule correctly and unambiguously parses C comments, however it's nowhere as clear as the initial rule shown in fig. 28.

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

Such precedence or priority in SEVM can be specified using *dominating terminal symbols*. In `north` grammars user may annotate grammar expressions with `dom_g` specifier, which would cause the last characters or all descendant string grammar subexpressions to parsed with higher precedence. That way the grammar rule for parsing C comments may be rewritten to the one shown in fig. 30.

The two primary reasons for implementing dominating terminals is that they can be simplify the definition of various grammar rules without any reduction of parsing performance: such behaviour may be implemented statically in SEVM optimizer. The rule shown in fig. 30 may be translated to unoptimized MIR graph shown in fig. 31.

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

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

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

Such implementation of dominating terminals is not only simple and effective, but also enables using dominating symbols to disambiguate tokens at character-level as described in chapter 5.3.6.

5.3 Ambiguity elimination

Even though original EVM could parse real-world programming languages, it could not do so without ambiguities. As such, before the resulting parse-tree could be used, it needed to be filtered by disambiguation filters [31], which would remove the parse nodes that represent invalid parse paths. This approach causes two main issues:

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

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

5.3.1 Negative reductions

Negative reductions are an adaptation of SGLR's reject reductions [8] for SEVM. In Scannerless GLR family of parsers, reject reductions/productions are used to disambiguate reserved keywords from identifiers.

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

Table 20: 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

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

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

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

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

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

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

In order to be able to compare reduction kinds, each reduction kind is assigned a unique integer value (see table 20). Then the reduction kinds are compared by these integer values.

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

```
#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: Unoptimized MIR for grammar rule that defines an identifier

The grammar shown in 32 defines a rule for parsing identifiers, which may be composed from lower case or upper case characters followed by a space. However, if the identifier matches the keyword `if`, then a negative reduction is produced, which prevents any other normal identifier reductions from being added. This effectively disambiguates identifiers from keyword `if` (see fig. 33 for 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.

5.3.2 Strict execution ordering in SEVM runtime

EVM, much like the original Earley parser [7], performs mostly breadth-first search (with the exception when fiber priorities were involved, which were used primarily to implement regular look-ahead). Other than this, the rest of the execution of the parser was unordered: `i_fork` instruction for creating duplicate fibers would queue the fiber for execution, but in arbitrary order. `i_call` family of instructions also behaves similarly: the newly created tasks are also queued in an unspecified order.

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

- `CtlFork` $B_1, B_2, ..., B_N$ instruction has to ensure, that the basic blocks $B_1, B_2, ..., B_N$ complete in the same order as they are given to the `CtlFork` instruction.

- `StmtCall` family of instructions has to ensure that the callee will begin execution immediately after the current task completes or is suspended with `CtlMatchSym`.

Internally, this is implemented by a two layer stack:

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

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

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

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

1. Locate the currently active chart entry by retrieving it's chart index from the top of the primary call stack. If the primary stack is empty, then parser terminates.
2. If the secondary stack is empty, then attempt to populate it by *failing* the current entry (see chapter 5.3.3). If failing yields no new tasks, then remove the top element from primary stack index and go to step 1.
3. Pop a task from the secondary stack stored in the current chart entry.
4. Resume the task.
5. Go to step 1.

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

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

- Members grammar expressions $E_1, E_2, ..., E_n$ of the alternative grammar expression $E_1|E_2|...|E_n$ now complete in the order in which they are given.

- Call grammar expression *C*, where *C* is a valid call target, now fully completes (all of the possible alternative parse paths are analysed), before resuming the caller. This happens because each call grammar expression is translated into a pair of `StmtCall` and `CtlMatchSym` instructions. The first one queues the callee for immediate execution and the 2nd one causes the current task (caller) to be suspended, effectively yielding execution control to the callee. If the callee creates new subtasks (for example, as a result of `CtlFork` or `StmtCall`), they are placed on the top of the current secondary stack (if the call is left recursive) or on top of another secondary call stack, which causes causes the currently active chart entry to shift.

- Reject statements `reject` *E*, where *E* is another grammar expression now complete before any subsequent statement completes. This is because reject statements fork execution with `CtlFork` into two parse paths: the primary parse path, which contains the code for grammar expression *E* and terminates with `REJECT` reduction, and the secondary parse path, which contains the remainder of the current parse rule. Because of this, it is guaranteed that the `REJECT` reductions will always happen before `NORMAL` reductions, thus fulfilling the strict execution ordering requirement for negative reduction implementation.

5.3.3 Negative matches

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

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

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

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

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

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

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

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

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

5.3.4 Greedy non-terminal repetition

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

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

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

5.3.5 Strict execution ordering in SEVM optimizer

So far we described how the runtime `north` preserves the strict execution order that is required to implement negative reductions and negative matches. However, ensuring proper execution ordering just in runtime is not enough: SEVM relies heavily on it's optimizer, which can merge multiple grammar rules by performing a variation of subset construction on MIR graphs (the algorithm for which is inspired by Efficient Earley Parser [13]). In this chapter the description is given how the strict execution ordering is preserved during optimizer's subset construction.

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

1. Add the instruction pointers to be merged into a initial set S_I.
2. Add this set into resolution queue Q.
3. Remove one set S_0 from the Q.
4. Find ε-closure of the set S_0 and store it as set S_1.

Table 21: Rules for computing SEVM ε-closures

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

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

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

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

8. Continue until Q is empty.

Much like the original subset construction for converting NFAs to DFAs [20], the one used for EVM uses sets to represent instruction ε-closures and a queue to control the order of individual subset construction steps.

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

Rules for constructing ε-closures in SEVM are given in table 21. Whenever one of the given instructions is encountered, the appropriate actions are executed:

- `VISIT` E recursively visits the entity E:
  - If E is an instruction, then it's visited according to the rules provided in table 21.
  - If E is a basic block, then the first instruction of that basic block is visited.
  - If E is a concrete rule, then the first basic block of that rule is visited.
  - If E is an abstract rule, then all of it's implementations are visited.
- `IGNORE` ε-closure construction.
- `RELEVANT` I adds instruction I to the resulting ε-closure.

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

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

The main disadvantage is that this results in a significantly higher amount of code generated: in original EVM generated rule suffixes were reused possibly several times across entire grammar. In SEVM, the reuse may only happen internally within one generated rule. In other words, if optimizer constructs merged rule for parsing `A | B` and later for `A | C`, then no code between these two generated rules will be shared, whereas EVM may reuse some part of `A | B`, which represents a unique suffix of `A` in `A | C`. To combat this duplication of code, matching state transition tables in deterministic DFAs are cached and de-duplicated, as described in chapter 5.4.3.

5.3.6 Token level ambiguity elimination

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

```
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

Consider the grammar shown in fig. 34. It defines two grammar rules: one for parsing identifiers and another for parsing a reserved keyword `self`, both of which must be followed by a space. If these non-terminals are used in a grammar expression like `ident | kw_self`, then the result would be ambiguous, because both rules would match. To resolve this ambiguity, negative reductions can be used.

Alternatively, the grammar may be modified as shown in fig. 35. In this case, the terminating whitespace symbol (in practise an alphanumerical boundary symbol is typically used instead) is changed to be dominating with `dom_g` specifier. As a result, when `ident | kw_self` expression is encountered, `ident` and `kw_self` rules are merged. Subset construction continues until the terminating symbol is encountered, at which point the lower priority (non-dominating) terminating symbol for `ident` is filtered-out, allowing only `kw_self` reduction to occur, thus eliminating the ambiguity.

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

For this reason, `#[token_group]` attribute is introduced to `north`, which can be used to annotate named precedence groups. When a call is made to a rule that is part of a

```
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 22: 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

`#[token_group]` group, then the call to that rule is replaced with a call to the whole group, without changing the way `CtlMatchSym` instruction is generated. This means that the members of a token group are always guaranteed to be merged during subset construction. As a result, combining token groups with dominating terminals allows to effectively disambiguate keywords from identifiers.

Comparing this approach to negative reductions reveals significant performance gains for parsing reserved keywords. Table 22 shows performance cost (in terms of total reductions needed) to recognize disambiguated keywords and identifiers with both of the described approaches.

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

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

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

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

5.4 Parser optimizations

5.4.1 Profiling EVM

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

- Built-in performance counters. During various steps of `north` execution, execution times for most important components are measured and stored. This information is then optionally displayed after the execution to the user.

- `callgrind` code profiler. This is a tool designed to profile program performance. It works by instrumenting input programs and keeping detailed logs of their execution. As a result, the input program is executed significantly slower, but the additional instrumentation allows to obtain detailed metrics about the entire process of program execution.

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

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

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

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

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

5.4.2 Just-in-time grammar compilation

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

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

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

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

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

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

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

`CtlMatchChar` instructions are compiled into several LLVM instructions:

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

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

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

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

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

5.4.3 DFA extraction

Terminal symbol matching shortcomings

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

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

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

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

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

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

```
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 3 keywords

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

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

Simple DFA extraction

Consider the grammar shown in fig. 37. It defines 3 keywords: `self`, `static` and `struct`. During subset construction shared prefixes of these keywords will be merged and MIR shown in fig. 38 will be produced. This MIR may also be visualised as a deterministic finite automaton as shown in fig. 39, which captures the essence of DFA extraction method: the segments of deterministic source MIR get extracted into a separate DFA, which is then used for matching terminal symbols. Then, instead of `CtlMatchClass` (and `CtlMatchChar`) instructions, the resulting MIR contains a new `CtlExecDFA` instruction, which executes the DFA and transfers the control based on success or failure of DFA match result.

Optimized MIR for abstract rule `kw` with DFA extraction enabled is shown in fig. 40. `CtlExecDFA` instruction takes 2 operands: the DFA to be executed and transition table

```
 #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: Optimized MIR for matching 3 keywords

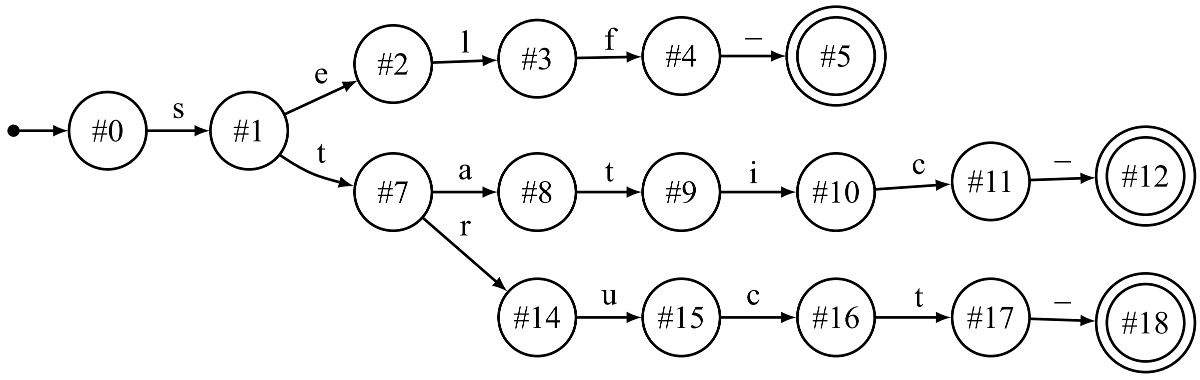

Figure 39: Traditional DFA for matching 3 keywords

```
#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: Optimized MIR for matching 3 keywords (with DFA extraction enabled)

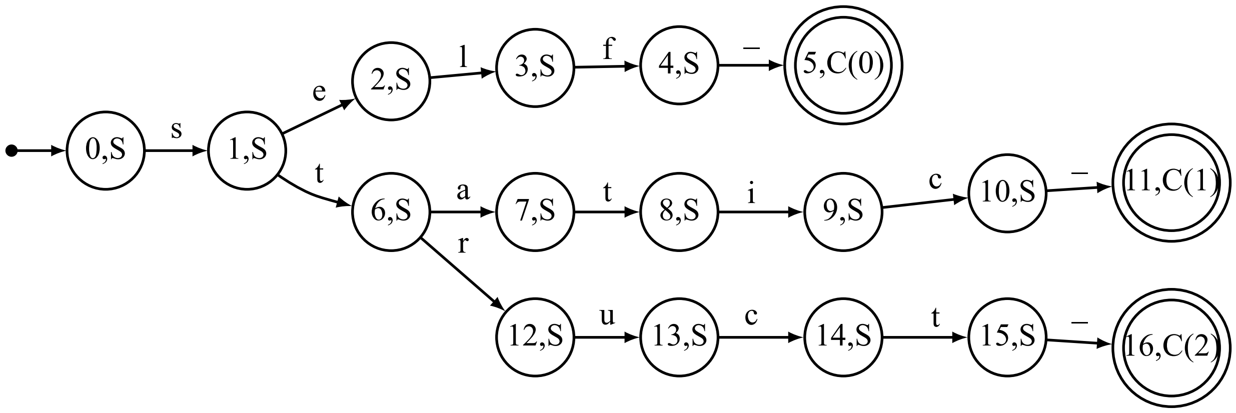

Figure 41: SEVM DFA for matching 3 keywords

that pairs the result of DFA with the target *state_id* of the task. Note the significant reduction of basic blocks in this version of optimized MIR.

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

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

- *Shift* ("S") states only consume a single input symbol and move to a different state. Each shift state contains a transition table.
- *Fail* ("F") states fail the DFA simulation immediately upon entering them. Typically each DFA contains exactly 1 fail state, which is reachable from all other states with unexpected terminals. They are not shown in any of DFA visualisations, because there would be an edge from each *shift* state to the fail state with all other characters from ASCII range 0..255.
- *Complete* ("C") states terminate the DFA simulation with given result. The result is a number that then is used in MIR to transfer control of execution.
- *Lookahead* ("L") states are used to implement lookahead. See chapter 5.4.3 for more.

Furthermore, shift state transition tables are split into two parts: transition index table and transition state table: transition index table store indices of transition state table, which store actual destination state indices. This two layer transition-table approach allows to de-duplicate and reuse transition index tables. All transition index tables have

256 entries (1 byte each), where 1 entry is reserved for each possible input character. Transition state table is variable sized and it's size corresponds to the number of unique transition destinations from a specific state.

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

Dominating terminals in extracted DFAs

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

Greedy tokens

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

They way the extracted DFA works already resembles traditional lexers. By extending this analogy can farther we can implement *greedy tokens*: in case where there are multiple token matches available (such as `&` and `&&`), select the longest.

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

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

```
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

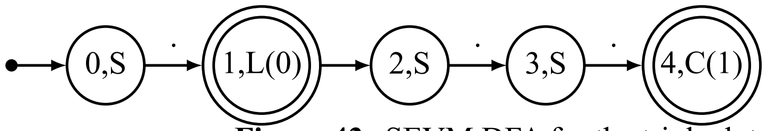

Figure 43: SEVM DFA for the triple dot grammar

to paths: one task with another DFA that performs character matching, and another that contains other instructions.

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

An example grammar for parsing and disambiguating operators `.` and `...` is shown in fig. 42. During subset construction, terminal symbol matching will be extracted into 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.

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

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

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

5.4.4 Partially incorporated reductions

Reduction incorporated parsers

The LR family of parsers [17] use a stack to track the execution of overall parsing process. The stack contains state indices which represent the path through which the current parser position was reached from the initial parsing position. Additional elements to the stack are added with *shift* actions, and multiple stack entries are removed and consolidated into one with a *reduce* action. The top element of the stack always represents the current parsing state. Out of the two actions, computationally more expensive is the reduce action.

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

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

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

A more involved and recent approach for reducing stack activity is reduction incorporated parsers [23]. At the cost of significantly increased number of parser states (and thus transition table), it is possible to record target state index as part of reduction entry in transition tables. As a result, such parsers in many cases no longer need two separate transition table lookups to perform a single reduction. Where a typical reduction entry contains only the non-terminal symbol being reduced and the reduction length, an incorporated reduction entry additionally contains a target state index which determines the next state of the parser, thus eliminating the third step of reduction sequence.

```
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

The cost of a reduction in SEVM

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

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

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

Reduction incorporation in SEVM

Optimized rules in SEVM are entirely defined by a call specifier and the currently used grammar. Consider the grammar shown in fig. 44. When optimizing `main` rule, the direct call to rule `ABCD` would be replaced to a dynamic call with call specifier `[REDUCE_ID (:ABCD), 0]`. Then optimizer would use this call specifier to drive subset construction and generate the optimized version of `ABCD`.

Because `ABCD` starts with both `AB` and `CD` rules, both of these rules would be merged into optimized version of `ABCD`. Continuing this process recursively, optimizer would merge `ABCD`, `AB`, `A`, `B`, `CD`, `C` and `D` rules in this order. Eventually MIR code shown in fig. 45 would be produced (to make the MIR more readable, DFA extraction was disabled).

```
#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: Optimized MIR for `ABCD` grammar rule

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

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

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

```
#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: Optimized MIR for `ABCD` grammar rule (with partial reduction incorporation)

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

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

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

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

Fig. 46 shows the optimized MIR for `ABCD`, but with partial reduction incorporation enabled. With this MIR, parsing character `a` is significantly more straightforward:

1. Task 0 starts execution in basic block `#0`.
2. Task 0 executes `#0` and matches character `a`, which transfers control to `#1`.
3. Task 0 internally reduces `A`, transferring control to `#5`.

4. Task 0 internally reduces `AB`, transferring control to `#7`.

5. Task 0 reduces `ABCD`, causing the callee of `ABCD` to be resumed.

6. Task 0 is discarded with `CtlStop`.

Only a single instance of a task is now needed (instead of 4). Furthermore, calls that are part of the prefix no longer require `CtlMatchSym`, because the control transfer of internal reductions is handled directly by `CtlReduceShort`. As a result, the reduction incorporated version of `ABCD` performs 3 less reductions and 1 less task suspension. This optimization also yields significant performance gains in real-world programming languages, as shown in chapter 6.7.3.

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

5.4.5 Garbage collection

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

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

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

- Entries that have active tasks within them will always be needed later and as such must not be freed.

- Parsing is typically done sequentially, with relatively little significant jumps due to ambiguities/backtracking.

- Ambiguities and backtracking are typically localized.

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

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

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

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

Such strategy of garbage collection may not be optimal (and may lead to significant slow-downs in worst-case ambiguity scenarios), however it works well when used with real-world programming language grammars: heap usage and processor time profiling reveal that the parser's runtime uses only minor amounts of memory (with the garbage collector enabled, the parse-tree becomes the largest memory consumer in `north`, followed by the index map of the chart), while taking insignificant amount of time to execute (below 5% of total execution time with ANSI C and Rust grammar tests).

5.5 Avoiding exponential complexity

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

This had several positive effects:

- Any form of infinite left recursion was eliminated, because it would result in two identical fibers in the same state.

- Exponential complexity of parsing was eliminated, when using trace with reduction duplication and incremental parse-tree construction: multiple reductions of the same type and length were merged together, multiple resumptions of same task but with different reduction were also merged when the task was resumed in the same instruction pointer.

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

Firstly, it's important to identify the situations that can lead to exponential complexity (and potentially hidden infinite recursion). We call these situations *conflicts* (the term is inspired by shift/reduce and reduce/reduce conflicts of (G)LR parsers [17]), as they potentially may lead to multiple parse paths. There are four types of conflicts in SEVM:

- **Reduce/reduce conflict**. These conflicts occur when two reductions of same type and length occur at the same starting origin. Resuming a task with both reductions may lead to exponential parsing complexity, because the same task will be resumed with duplicate reduction twice, which may lead to further conflicts.

- **Resume/resume conflict**. These conflicts are closely related to reduce/reduce conflicts. They occur, when two reductions occur of the same length, but with different *reduce_id* and result in resumption of the same task, in the same *state_id* twice. The existence of resume/resume conflict indicates that a rule has an ambiguous, but fixed-length prefix with matching suffix. Performing both resumptions means that the matching suffix is parsed multiple times, potentially leading to further conflicts.

- **Call/call conflict**. These conflicts occur, when the same non-terminal rule is called multiple times at the same position. Executing both calls would mean that the same input segment is parsed multiple times with same grammar rule(s). The callees may perform further calls, which may lead to more conflicts and/or infinite hidden recursion.

- **Match/match conflict**. These conflicts occur, when the same task is suspended at the same position twice. Accepting both matches may lead to scenario, where one reduction would awaken both tasks, which may lead to other conflicts (and exponential parsing complexity).

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

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

Call/call conflicts can be eliminated by making use of the following observation: whenever a new task is called, 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 bit mask, where the bit mask represents the set of concrete rules that were called. By performing bitwise-or operator between these masks it is possible to efficiently recreate the set that represents all the concrete rules that have been called so far at this position. If the newly called task is a subset of the previously called concrete rules, then the completion of the call can be aborted, because all of the currently called rules have been called before.

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

5.6 Parse-tree construction

SEVM constructs SPPFs automatically as described in chapter 4.7.1.

Automatic parse-tree construction was chosen over manual AST construction due to the following reasons:

- **Less-noisy grammars**. Including AST construction statements and expressions to 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 easier grammar merger, as now it is 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 minimal amount of memory and are laid out sequentially in heap (the same cannot be said about traditional ASTs, which may contain additional fields needed for further compilation steps). This ensures that creation of new SPPF nodes is cheap. Furthermore, all constructed SPPF nodes can be erased from memory in one sweep.

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

6 Evaluation of SEVM

6.1 Overview of evaluation process

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

6.2 Language selection

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

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

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

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

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

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

6.3 Implementation selection

The following parser implementations are included in this evaluation:

- **north**: it's the implementation of SEVM described in this work, written in Rust programming language.
- **bison** with **flex**: `bison` is a yacc-compatible LALR(1) parser generator. It is perhaps de-facto LALR(1) parser generator. It it/was used in various prominent open-source projects, such as: Bash, GCC before v3.4, Perl 5, PHP and others. It is commonly taught in universities and has integrations for wide variety of programming languages. Because `bison` works only with tokens (it's not a scannerless parser), a lexer is needed in order to be able to parse textual inputs. As such, lex-compatible `flex` was chosen, which is commonly used in conjunction with `bison`.
- **yaep** with **flex**: `yaep` is one of the few complete (as of writing this work) implementations of Earley parser with various optimizations to make it suitable for use in practise. It is also a non-scannerless implementation, and thus is used in conjunction with `flex` during evaluation.

- **dparser**: `dparser` is a scannerless implementation of GLR parsing algorithm. It is one of the very few still maintained projects capable of generalized scannerless parsing. Therefore `dparser` is the closest match in this list to `north`.
- **syn**: `syn` is a parser for Rust programming language, implemented with a hand-written recursive descent parser. It is a non-scannerless parsing method, but comes with it's own lexer and as such, no external lexer is needed. `syn` is primarily used as a library for developing language extensions for Rust programming language.

6.4 Comparison method

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

Each scenario is comprised of the following steps:

1. An input file containing the source code to be parsed is read.
2. An accurate measurement of system time is made called $start\_time$.
3. The input file is lexed, if the parsing method being tested requires a dedicated lexer. Otherwise this step is skipped.
4. The input is parsed. Some of the parsing methods may produce parse trees or abstract syntax trees during parsing.
5. An accurate measurement of system time is made called end_time.
6. $end\_time - start\_time$ of each scenario is added to a vector.

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

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

6.5 Test environment

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

- **Processor**: Intel i9-3930k.
- **Memory**: 16 GB of DDR3 RAM, 1333 MHz.
- **Operating system**: Ubuntu 18.04.1 LTS.
- **Linux kernel**: 4.15.0-36.
- **GCC**: version 7.3.0.
- **rustc**: version 1.30.0-nightly (90d36fb59 2018-09-13).
- **flex**: version 2.6.4.
- **bison**: version 3.0.4.
- **dparser**: version 1.30.
- **yaep**: obtained from GitHub with revision 1f19d4f5.

6.6 Test data

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

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

Table 23: A chart showing 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

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

6.7 Test results

6.7.1 Relative performance comparison

The relative performance comparison results of different parser implementations are shown in table 23.

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

`yaep` parser is slightly less performant than `bison`, but it's significantly more general, as it's an Earley parser. It still requires the use of dedicated lexer, however, no

Table 24: Table showing the median time needed to parse `input_gcc_470k.i` with and without garbage collection in `north`

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

Table 25: Table showing the median time needed to parse `input_rust_650k.rs` with and without garbage collection in `north`

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

semantic analysis was performed during parsing, because Earley parsers can produce ambiguous parse forests to represent different parse paths.

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

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

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

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

6.7.2 Performance influence of garbage collector

The primary purpose of garbage collector in SEVM/`north` is to reduce memory usage of the parser. It works, as described in section 5.4.5, by periodically scanning all of the currently existing chart entries and removing the ones that are believed to be no longer needed. Because the unneeded entries are identified by a heuristic, it is possible that the garbage collector may remove a chart entry that will be needed in the future. When that happens, SEVM runtime has to recreate the required chart entries by reparsing corresponding input fragments.

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

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

To see the actual performance impact of parsing ANSI C and Rust, additional tests were carried out: ANSIC (`input_gcc_470k.i`) and Rust (`input_rust_650k.rs`) inputs were parsed both with and without using garbage collector. The results of these tests are displayed in tables 24 and 25.

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

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

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

6.7.3 Performance influence of incorporated reductions

Partial reduction incorporation (described in chapter 5.4.4) is a further optimization made possible by performing MIR subset construction. In short, whenever a reduction is performed, SEVM runtime checks the list of suspended tasks in the origin entry of the task that is performing the reduction and resumes the appropriate task. This process is consists of several steps that are computationally expensive:

Table 26: Table showing the time needed to parse `input_gcc_470k.i` with and without reduction incorporation in `north`

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

Table 27: Table showing the time needed to parse `input_rust_650k.rs` with and without reduction incorporation in `north`

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

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

In order to test the effect of reduction incorporation on parsing performance, additional tests were carried out: ANSI C and Rust inputs (`input_gcc_470k.i` and `input_rust_650k.rs`) were parsed both with and without enabling partial reduction incorporation. The results of these tests are shown in tables 26 and 27.

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

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

```
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 test `north` grammar

Table 28: Table showing the benchmark results for parsing `input_a_1k.txt` with left and right recursive grammars

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

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

6.7.4 Performance influence of recursion type

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

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

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

```
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 test `north` grammar

Table 29: Table showing the benchmark results for parsing `input_5a_10k.txt` with left and right recursive grammars

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

The inputs are then parsed with grammars shown in figures 47 and 48 both with and without reduction incorporation.

The results for parsing `input_a_1k.txt` are shown in table 28. This test scenario triggers quadratic complexity when performing right recursion, and therefore right recursion is on average two orders of magnitude slower than left recursion. This is a well known characteristic of Earley parsers, and is inherited by SEVM/`north` as well. Optimizations to eliminate quadratic complexity of right recursion in Earley parser exists [18], however they are not implemented in `north`.

The results for parsing `input_5a_10k.txt` are shown in table 29. In this scenario, the difference in parsing times between left and right recursion is significantly lower, because the recursion depth is limited to 5 layers of rule calls (as opposed to 1000 in the previous test). This represents a more realistic scenario, because of the following observations:

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

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

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

6.8 Validity

6.8.1 Internal validity

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

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

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

6.8.2 External validity

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

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

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

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

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

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

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

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

- Many existing programming and mark-up languages have been designed with simpler parsing methods in mind: primarily, many of such languages can be parser either with simple LALR(1) parsers or with recursive descent parsers.
- Very few languages require any semantic analysis to be performed during parsing (C/C++ are the exception to this rule). The languages that do require semantic analysis for parsing, can be parsed in `north` or other generalized parsing methods ambiguously and filtered after parsing [31]. As such, ANSI C language and it's input can be considered as the worst-case real-world scenario regarding the ambiguity of the input grammar in `north`.

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

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

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

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

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

6.9 Conclusions

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

However, further research is needed in the following topics:

- Further SEVM optimizations. SEVM may be further optimized by eliminating external stacks and driving execution with native recursion, much like it is done in Packrat parsers [10].
- Greedy calls and ordered choice. These additional operations should be added to SEVM to boost its disambiguation capabilities.
- Error reporting. `north` currently implements no error reporting, but this can be done by analysing the contents of *susp_list* in the final chart entry.
- Error recovery. SEVM aborts execution upon encountering first parse error. It should be modified, so the parsing process may continue (by skipping fragments of invalid input). Error recovery algorithms for Earley parser exist, but none of them are designed for scannerless parsing [1]. There have been some work on error recovery in SGLR parsers [30], but it's uncertain how well such method may translate to SEVM.

7 General conclusions

This thesis began with a single wish: a wish for an extensible programming language. To ensure that it was clear what was meant by an extensible programming language, we firstly defined what an extensible programming language is. Then, we set on and searched for all the languages that suit our criteria. After our search was done, we realized that very few such languages exist and all of them come with various restrictions and limitations.

One of the reasons why so few extensible programming languages exist is their implementation difficulty: not only the compiler (or interpreter) has to be extremely generic, the parsing algorithm used to parse such languages has to be able to accommodate dynamically changing grammars. In fact, the lack of generic and high-performance parser might be one of the reasons why there are so few extensible languages.

To ensure that this is indeed the case, we defined the requirements for an extensible parser and analysed all available parsing algorithms and their implementations that even remotely matched our criteria. After the analysis, we concluded that no existing parsing algorithm is fully suitable to parse such extensible languages: the ones that do exist are either so inefficient that they can only be used as proof-of-concepts or so much generality is sacrificed in favour of increased parsing performance that it makes defining new syntactic language extensions problematic.

As a result, by using the ideas of existing parsing algorithms (primarily from Earley parser and its derivatives), we created a new parser called Earley Virtual Machines. It is a virtual-machine based, generalized context-free parsing algorithm that supports parsing adaptive grammars. We also created a grammar language for defining other languages that exposes additional internal features of EVM that are not present in more conventional parsing algorithms. These features allow defining new languages in more modular fashion and can be used to extend existing grammars without modifying them.

After testing our prototype implementation of EVM, we concluded that additional changes and optimizations are needed to EVM so it could maintain acceptable performance. These changes and optimizations were incorporated into a successor of EVM, which we called Scannerless Earley Virtual Machines (or SEVM for short).

As the name implies, the main focus of SEVM is the scannerless aspect of the parser: faster terminal symbol matching, cheaper (performance-wise) token-level disambiguation, just-in-time grammar to machine code translation and other improvements. An improved language for defining grammars was created for SEVM to enable access to some of these new features. To ensure that SEVM possesses improved performance, an implementation for SEVM was created (called `north`).

This implementation was tested to ensure that our optimizations yield meaningful performance improvements. Furthermore, the performance of `north` was compared to other parser implementations (such as the commonly used `flex` and `bison` combination). We found that SEVM, while being less performant than more constrained/specialized parsing algorithms, possesses impressive performance for a generalized scannerless parser that can parse adaptive grammars as a bonus.

As a result, we can safely conclude that Scannerless Earley Virtual Machines satisfy all the requirements for parsing extensible programming languages, which was the goal of this thesis.

Appendices

A bench_parsers utility

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

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

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

The following benchmarks are available:

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

B north_cli utility

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

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

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

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

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

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

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

References

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

[2] J. Aycock and R. N. Horspool. Practical earley parsing. *Compututer Journal*, 45:620–630, 2002.

[3] C. Brabrand and M. I. Schwartzbach. The metafront system: Safe and extensible parsing and transformation. *Science of Computer Programming*, 68(1):2–20, Aug. 2007.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, June 2008.

[5] W. H. Burge. *Recursive Programming Techniques*. Addison-Wesley, Oct. 1975.

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

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

[8] G. Economopoulos, P. Klint, and J. Vinju. *Faster Scannerless GLR Parsing*, pages 126–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[9] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. *SIGPLAN Not.*, 46(10):391–406, Oct. 2011.

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

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

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

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

[14] T. Jim and Y. Mandelbaum. Delayed semantic actions in yakker. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, LDTA '11, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[15] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 417–430, New York, United States, 2010. ACM.

[16] L. C. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. *SIGPLAN Not.*, 45(10):918–932, Oct. 2010.

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

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

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

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

[21] L. V. Reis, R. S. Bigonha, V. O. Di Iorio, and L. E. S. Amorim. The formalization and implementation of adaptable parsing expression grammars. *Sci. Comput. Program.*, 96(P2):191–210, Dec. 2014.

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

[23] E. Scott and A. Johnstone. Generalized bottom up parsers with reduced stack activity. *Comput. J.*, 48(5):565–587, Sept. 2005.

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

[25] C. Seaton. A programming language where the syntax and semantics are mutable at runtime. Master's thesis, University of Bristol, 2007.

[26] T. A. Standish. Extensibility in programming language design. *SIGPLAN Not.*, 10(7):18–21, July 1975.

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

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

[29] E. Vacchi and W. Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

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

[31] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized lr parsers. In R. N. Horspool, editor, *Compiler Construction*, pages 143–158, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

Audrius Šaikūnas

EXTENSIBLE PARSING WITH EARLEY VIRTUAL MACHINES

Doctoral dissertation

Tehnological Sciences
Informatics Engineering (07 T)

Editor: ???