Fall 2022

Jparsec - a parser combinator for Javascript

Sida Zhong
San Jose State University

Jparsec - a parser combinator for Javascript

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Sida Zhong

December 2022

The Designated Project Committee Approves the Project Titled

Jparsec - a parser combinator for Javascript

by

Sida Zhong

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2022

Dr. Thomas Austin	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science

ABSTRACT

Jparsec - a parser combinator for Javascript

by Sida Zhong

Parser combinators have been a popular parsing approach in recent years. Compared with traditional parsers, a parser combinator has both readability and maintenance advantages.

This project aims to construct a lightweight parser construct library for Javascript called Jparsec. Based on the modular nature of a parser combinator, the implementation uses higher-order functions. JavaScript provides a friendly and simple way to use higher-order functions, so the main construction method of this project will use JavaScript's lambda functions. In practical applications, a parser combinator is mainly used as a tool, such as parsing JSON files.

In order to verify the utility of parser combinators, this project uses a parser combinator to parse a partial Lua grammar. Lua is a widely used programming language, serving as a good test case for my parser combinator.

ACKNOWLEDGMENTS

I thank Dr. Thomas Austin for his guidance and patience as this project developed.

SECTION 1

Introduction

1.1 Parser Problem

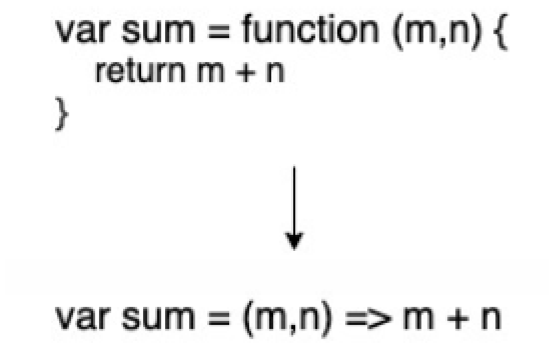
A handwritten parser today is like an editor or calculator 50 years ago. It is designed only for specialized tasks. A Java parser cannot be used to parse Javascript. To handle different languages, the parser must be redesigned from scratch. In contrast, Von Neumann designed as early as 1945 process programs like data, so that programs are only different in ordering CPU instructions [1]. A parser combinator is similar to the Von Neumann architecture in hardware. The same parser combinator can be used to parse different languages. The benefits of doing so are obvious. First, languages have many similarities, such as loops and conditional expressions. People use different syntaxes to describe the same logic. These commonalities can be abstracted as a parse pattern, which can then be inherited in other languages. Secondly, the grammar of some languages contains smaller grammatical structures. For example, Lua's expression contains three assignment expressions, and the assignment expression also contains the variable expression. These sub-expressions can be collocated or integrated into other grammatical structures as small modules to achieve encapsulation. The inheritance and encapsulation mentioned above are object-oriented programming concepts in software. The code in the software can be split into module tests and easily replaced and refactored. However, these are out of reach for the traditional parser due to the high coupling and immutable structure.

1.2 A Javascript library

Currently, most parsers for mainstream languages have not yet used the concept of the parser combinator. The first parser combinator is Parsec, designed by Frost, Hafiz, and Callaghan in 2008 [2][3]. Parsec is a parser combinator written in Haskell that supports left-recursive grammars. Although the Parser was very successful, it was mainly used as a tool, not an industrial parser. Haskell is a functional programming language with a very different style of programming than object-oriented programming, and has a high learning curve. Advanced concepts such as monads in Haskell push away a lot of people. In short, Haskell parsec is a parser combinator that is difficult to use. After the introduction of Parsec, Scala has also successfully introduced some parser combinators. But Scala’s design is similar to Haskell and has the same problems. Odersky in 2015 [4] states "Scala is a gateway drug to Haskell, ". The reason parser combinator are more popular in a functional programming languages is because of its unique mechanism. Functional programming is taking a function as an argument to another function so that a function can produce another function. Based on this feature, the parser combinator can pass an independent parser as a parameter to another parser, so a parser can also generate another parser. In the parser combinator, not only is the token parsed, but the parser itself.

In addition to the functional programming language, the high-order functions of other languages also meet the requirements, such as Javascript.

Figure 1: Convert general function to lambda function



```
var sum = function (m,n) {
   return m + n
}
```

```
var sum = (m,n) => m + n
```

The arrow in Figure 1 is a Javascript lambda higher-order function, also known as the arrow function. The higher-order functions of javascript are elegant and concise. In addition to the excellent design of high-level functions, the Javascript community has also been very active, consistently ranked among the top three most popular languages. The reason to use Javascript is not only to avoid complicated functional programming but also because of the overall advantages of Javascript, such as simplicity, popularity, and versatility. In addition, this project will explore the feasibility of a parser combinator in scripting languages.

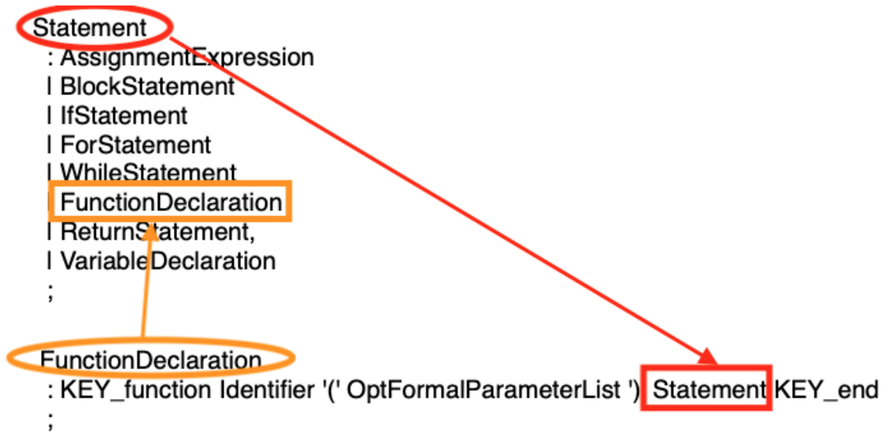
1.3 Key features of Jparsec

The Jparsec library of this project can record the current parsing status and detect syntax errors. It also has the same Parsec-way interface of the operators, such as selection and sequence. Apart from these same features, this project has four differences:

- **The Backus–Naur form (BNF).** A BNF is a language for describing a context-free grammar proposed by John Backus, which is used on a standard parser or parser generator. The Lua parser application of this project is written as a Backus–Naur form (BNF) expression. Usually, parser combinators are written according to specific needs, but this is only used as a tool. Once the grammar scales up to the level of a whole programming language, there must be a standard specification such as BNF.

- **Lookahead.** The lookahead approach uses a terminal to decide the next correct production, a common approach to the LL(k) Recursive-Descent Parsing algorithm. The difference in this project is that lookahead is determined by the terminal and the nonterminal symbol. This parser combinator does not distinguish between nonterminals, terminals, production, and start(NTPS) in context-free grammar like a traditional parser. Both terminals and nonterminals belong to the same parser and can be treated in the same way. Lookahead is looking at the next parser, not the next token.
- **Lazy evaluation.** Lazy evaluation strategies will not evaluate all the expressions at once, but evaluate the expressions when needed, so it is also called “call by need” evaluation. Since Javascript does not have the lazy evaluation feature of Haskell, this project has done lazy evaluation for all parsers. The reason for this is not only to improve performance but for compatibility with BNF expressions.

Figure 2: BNF expression conflict.



```
Statement
  : AssignmentExpression
  | BlockStatement
  | IfStatement
  | ForStatement
  | WhileStatement
  | FunctionDeclaration
  | ReturnStatement,
  | VariableDeclaration
  ;

FunctionDeclaration
  : KEY_function Identifier '(' OptFormalParameterList ')' Statement KEY_end
  ;
```

As shown in Figure 2, BNF expressions with eager evaluation have logical conflicts. The Statement contains FunctionDeclaration, and FunctionDeclaration also contains Statement. In this case, the order of functions cannot be defined, and eager evaluation will report an error that the function cannot be found.

- **Generator function.** Generator functions do not return a value, but use a special keyword “yield” to return multiple currently running results. At the same time, the value of the previous yield can be re-passed as a parameter to the subsequent yield. The generator function can maintain the current parser state. Its principle is like an assembly line, each parser is processed one after another in the generator function, and the parsers are passed to each other with higher-order functions. In the generator function, non-functional programming syntax, such as iteration or if statement, can be used to control the parser stream of higher-order functions, which makes it possible to use it to write powerful parser combinators with complex logic.

To sum up, this project adopts some traditional parser approaches such as lookahead and BNF expressions. It also includes the features of functional language parser combinators, such as lazy evaluation. Finally, it combines them using Javacript-specific generator functions.

1.4 Contributions

- JParsec combines functional programming features of parser combinators with features taken from more traditional parsers, such as BNF expressions and lookahead. To the best of my knowledge, this combination of features has not been used in previous parsing libraries.

- I implemented Soshnikov D’s algorithm for transforming left-recursive grammars [18], which was not previously implemented for a parser combinator library to the best of my knowledge.

1.5 Guide to the rest of the paper

This paper is divided into five main sections. Section 2 briefly describes the parser and parser combinator. Section 3 introduces the top-down and bottom-up parsing strategies, as well as the various parsing algorithms. Section 4 is an in-depth walk-through of parser combinator implementation and principles. Section 5 describes how to use parser combinators. This section constructs a complete Lua parser step by step from BNF expressions.

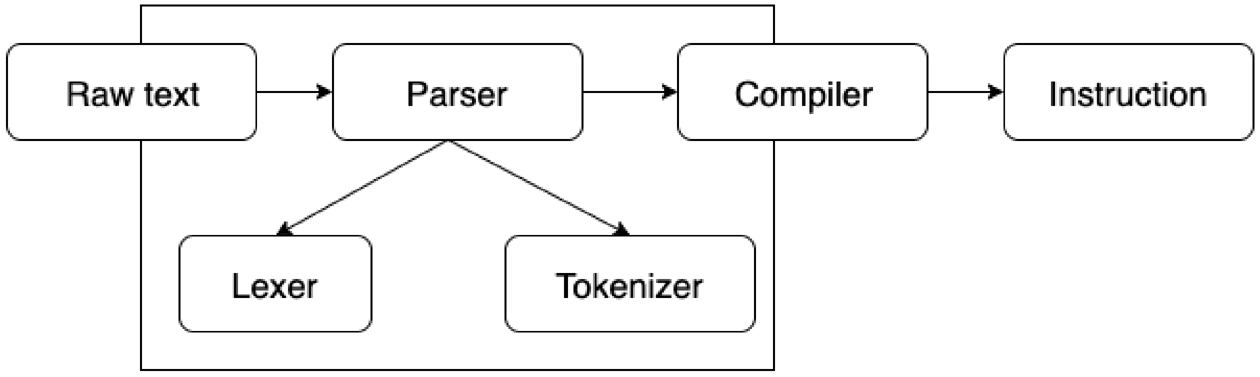
SECTION 2

Background

2.1 Parsers

A parser plays an essential role as a bridge connecting human and machine communication. The development of the parser directly determines the grammar development of programming languages. The programming language is a way for human beings to express the existing world to a machine. The grammar contains only a few logical symbols, such as `loop`, `if`, `class`, and `function`. These few symbols can express any known human logic or even create artificial intelligence. Programming languages are Turing-Complete. Programming code is not a simple string combination, but contains a complex logical structure behind it, like a circuit board. Early computers did not have programming languages. Machines were operated by obscure instructions. Therefore, people created programming languages, a language closer to natural language, to replace these instructions. The purpose of a parser is to convert a programming language into a data structure containing instructions. The generated data structure is called an abstract syntax tree, a top-down tree describing operations and logic. The compiler will use the AST to generate executable programs. The final AST generated by different programming languages is the same, because the AST transcends the syntax limitation and is a structure that describes the world logically.

Figure 3: A parser life cycle



As shown in Figure 3, the life cycle of parsing can be subdivided into three parts: lexer, tokenizer, and parser. The first step is the lexer, which is to convert the raw text into a steam of meaningful tokens, such as stripping spaces, newlines, and comments. The second step is tokenizer, in which regular expression is used to grab matching tokens, and assign these tokens to meanings such as numbers, letters, or special symbols. The last step is parsing, written in the Backus–Naur form (BNF) in an industrial-grade parser. BNF is a notation to describe a context-free grammar consisting of nonterminals, terminals, production, and start(NTPS). The tokens generated in the second step belong to terminals and to start symbol. The parsing process will combine nonterminals from production to recursively descend to terminals step by step. This type of parsing algorithm is called recursive-descent, and is currently the most popular and powerful parsing approach.

2.2 Parser combinators

In computer science, a combinator means a lambda function without free variables. Combinators make the parser process always modify the same variable, thus reducing the coupling between parsers. The purpose of the parser combinator is the same as the parser. It will generate the same AST in the end, but the difference is that it uses the divide-and-conquer strategy. The parser combinator will first divide the raw text into several sub-texts according to patterns. A sub-text may also contain multiple sub-subtexts, and each divide represents a new sub-parser. There is no essential difference between each sub-parser. Even the smallest sub-parser has all the functions of parsing and the final generated AST. Finally, these sub-parsers are combined into a new parser with “glue”. The so-called "glue" is a technique of using higher-order functions. For example, “sequence-of” defines a sequence of several consecutive parsers, and “choice-of” selects from different parsers.

Figure 4.1: A sequence-of parser combinator

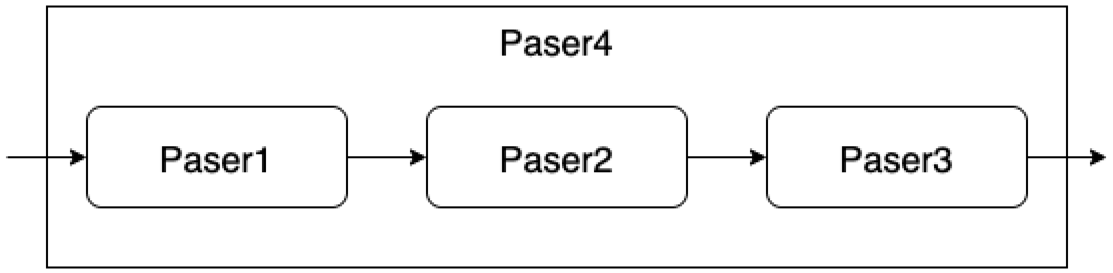
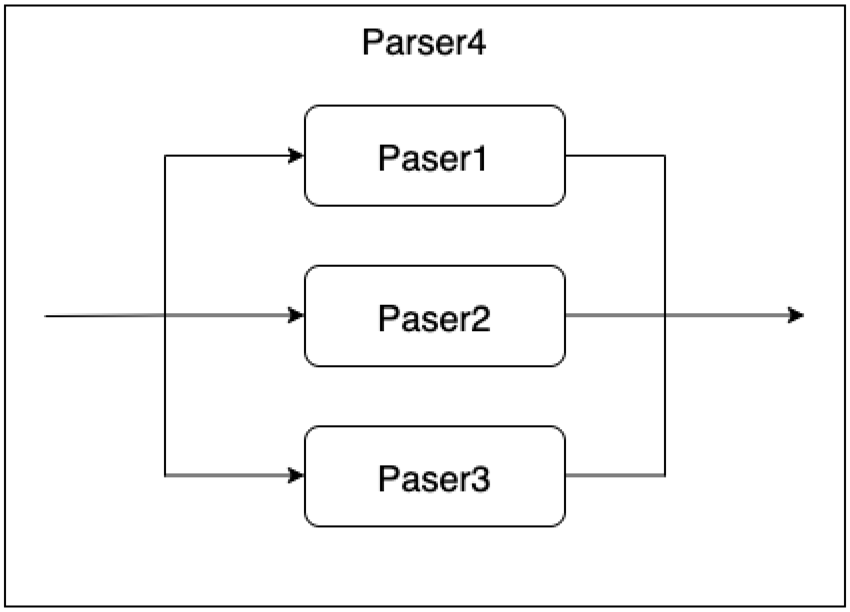


Figure 4.2: A choice-of parser combinator



As shown in Figure 4, a parser combinator is like the hardware circuit diagram. The "combinator" is the motherboard, which provides the consistency of the overall template, and the "parsers" are the various electronic components. Sequence-of represents the parser in parallel, and choice-of is the parser in series. A language can be parsed by assembling all the electronic components according to the rules. If the combination is changed, another language can be parsed. Briefly, a parser combinator is a parser for multiple languages, a technique for manipulating language syntax. Its core idea is to focus on syntax in different languages rather than to develop new grammar.

SECTION 3

Related work

3.1 Context-free grammar

Giving an alphabet ∑={a,b} that only contains two characters, "a" and "b." The language is an infinite combination of any possible strings or permutations of any length from the alphabet L(∑)={ab,ba,aa,bb,ababa….}. However, If we limit the combination length to only two. The set of strings becomes finite L(∑)={ab,ba,aa,bb}. Therefore, the characters can be arranged and terminated. So, grammar is a set of restrictions on a language. Chomsky [5] divided grammar into four levels.

- Unrestricted, natural language.
- Context-sensitive, programing language description
- Context-free, programing language parsing
- Regular, regular expression

Context-free grammars (CFG) are mainly used for the parsing process. Its grammars are described as a tuple with four elements G=(N,T,P,S). “N” stands for the nonterminal, “T” stands for the terminal, “P” stands for production, and “S” stands for starting symbol. For example, consider S->aX, X->b. The nonterminal are {S,X}, which is described as a variable to represent rules. The terminals are {a,b}, specific characters or tokens in a language. The production is the grammar itself. It contains all rules {S->aX, X->b}. The start symbol is the first produced symbol {S}, the first symbol to start parsing. A grammar has to follow three rules to become a context-free grammar. First, there may only be one nonterminal on the left side. Second, the right side can have terminals, nonterminals, or a mix. Third, the left side cannot have context. The third rule means if a nonterminal “X” appeared on the right side of a production `{S->aX}`. The left-hand side of the production `(X->b)` must have precisely one nonterminal “X”, not multiple “X” with ambiguous definitions.

3.2 Backus-Naur form

Inspired by formal grammar, Backus [6] created another grammar to describe CFG called the Backus-Naur form (BNF). As a powerful metasyntax, BNF is a notation to describe another programming grammar. It is the core of parser design. The syntax of BNF is very similar to CFG. For example, consider a math BNF `E::=E+E|E*E|number`. The separator for the left and right is `:==` to indicate the derivation process. `E` is a nonterminal symbol that can recurse deeply in the grammar. `number` is a terminal symbol that stands for a type of token. `|` means or, which a new left-handed production replaces. Using the example BNF to parse a mathematical formula `1+2*3`. The first process is to use the production `E::==E+E` to substitute rules that match the mathematical example `1+2*3`. The production derives from the left side to `E:==1+E, E:==1+E*E`, then `E:==1+2*3`. If the production starts the right side derivation first, the production becomes `E:==E+E*E, E:==E+2*3`, then `E:==1+2*3`. The result is the same, but the process is different. These two derivation strategies correspond to left-to-right leftmost derivation(LL) and left-to-right rightmost derivation(LR). However, in this BNF example, neither LL nor LR algorithms can be used as a parser, because they both face the problem of ambiguous semantics. The

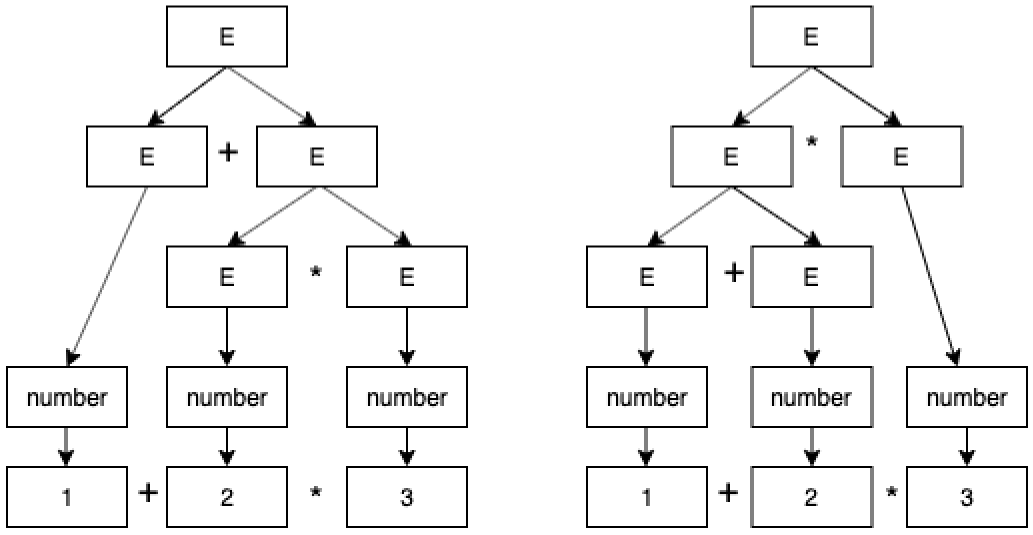
BNF generated by CFG usually has ambiguity in a grammar. It is necessary to remove the unambiguity in a grammar before parsing.

3.3 ambiguity in a grammar

An ambiguity in a grammar means the presence of the same grammar in a BNF but produces different results, which can lead to different parsing trees and unintended effects during compilation. Ross [7] criticized this problem in an article discussing context ambiguity, and stated that some attempts to resolve ambiguity problems have not yet been satisfactorily resolved. Ross pointed out that in addition to matching tokens, parsing is more essential to identify the sequencing of interpretation, the direction of the scan, and the determination of scopes.

In the example of section 3.2 `E::= E+E|E*E|number`, the parsing process chooses the first production `E::==E+E` to start derivate. However, if the process chooses another production `E::==E*E` to start, it will get completely different results.

Figure 5: different precedence results in two parse trees



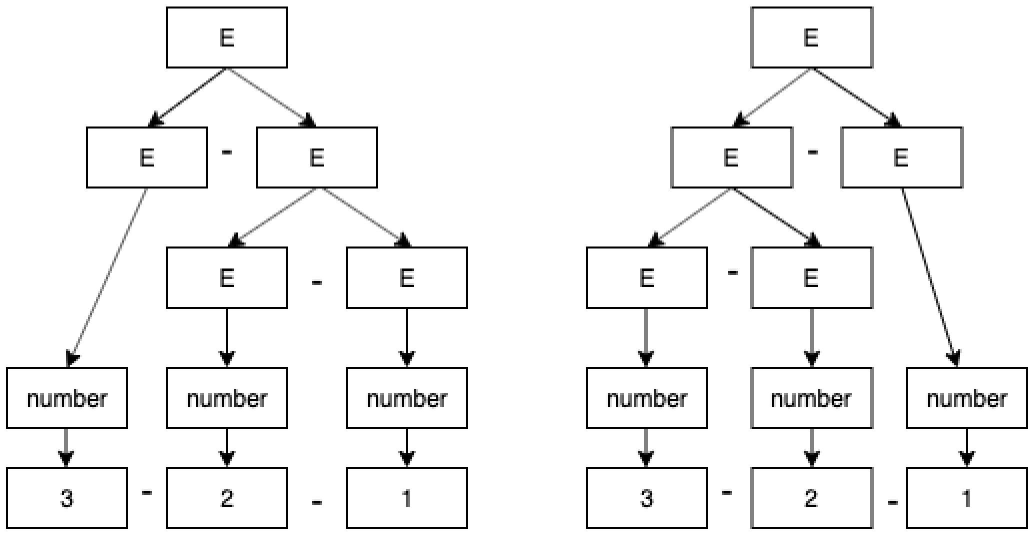
According to the parsing tree generated on the left, multiplication has a higher priority. But in the tree generated on the right, the addition will be done first, and then the multiplication, which is wrong. The problem is that the parser doesn't know which production to start with is correct, because productions have different precedence. Different precedence is a common cause of ambiguity in a grammar. The solution to the precedence problem is to introduce a new layer of nonterminals to enforce the correct productions, i.e. `E:==E+E`, starting first.

```
//ambiguity in a grammar                //unambiguity in a grammar
E:==E+E                                 E:==E+T|T
   |E*E                                 T:==T*F|F
   |number                              F:==number
```

The production `E:==E+E` transforms to `E:==E+T|T`. The production `E:==E*E` transforms to `T:==T+F|F`, and adds a new production `F:==number`. Since the start symbol is E, the parser must first choose `E:==E+T`, then `T:==T*F`, to get an unambiguity in a grammar by eliminating the same left factor.

The second major cause of ambiguity in a grammar is the association of parsing. The associated direction is not derived (LL or LR) as described in section 3.2.

Figure 5: different association results in two parse trees



For example, consider parsing a mathematical expression `3-2-1`, with the BNF being `E:==E-E|number`. The correct way is to do the operation of `3-2` first, then do the operation of `-1` to get the result of `0`. However, if the operation does `2-1` first, then subtracts by `3`, the result is `2`, which is not correct. These two operations correspond to left-associative and right-associative operations, and subtraction can only do left-associative operations in mathematics. The grammar `E:==E-E|number` is an ambiguity in a grammar that fails associated rules. The parser cannot choose whether to start the first E or the second E.

```
//ambiguity in a grammar            //unambiguity in a grammar
E:==E-E                             E:==E-number
   |number                             |number
```

The solution is to force the parser to be left-associative with left-recursive grammar. The production `E:==E-E|number` transforms to `E:==E-number|number`. This way, the parser will always deduce the `E` on the left, avoiding ambiguous semantics. The left-recursive grammar is elegant and straightforward. However, not all parsing algorithms are compatible with left-recursive grammars, such as LL. All industrial-level parsing combinators use the LL. This is why the parser combinator has not been able to become the parser of mainstream languages.

3.4 LL parser

LL and LR algorithm, mentioned in Chapter 3, are the two main strategies of parsing. These two parsing algorithms have been at war with each other. The LR algorithm has dominated the parsing algorithms for modern languages like Java. The main reason why LL algorithm is not popular is that it cannot solve left recursion problems efficiently. The LL-constructed parsers are also known as top-down parsers, which was proposed by Edwin and Lewis [8]. Shortly after the LL algorithm was published, Rosenkrantz and Stearns [9] applied the LL algorithm in CFG to verify its correctness. The top-down parser's process starts at the top of the parse tree and goes to the bottom until it reaches a terminal. The root is the grammar's start symbol, making the logic very natural to understand. The prototype of the LL parser uses a backtracking algorithm. The backtracking algorithm traverse every node of the parsing tree. Whenever the parsing attempt fails, it returns to the previous token and creates a new attempt. Many articles (Watson[10], Birman [11]) criticize the performance issues of this algorithm. The worst-case time complexity is $o(n)^3$. Another upgraded version of the LL parser is called the LL(k) parser, where k represents the following lookahead token. The look-ahead mechanism was first introduced in LR parsers by Deremer and Pennello [12]. It was later introduced into the LL parser by Edwin and Lewis [5]. The lookahead predicts the next production based on the following K tokens, so this kind of parser is also called a predict parser or a recursive decent parser. This project uses the algorithm of the recursive decent parser.

3.5 Left Factoring

However, the LL(K) recursive descent parsers do not entirely avoid backtracking algorithms. For example, a BNF is `E::=T+E|T`. The two productions `T+E` and `T` have the same prefix `T`. The same nonterminal prefix represents two productions, causing the lookahead to fail to predict the correct one unless one fails, leading back to the backtracking algorithm again. To improve the performance of the LL(K) parser, the backtracking algorithm must be avoided. The solution is to convert the BNF grammar to left-factored.

```
E::=T+E                                 E::=T+E′
   |T                                   E′::=+E
                                           |ε
```

Left factoring means extracting grammar that follows the same prefix into separate rules. The new production is `E'::=+E|ε`, where ε means nothing. In this case, the lookahead token split from the same `E` into `E` and `+`, thus avoiding backtracking issues while keeping the grammar the same. Another practical example is the dangling Else problem.

```
Statement ::= if Expression then Statement
            | if Expression then Statement else Statement
```

The above BNF is a non-left-factored grammar, which has the same prefix production `if Expression then Statement`. Using the left factoring technique, it will transfer to a left-factored grammar.

```
Statement ::= if Expression then Statement Statement′
Statement′ ::= else Statement
             | ε
```

After left factoring, the production can be predicted by lookahead token `if` or `else`.

3.6 Left recursion

Sections 3.4 and 3.5 mentioned two approaches of lookahead and left-factoring to improve the performance of the top-down parsers. However, there is still one problem left. The top-down type of parser cannot solve the left-recursion problem. For example, consider BNF `E::=Ex|y`. To parse grammar `x`, the parser first parses nonterminal `E`, then it will try to parse `x` again, resulting in an infinite recursive loop. The parser jump between `x` and `E`, having no opportunity to consume any tokens. The grammar `Ex` uses the nonterminal `E` itself as a substitute for deriving infinite productions. A grammar with such properties is called a Left recursion grammar. The solution is to convert the BNF grammar to the right recursion.

```
E::=Ex                                   E::=xE′
   |y                                    E′::=yE′
                                            |ε
```

The process is to put all terminals on the left to ensure that each recursion has terminals consumed. Then duplicate the nonterminal symbol `E` into `E` and `E'` with two productions separately. Processing could be more complicated in indirect left-recursion syntax, like E:==ET, T:==TF, F:==FE. Suonio [13] gave a more specific and complete solution to eliminate left recursion in his paper, but the overhead cost is high. The most significant advantage of the top-down parsers are that it parser language similarly to how humans comprehend language. But this advantage is compromised by the left recursion problem. The transformed right-recursive grammar is no longer elegant and is prone to errors. This makes the top-down parsers not powerful for modern programming languages.

3.6 Parser combinator

Here, I brief review the history of parsing leading to parser combinators. A CFG defines a programming language. Then a CFG is further abstracted into BNF. BNF extends EBNF to solve the problem of grammar ambiguity. EBNF leads to the performance problem of backtracking. To solve backtracking, lookahead and left-factoring methods are designed. Finally, an alternative solution is to use a right-recursive transformation to solve the left-recursive problem. Top-down parsing evolution has been stagnant for a long time, during which many algorithms were trying to solve the left recursion problem. The memoization algorithm proposed by Johnson greatly enhances the performance of top-down parsers [14]. This algorithm introduced the idea of dynamic programming (DP). The purpose of the DP is to optimize performance. The core idea is to maximize the reuse of the calculated results and avoid repeated processing. However, Johnson pointed out that his method does not solve the left-recursion problem. It only improves the time complexity of the top-down parser. The real breakthrough in top-down parsing was the idea of a parser combinator proposed by Frost et al. [2] in 2008. The parser combinator is not an algorithm, but a novel and sophisticated parsing implementation. The core idea is to disassemble a parser and then reassemble it according to different situations. In 2010, Danielsson [15] verified that the parser combinator could solve the problem of left recursion based on the research of Frost et al.; furthermore Danielsson praised the elegant syntax and flexible structure of parser combinators. The earliest application of the parser combinator was Haskell’s Parsec library. Moors et al. [16] used a similar approach to invent a parser combinator for Scala. Scala is a hybrid of object-oriented and functional languages. More and more modern languages have embraced parser combinators, and have their own parser combinator tools.

SECTION 4

Implementation

This chapter shows how to construct a parser combinator from scratch in Javascript. This process will start with the most basic parser, and then gradually expand to a library of parser combinators.

4.1 Terminal parser

The parser combinator is composed of different parsers, so the first step is to build a simple parser.

```
const str = s = targetString =>{
 if(targetString.startsWith(s)){
   return s
 }
}

const parser = str("hello world")
```

`Str()` is a parser that can parse strings. When the target string is "hello world," the method in `str()` will match each string and return the result. Otherwise, it will throw an error message. This is a simple parser, but there are a few problems. First, the method for matching strings is `.startswith()`, which can only match fixed string formats, not patterns, such as letters or numbers. So this parser is missing a tokenizer. The tokenizer is an important part of parsing. The purpose is to allow the parser to recognize different types of strings. This parser’s tokenizer can be implemented using the regular expression method. Regular expressions are a method of normalizing languages proposed by Stephen Cole Kleene in 1950. It is used to match and retrieve documents that conform to specific rules.

```
if(targetString.match(/^[A-Za-z]+$/)){
 return s
}
```

The regular expression `/^[A-Za-z]+$/` is a tokenizer that matches strings like "hello world". This parser is used as a terminal or start symbol in applications. Another problem is that the parser lacks a mechanism to maintain the state. Returning a parsing result is not enough. Additional information may be required during the interaction of multiple parsers.

```
const initialState = {
  target,
  index: 0,
  result: null,
  isError: false,
  error: null
};
```

The initial state consists of 4 parts. A target field will display the following string to be parsed, an index field will record the current parsing position, an error field will indicate an error occurred, and an isError field will interrupt the parser process. This information is used as a parser state, allowing the parser to read/write information. The next parser can take the parser state and transform it into another new one. This way, a fixed pattern can be formed for each parser, that is, `Parser => ParserState in => ParserState out`.

4.2 Nonterminal parser

The above methods can construct multiple different types of parsers. However, these parsers cannot be combined, because the methods in these parsers can only recognize different strings. This requires a special parser to recognize different parsers instead of strings. This particular parser acts as a nonterminal in BNF. For example, the BNF expression of a while loop in Lua is `<while> ::= "while" <Condition> <BlockStatement>`. This while iteration logic comprises a `“while”` parser, a `Condition` parser, and a `BlockStatement` parser. A parser representing sequence can connect these three parsers, which act like glue. The difference between the nonterminal and the terminal parser is that the terminal parser takes the target text and parses it into a specific string type. In contrast, the nonterminal parser takes parses and converts them into a specific logical type.

```
const parser = sequenceOf([
   str("hello"),
   str("world"),
])
```

The `sequcenOf` parser takes multiple parsers, and passes each parser state to the next parser in turn, and finally returns the last parser state. Nonterminal parsers like sequenceOf obey the rule `parser => ParserState in => ParserState out`. Similar to `sequenceOf`, `choiceOf` can express `expr::=<term>|<factor>` in BNF, `many()` expresses right recursion in BNF like `<term>::=<term>{","<term>}`. This nonterminal parser can be seen as a grammar encapsulation in traditional parsers, such as `while` and `if` in the form of higher-order functions.

4.2 High-order function

Terminal parser and Nonterminal parser have two things in common. First, they are both parsers with the same purpose: to recognize different data types and parse them into specific formats. Second, they have the same function; the essence is to take one parser state and transform it into another. Based on these two factors, their concepts can be expressed by a parser object. This parser object will execute the function of parsing the parser state as a parameter inside it. In other words, the entire parsing process is achieved by passing parser objects one by one, and the specific parsing method will be passed as a parameter of the object.

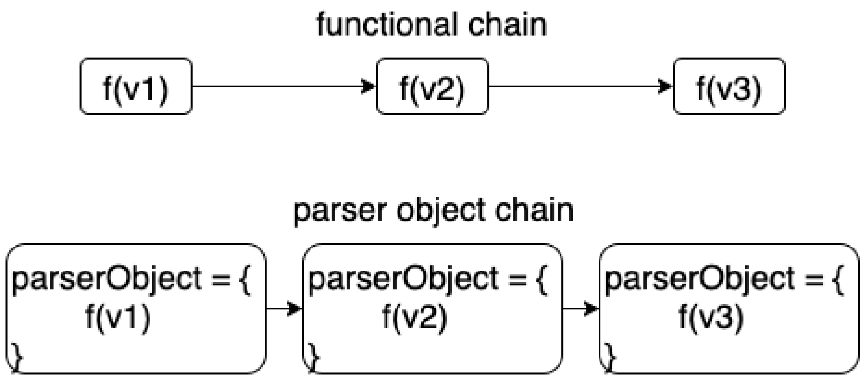
```
class Parser {
 constructor(parserStateTransformerFn) {
   this.parserStateTransformerFn = parserStateTransformerFn;
 }

 run(target) {
   return this.parserStateTransformerFn(target);
 }
}
```

The `parserStateTransformerFn` as a higher-order function in the constructor will be passed as an argument to the parser class and execute the parsing process.

```
const str = parsers => new Parser(parserState => {
 //...
})
```

The `str()` will take another parser as an argument and return a new Parser object. The specific parsing method is assigned to the new Parser object by the higher-order function `parserState => {...}` as a parameter.

Figure 5: Function chain vs parser object chain



As shown in Figure 5, in functional programming, the functions interact with values. However, in this project, the functions interact with a parser object, and change the parser state value inside. The parser object is like a piece of furniture made of wood, which is always a piece of wood, but will continue to change into different shapes until the final shape is completed. The final shape represents an AST with a complex structure.

4.3 AST format

The result field in the parser state is the AST. When the nonterminal parser processes multiple parser states, the results need to be combined according to a certain AST format. For example, the BNF expression for Lua's addition operation `1+1` is `<Addition> ::= <Left> <Operator> <Right>`. The sequenceOf parser needs to process the Left, Operator, Right three parsers continuously, and generate data in AST format `['operator':+,'left':1,'right':1]`. However, the AST in this example is an object with a specific key, and the order of the keys differs from the BNF expression (the AST Operator comes first, but the BNF comes second). Therefore, when updating the parser state, the result field cannot generate different AST data structures according to other parsers, nor can the order of parsing results be changed.

In this case, a `map()` function is introduced to solve this problem. Unlike Javascript's internal function `map()`, this parser `map()` does not work with arrays but with multiple parsers.

```
map(fn) {
  return new Parser(parserState => {
    const nextState = this.parserStateTransformerFn(parserState);
    return updateParserResult(nextState, fn(nextState.result));
  });
}
```

The `map()` is a function in the Parser class, which can be called in any parser. In addition, the `map()` must return a parser class to ensure that the parser object chain is not interrupted. The parameter fn in `map()` is the function of actually constructing the AST, and the result in the previous parser state is updated with `fn()` in the newly generated parser object. In general, this new `map()` function uses Javascript's native `map()` function as a parameter to pass to a parser object, then acts on the result in the parser object.

```
const parser = sequenceOf([
   left,
   operator,
   right
]).map((left,operator,right)=>{
   return {
       'operator':operator,
       'left':left,
       'right':right
   }
})
```

The AST construction function is passed as a parameter to `.map()`, so the results of `left`, `operator`, and `right` in the previous sequenceOf parser can be regenerated according to the AST structure.

4.4 Lookahead

The lookahead is a common recursive descent parser mechanism known as a "predictive parser." It can predict specific grammar rules based on the next K tokens. In the traditional LL or LR parser, every time a token is consumed, the lookahead mechanism is triggered, and the next BNF expression is executed according to the type of the lookahead token. Unlike the traditional lookahead method, the lookahead of this project will look for a specific parser instead of a token.

```
const ReturnStatement = sequenceOf([
 Return,
 Lookahead(Expression)
])
```

For example, in Lua's return statement, `ReturnStatement ::= 'return' {Expression}`. The Expression is optional. After parsing the `'return'` token, the parser will lookahead to the next `Expression`. The `Expression` is a complete parser that may contain more parsers. If the lookahead `Expression` can be parsed, the `'return'` parser will chain the `Expression` parser. Otherwise, it will be skipped.

4.5 Lazy evaluation

Javascript is an eager evaluation language, which means that the program will evaluate each function or parameter before calling it. This allows eager evaluation to make the code more transparent in its execution, making the program more restrictive.

However, eager evaluation is not suitable for a parser combinator. A parser combinator is an infinite data structure that contains many recursive nonterminal functions. It cannot define a logical precedence order, and the control flow of a function is an abstraction rather than a primitive one. Therefore, the parser must be converted to lazy evaluation, also known as "call-by-need." All the parsers will be evaluated when needed.

```
//error message:                        //no error
//parser_next is not defined
const parser = parser_next()            const parser = () => parser_next()
```

An error occurs when Javascript calls an undefined value. However, if the undefined value is put into a function, no error will be reported as long as the function is not called. This undefined value is a parser.

```
const parser = lazy(() => parser_next())
```

`lazy()` is a parser object like all parsers. Each parser will wrap a lazy parser like a shell. In this way, when the program executes to an uncalled parser, it will not evaluate because the parser is in the lazy function. The Lazy parser is like a pipe parser. It lets the parser pass through it and does nothing else.

4.6 Generator

```
const generator = function*()
   yield 1
   yield 2
   yield 3
}
generator().next() //1
generator().next().next() //2
generator().next().next().next() //3
```

A generator represented by `function*()` is a special javascript function. Its essence is an iterator consisting of the keyword `yield` and the function `next()`. When calling `next()`, `function*()` doesn't execute all contents, but returns a `yield` object containing the value and an indicator of whether the process is complete. After every `yield` value has been executed, `function*()` will return. Furthermore, calling `next()` can pass arguments, and the arguments will be reassigned to the yield value. Each yield returns a parser object, and the parsing method is performed on the parser object, and then the new parser object is passed as a parameter to `next()`. In this way, generators can be combined with functional programming, and provides asynchronous communication between each parser, which creates a powerful parser combinator.

```
const parser = generator(function* () {
 a = yield number("1")
 b = yield number("2")
 c = yield number("3")
 return [a,b,c]
});
```

The newly defined generator will pass an original generator `function*()` as a parameter. All parsers will be executed in the yield in `function*()` and return a defined data structure `[a,b,c]`.

```
const generator = generatorFn => {
 return succeed(null).chain(() => {
   const iterator = generatorFn();

   const runStep = nextValue => {
     const iteratorResult = iterator.next(nextValue);
```

```
      if (iteratorResult.done) {
        return succeed(iteratorResult.value);
      }

      const nextParser = iteratorResult.value;
      return nextParser.chain(runStep);
    };

    return runStep();
})}
```

The generator is a parser. It will take the `function*()` passed in the previous step, and return a parser object through `succeeded(null)`. The `succeed(null)` function call is a dummy parser object to convert the generator from a function to a parser object. The `.chain(() => {...})` will be called after `success(null)`. This method is similar to `.map()`. It takes the passed parser function as a parameter and uses it to convert the next parser object. In `chain()`, the `runStep()` will execute recursively until all yield parsers have been executed.

SECTION 5

Experimentation

This section shows how to build a Lua parser using the parser combinators written in this project. This process will start with the basic syntax and gradually cover the grammar for an imperative subset of Lua.

Lua is a programming language similar to tables and schemes created by Roberto Ierusalimschy in 1993, known for being lightweight (200K source code size) and for its high performance. Jparsec is a recursive-descent parser. The recursive descent parser has several advantages. A complex grammar can be implemented with intuitive algorithms. Some parsing rules may be achieved by manipulating the parser state. The recursive-descent parser uses a top-down parsing strategy to recursively remove the nonterminals to terminals. The top-down parsing strategy introduces the left recursion problem. The left-recursive solution is explained in this section using the right-recursive transformation method.

5.1 Lexer

As mentioned in Section 2.1, the first process in the parser is the lexeme. Without a lexer, the parser would mistake meaningless tokens like comments, spaces, and newlines. These tokens meant to help humans understand code and work efficiently, but have no meaning to machines. The purpose of the lexer is to filter at these types of tokens before parsing.

```
const LETTER = Lexeme(new Parser(parserState => {
 return tokenizer(parserState,'LETTER')
}));
```

```
const NUMBER = Lexeme(new Parser(parserState => {
 return tokenizer(parserState,'NUMBER')
}));
```

There are multiple implementations of `Lexer`. In this project, `Lexer` uses a regular expression to filter each terminal parser result. In other words, `Lexer` is a parser specially used to chain the terminal parser to remove unnecessary tokens. It takes the terminal parser as an argument, and consumes the matching results without updating any parser state. One question is, why not filter the entire text using regular expressions? In Lua, the syntax for an `expression end` does not have any token representation, but in other languages, it usually ends with "`;`". Some parsers like `expression end` overlap with the lexer's regular expression. In this case, the lexer filters for a specific parser, not all text. The lexer reflects the flexible features of a parser combinator. Any functionality can be encapsulated into a parser object. The parser state inside an object can be manipulated to achieve different results. For example, updating the parser index instead of the parser result skips some tokens like the lexer. On the contrary, updating the parser result instead of the parser index predict specific tokens like the lookahead mechanism.

5.2 Tokenizer

As mentioned in Section 4.1, tokenizers function as terminal parsers. The first step in implementing terminal parsers is building a Lua tokens vocabulary. The tokens of Lua are divided into four types.

- The variable type, such as `NUMBER, STRING, TRUE, and FALSE`.
- The logical type, such as `+, >, =, and &&`.

- The special characters, such as `(`, `{`, `:` and `;`.
- The keywords, such as `IF`, `ELSE`, `WHILE`, and `FUNCTION`.

```
const tokenizer = (parserState,type)=>{
 var {target,index,isError} = parserState;

 // consume tokens
 const slicedTarget = target.slice(index)

 // match tokens
 const regexMatch = slicedTarget.match(Token[type]);

 // update parser state
 index += regexMatch[0].length;
 return updateParserState(parserState, index, regexMatch[0]);
}
```

I map these four types of tokens to their corresponding regular expressions, and use regular expressions to separate each token. When a BNF expression recursively descends to the terminal process, the terminal parser consumes a token and increments the parser index, which means that one text token has been successfully parsed and is ready for parsing of the following text. Finally, the parser state information is updated and passed it to the next parser.

5.3 Left recursion

At this point, this parser combinator can derive a single token, such as the keyword `IF` or the number `1`. The next step is to generate a statement list. The statement list contains either one statement or several different types of statements.

```
statementList
   :== statementList statement
     | statement
```

The way it defines statement lists in BNF is by using recursive grammar. In the example above, the same nonterminal symbol `statementList` appears in the far left position, which results in this syntax being left recursion.

```
A :== Aα|b                          A :== bA′
                                    A′ :== αA′|ε
—--------------------------------   —-------------------------------------
statementList                       statementList
   :== statementList statement         :== statement statementList′
     | statement                    statementList′
                                       :== statement statementList′
                                         | ε
```

The solution is to convert left recursion to right recursion.

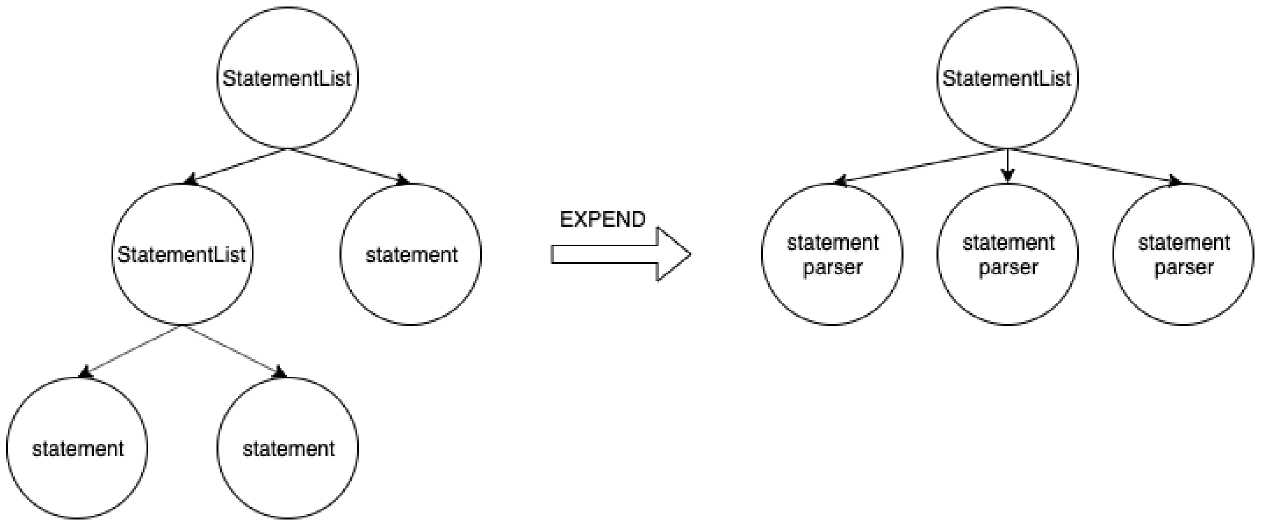
```
statementList                       statementList
   :== statementList statement          :== (statement)*
     | statement
```

The solution provided by this project is similar to the right-recursive transformation, but more elegantly, it uses iteration to extend the recursive parser.

```
const StatementList_parser = generator(function* () {
   var statementList = []
   while(yield lookAhead(STRING)){
      statementList.push(yield Statement_parser)
   }
   return statementList
});
```

The implementation is to define an array of statementList and use a while loop to push new statements into it continuously until no tokens are left. The solution is very straightforward, but behind it is the essence of the parser combinator. In the parser combinator, all parsers are combined with smaller parsers. This statement parser acts as an independent parser that can be manipulated with any data structure.

Figure 6: BNF grammar expends to a parser combinator



As shown in Figure 6, If the statementList is the root of the AST, then the statement parser is the leaf of the first level below, and the next step is to decompose the statement parser. With such a direct algorithm, there is no left recursion problem. The importance is to combine all the parsers based on the specific AST structure.

5.4 Right recursion

The statement parser consists of various types of parsers, such as the if statement, for statement, and while statement parsers.

```
statementList :== (statement)*
Statement :== WhileStatement
WhileStatement :== 'while' Condition BlockStatement
BlockStatement :== 'do' StatementList 'end'
```

Take the while statement parser as an example. Its BNF expression contains a BlockStatement. The BlockStatement contains a `statementList`. The `statementList` contains many statements. The while statement is one of them. The starting point is a while statement, and the ending point is a `whileStatement`, so this is a right recursion with nested structures. However, unlike left recursion, derivation occurs at the right position. Right recursion does not have the problem of infinite loops, because the tokens will be consumed with each recursion. Therefore, production will eventually be terminated.

```
const WhileStatement_parser = generator(function* () {
   var key_while = yield KEY_while;
   var condition = yield Condition_parser;
   var blockStatement = yield BlockStatement_parser;

   return [key_while,condition,blockStatement]
});

const BlockStatement_parser = generator(function* () {
   var key_do = yield KEY_do;
   var StatementList = yield StatementList_parser
   var key_end =yield KEY_end;
   return [key_do,StatementList,key_end]
});
```

The implementation uses a generator function, and each yield step is executed according to the BNF expressions syntax without any modification. Jparsec allows parsers to be defined in a similar way to grammar rules.

5.5 Left recursion chains

The binary expression parser is one of the statement parsers, and the most complex expression in many parser. It contains a series of sub-expressions with precedences. This chapter presents a solution to a binary expression by using left recursive chains.

```
if (false or 1+2*5 > 10 == true) then end
```

In the above `if` condition, there is a binary expression `(FALSE or 1+2*5 > 10 == TRUE)` that contains parsers with five different priorities.

- "*" represents a multiplicative parser
- "+" represents a additive parser
- ">" represents a logic parser
- "==" represents a equality parser
- "or" represents a relation parser

The logic flow starts with the multiplication of `2*5`, then the addition of `2*5+1`, then the logic comparison with `2*5+1 > 10`, then the equation of `2*5+1 > 10 == TRUE`, and finally, the relation comparison `FALSE` or `2*5+1 > 10 == TRUE`. Each of these five binary parsers is a left recursion grammar.

```
AdditiveExpression
   ::= NUMBER
     | AdditiveExpression OPERATOR_ADDITIVE NUMBER
```

```
const AdditiveExpression_parser = generator(function* () {
   var left,right,op;

   left = yield NUMBER
   while(yield lookAhead(OPERATOR_ADDITIVE)){
       op = yield OPERATOR_ADDITIVE,
       right = yield NUMBER
       left = [op,left,right]
   }
```

```
    return left
});
```

Take the additive parser as an example, the solution is the same as the `statement` parser in section 5.3; that is, using a while loop to expend the `AdditiveExpression` in the second production.

```
MultiplicativeExpression
    ::= NUMBER
      | MultiplicativeExpression OPERATOR_MULTIPLICATIVE NUMBER
```

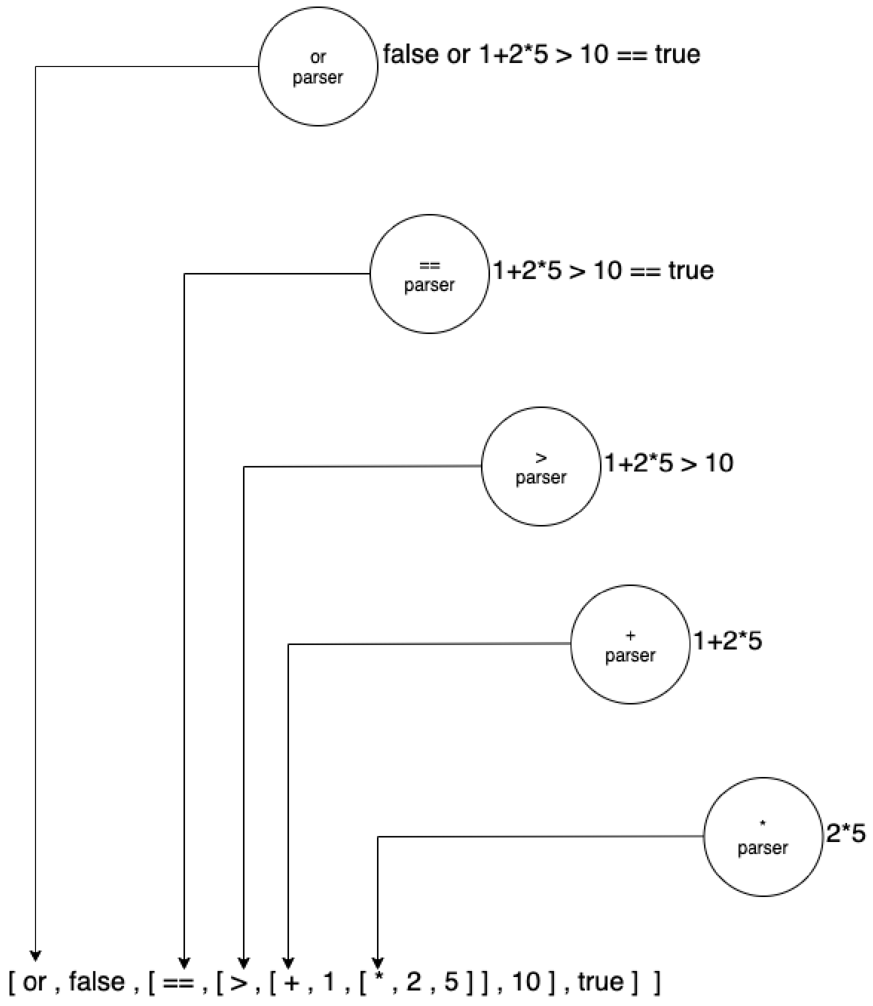
The BNF of the multiplicative and the `additiveExpression` is the same except for the operator. However, multiplication takes precedence over addition. In other words, to evaluate addition, it must first evaluate multiplication.

```
AdditiveExpression
    ::= MultiplicativeExpression
      | AdditiveExpression OPERATOR_ADDITIVE MultiplicativeExpression

MultiplicativeExpression
    ::= NUMBER
      | MultiplicativeExpression OPERATOR_MULTIPLICATIVE NUMBER
```

Replace nonterminal `NUMBER` in production of `AdditiveExpression` with `MultiplicativeExpression`. By concatenating the BNF of addition and multiplication, the `additiveExpression` is one or more `multiplicativeExpressions` followed by the + operator. In this way, the `additiveExpression` represents `multiplication + multiplication`. The `multiplicativeExpression` is either a number or multiple multiplications. Using the same concept, all binary parsers can be chained by their priority order.

```
const AdditiveExpression_parser = generator(function* () {
   var left,right,op;

   left = yield MultiplicativeExpression_parser
   while(yield lookAhead(OPERATOR_ADDITIVE)){
       op = yield OPERATOR_ADDITIVE,
       right = yield MultiplicativeExpression_parser
       left = [op,left,right]
   }

   return left
});
```

Figure 7: shows how the example `FALSE or 1+2*5 > 10 == TRUE` is parsed.



or parser
false or 1+2*5 > 10 == true
== parser
1+2*5 > 10 == true
> parser
1+2*5 > 10
+ parser
1+2*5
* parser
2*5
[or , false , [== , [> , [+ , 1 , [* , 2 , 5]] , 10] , true]]

6.1 Lua test cases

This section shows different parsing cases for Lua, including parser input and output. The input is a Lua file. The output is an AST of symbolic expression (s-expression) similar to a function programming structure. Jparsec covers most but not all Lua syntax applications. Due to the modularity of parser combinators, each Lua grammar corresponding to the parser can be combined and nested. In future work, the Lua syntax of the project will gradually improve. The different test cases are as follows.

Listing 6.1.1: variable declaration and math expressions

```
local a,b,c=1+2*3/4,5,6
```

```
{
  "target": "a=1+2*3/4",
  "index": 9,
  "result": [
    [
      [
        "=",
        "a",
        [
          "+",
          "1",
          [
            "/",
            [
              "*",
              "2",
              "3"
            ],
            "4"
          ]
        ]
      ]
    ]
  ],
  "isError": false,
  "error": null
}
```

Listing 6.1.2: while loop statement

```
while( true )
do
  a = 1
end
```

```
{
  "target": "while( true )\ndo\n   a = 1\nend",
  "index": 29,
  "result": [
    [
      "while",
      "true",
      [
        "do",
        [
          [
            [
              "=",
              "a",
              "1"
            ]
          ]
        ],
        "end"
      ]
    ]
  ],
  "isError": false,
  "error": null
}
```

Listing 6.1.3: if statement

```
if (4 > 1 + 2 * 3 == false)
then
end
```

```
{
  "target": "if (4 > 1 + 2 * 3 == false)\nthen\nend",
  "index": 36,
  "result": [
    [
      [
        "if",
        [
          "==",
          [
            ">",
            "4",
            [
              "+",
              "1",
              [
                "*",
                "2",
                "3"
              ]
            ]
          ],
          "false"
        ],
        "then",
        []
      ],
      null,
      null,
      "end"
    ]
  ],
  "isError": false,
  "error": null
}
```

Listing 6.1.4: function declaration

```
function f (a,b)
 return a+b
end
```

```
{
  "target": "function f (a,b)\n  return a+b\nend",
  "index": 33,
  "result": [
    [
      "function",
      "f",
      [
        "a",
        "b"
      ],
      [
        [
          "return",
          [
            "+",
            "a",
            "b"
          ]
        ]
      ],
      "end"
    ]
  ],
  "isError": false,
  "error": null
}
```

Listing 6.1.4: function call

```
f(1,2)
```

```
{
  "target": "f(1,2)",
  "index": 6,
  "result": [
    [
      "func",
      "f",
      [
        "argv",
        [
          "1",
          "2"
        ]
      ]
    ]
  ],
  "isError": true,
  "error": "ASSIGN: Got Unexpected end of input."
}
```

Listing 6.1.4: for statement

```
for i=10,1,-1 do
 a=1
end
```

```
{
  "target": "for i=10,1,-1 do\n  a=1\nend",
  "index": 26,
  "result": [
    [
      "for",
      [
        "=",
        "i",
        "10"
      ],
      "1",
      [
        "-",
        "1"
      ],
      [
        "do",
        [
          [
            [
              "=",
              "a",
              "1"
            ]
          ]
        ],
        "end"
      ]
    ]
  ],
  "isError": false,
  "error": null
}
```

Listing 6.1.4: table

```
a={b,c,3,{4,5,6,{7,8},{9}}}
a[b]=1
```

```
{
  "target": "a={b,c,3,{4,5,6,{7,8},{9}}}\na[b]=1\n",
  "index": 34,
  "result": [
    [
      [
        "=",
        "a",
        [
          "b",
          "c",
          "3",
          [
            "4",
            "5",
            "6",
            [
              "7",
              "8"
            ],
            [
              "9"
            ]
          ]
        ]
      ]
    ],
    [
      [
        "=",
        [
          "a",
          [
            "b"
          ]
        ],
        "1"
      ]
    ]
  ],
  "isError": false,
  "error": null
}
```

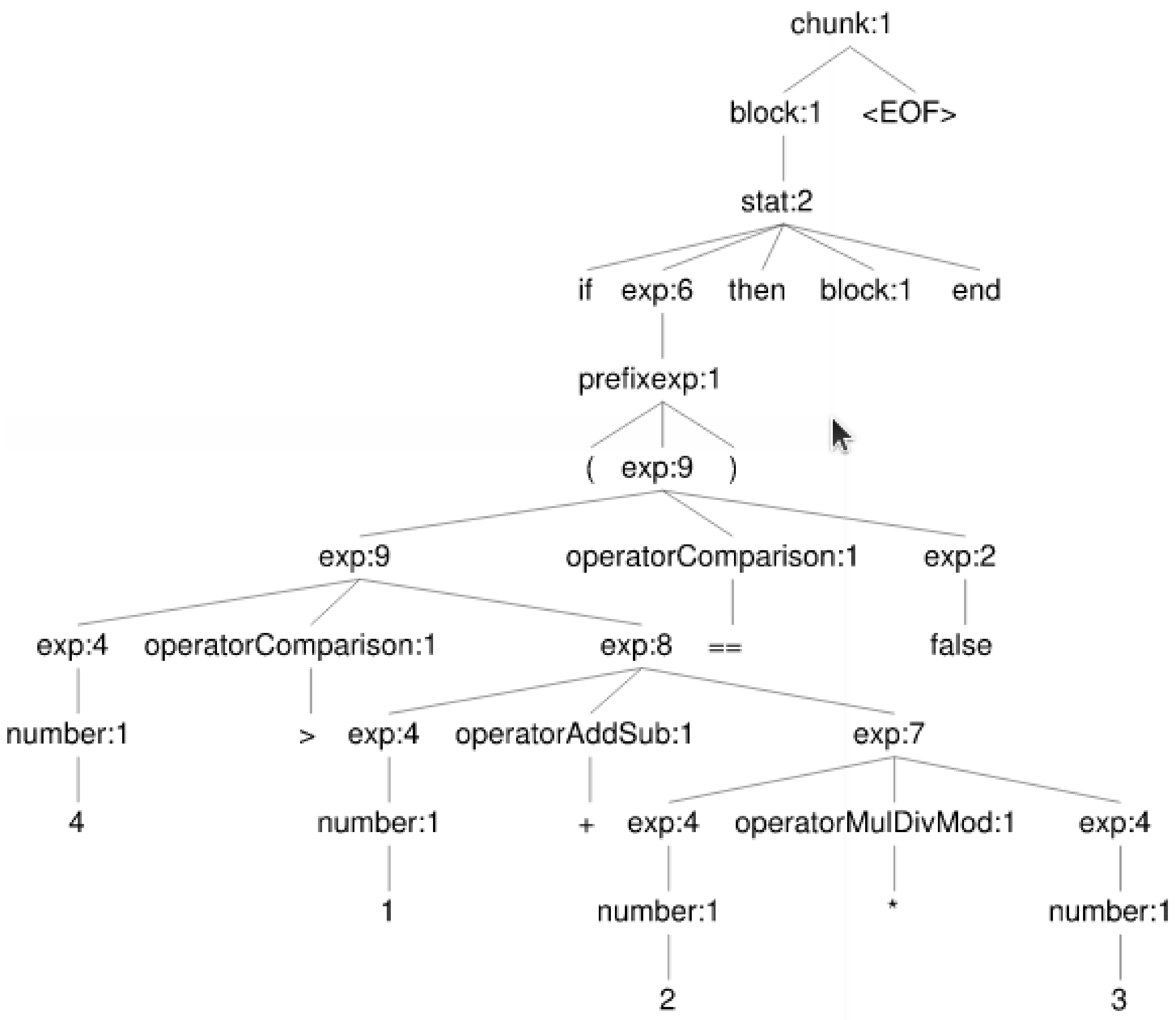
6.2 Comparison with ANTLR

This section compares Jparsec with ANTLR, especially the usage of binary expression. ANTLR was the first parser generator to use a top-down parsing strategy. A parser generator using a BNF grammar automatically generates source code for a parser. The Lua grammar for binary expressions in ANTLR is as follows.

Listing 6.2.1: ANTLR input grammar

```
exp
    : 'nil' | 'false' | 'true'
    | number
    | string
    | prefixexp
    | exp operatorMulDivMod exp
    | exp operatorAddSub exp
    | exp operatorComparison exp
    ;
```

Listing 6.2.1: ANTLR output parsing tree



The result shows that ANTLR gives exactly the same parse tree as Jparsec, with the correct precedence. The BNF `exp: exp operatorMulDivMod exp` indicates that ANTLR supports left-recursive grammar. Parr, Harwell, and Fisher also clearly pointed out that ANTLR supports left-recursive grammars, but does not support indirect left-recursive grammars because it is uncommon [19].

$$\begin{array}{l} A \rightarrow \alpha_1 A' | \ldots | \alpha_s A' \\ A' \rightarrow \beta_1 A' | \ldots | \beta_r A' | \epsilon \end{array}$$

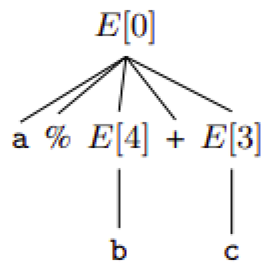
Their paper provide a detailed left recursion elimination method in ANTLR [19]. The elimination method converts left recursion into right recursion, which is exactly the same as Jparsec. However, the method of expressing precedence with the converted right recursion is different.

$$\begin{array}{ll} E' \rightarrow M\ (+\ M)^* & \text{Additive, lower precedence} \\ M \rightarrow P\ (\%\ P)^* & \text{Modulo, higher precedence} \\ P \rightarrow \textbf{id} & \text{Primary } (\textbf{id} \text{ means identifier}) \end{array}$$

ANTLR further converts right recursion into an iteration grammar [19]. The principle of Jparsec implementation is the same as that of ANTLR. The corresponding syntax is slightly different. However, ANTLR's implementation of the precedence grammar is completely different from Jparsec.

$$E[pr] \rightarrow \textbf{id}\ (\{3 \geq pr\}?\ \%E[4]\ |\ \{2 \geq pr\}? + E[3])^*$$

According to Parr's description of ANTLR [19]. Precedence is the use of prefix numbers to mark the correct order of evaluation in the parse tree.



(a) a%b+c
assoc (a%b)+c

In the above parsing tree, the larger prefix number of nonterminal is always evaluated first. in the examples in this section, The BNF of `exp operatorMulDivMod exp` is set to prefix1, `exp operatorAddSub exp` is set to prefix2, and `exp operatorComparison exp` is set to prefix3 in order. Evaluation starts from prefix 3 to prefix 1.

in conclusion, the ANTLR grammar is more consis than my Parsec solution, because ANTLR has an internal mechanism to transform the grammar. Jparsec implementations follow the grammar strictly. ANTLR precedence calculations show the heavy design of the parser generator. It shows a highly coupled internal system. If languages have different rules, it is difficult for ANTLR to modify the internal code. In fact, ANTLR has several libraries for different languages. Instead, Jparsec has only one library that supports different languages as long as the BNF grammar is correct. That clearly shows the advantage of a parser combinator.

SECTION 6

Conclusion

In the section 3 of this paper, the history of parser combinator is presented. From the initial CFG language classification to the BNF expression, to the processing of ambiguity in a grammar, to the lookahead(k) performance improvement, to the solution of left factor and left recursion. All these milestones led to the birth of parser combinators, and demonstrated their utility.

The parser combinator plays a very important role in parsing. However, unlike traditional parser combinators such as Parsec, Jparsec abandons functional programming and uses Javascript's objects and lambdas to create a new strategy for parser combinators. Moreover, some generic parser mechanisms are also used in Jparsec, such as look-ahead and BNF. Jparsec can effectively use the modularity of the parser combinator to support left-recursive or right-recursive grammars. This makes the parser construction code clear and concise, much like BNF. The lazy evaluation significantly reduces the coupling between each parser module, allowing Jparsec to retain human-friendly comprehension features, which is the biggest advantage of top-down type parsing algorithms. In the experiment of Lua parsing, Jparsec can support basic Lua syntax, including while loops, for loops, if conditions, etc. In some complex cases, such as binary arithmetic, Jparsec can combine look-ahead and left-recursion to handle grammar ambiguity caused by different precedences. This shows that the framework built by Jparsec can parse industrial-grade languages.

There are still further improvements that can be made to Jparsec. Parser combinators constructed with Jparsec are very close to the syntax of BNF, but it is also possible to embed a Javascript macro system in Jparsec to further improve brevity. The macro-formed syntax makes code look shorter, cleaner, and more readable. In addition, macros allow designers to avoid obscure and repetitive code and abstract complex logic into a short statement [17]. A parser combinator with a macro is expected to be driven precisely by BNF expressions. Designers can focus on grammar, and write BNF to generate parser combinators. Besides macro, Jparsec can also abstract more nonterminal parsers to represent the same type of grammar. A typical example is left recursion in Section 5.3. A left recursion can be further split into left-recursive factors and right-hand expressions. A further encapsulated left-recursive nonterminal takes two array parameters, expected as `leftRecursion([parser1],[parser2,parser3])`.

Parser combinators have parsing methods similar to human reading. The process of human language learning is to understand words first, then combine words into sentences, and finally connect sentences into articles by conjunction. However, the cost of human comprehension algorithms is performance. The performance of the parser combiner generally loses to the parser generator with the bottom-up algorithm. The bottom-up algorithm is difficult for humans to understand, but it is friendly to machines, and has no left recursion problems. This leads to the question, whose time is more important, human or machine? Whether human thinking should be closer to machines, or machine thinking should be closer to humans, the future parser should find a balance between performance and readability.

LIST OF REFERENCES

[1] Arikpo, I. I., F. U. Ogban, and I. E. Eteng. "Von Neumann architecture and modern computers." *Global Journal of Mathematical Sciences* 6.2 (2007): 97-103.

[2] Frost, Richard A., Rahmatullah Hafiz, and Paul Callaghan. "Parser combinators for ambiguous left-recursive grammars." International Symposium on Practical Aspects of Declarative Languages. Springer, Berlin, Heidelberg, 2008.

[3] Frost, Richard, Rahmatullah Hafiz, and Paul Callaghan. "Modular and efficient top-down parsing for ambiguous left-recursive grammars." Proceedings of the Tenth International Conference on Parsing Technologies. 2007.

[4] A. Moors, F. Piessens, and M. Odersky, "Parser combinators in Scala." *CW Reports 54* (2008).

[5] N. Chomsky, "Three models for the description of language." *IRE Transactions on Information Theory, 2(3):113–124,* September (1956).

[6] J. W. Backus, "c The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference." *In Proceedings of the International Conference on Information Processing, UNESCO, Paris,* (1959).

[7] D. T. Ross, "On context and ambiguity in parsing." *Communications of the ACM 7.2* (1964): 131-133.

[8] S. R. Edwin, and P. áM Lewis, "Property grammars and table machines." *Information and Control 14.6* (1969): 524-549.

[9] D. J. Rosenkrantz and R. E. Stearns, "Properties of deterministic top-down grammars." *Information and Control 17.3* (1970): 226-256.

[10] D. Watson, *A Practical Approach to Compiler Construction. Springer* (2017).

[11] Ford, Bryan. *Packet parsing: a practical linear-time algorithm with backtracking*. Diss. Massachusetts Institute of Technology, 2002.

[12] F. DeRemer, and T. Pennello. "Efficient computation of LALR (1) look-ahead sets." *ACM Transactions on Programming Languages and Systems (TOPLAS) 4.4* (1982).

[13] R. K. Suonio, "On top-to-bottom recognition and left recursion." *Communications of the ACM 9.7* (1966): 527-528.

[14] J. Mark, "Memoization of top down parsing." *arXiv preprint cmp-lg/9504016*

(1995).

[15] N. A. Danielsson, "Total parser combinators." *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (2010).

[16] A. Moors, F. Piessens, and M. Odersky, "Parser combinators in Scala." *CW Reports 54* (2008).

[17] Pang, Derek, "Pantry: A Macro Library for Python" (2018). Master's Projects. 657. DOI: https://doi.org/10.31979/etd.pydk-c57j

[18] Soshnikov, D. (2020) *Essentials of parsing.* Udemy.DOI: https://www.udemy.com/

[19] Parr, Terence, Sam Harwell, and Kathleen Fisher. "*Adaptive LL (*) parsing: the power of dynamic analysis.*" ACM SIGPLAN Notices 49.10 (2014): 579-598.