Parsing via Regular Expressions

Dávid Magyar
John von Neumann Faculty of Informatics
Óbuda University
Budapest, Hungary
magyar.david@stud.uni-obuda.hu

Sándor Szénási
John von Neumann Faculty of Informatics
Óbuda University
Budapest, Hungary
szenasi.sandor@nik.uni-obuda.hu

***Abstract*—This paper describes an approach of interpreting program code text using parsing expression grammar, building an abstract syntax tree as its interpretation progressively. The presented algorithm uses a declarative system of regular expressions applicable not to text, but to tokens and larger, already interpreted parts of the code - making the algorithm itself and its usage easy to understand. This approach allows parallelization of the interpretation process as well, which optimization would not be possible otherwise.**

***Index Terms*—program-code parsing, regular expressions, declarative parsing, imperative parsing, runtime parser, flexible parser**

I. Introduction

Since programming languages exist, parsing the code - primarily for compilation - is an ever-present necessity. As of today, there is a deep theory of this topic, describing approaches and useful constructs, categories and capabilities of parsers, which this paper does not intend to dive in deep, but to describe and use the official technical terms where possible. The presented approach of parsing utilizes regular expressions and forms a PEG (Parsing Expression Grammar [1]), which is more expressive than simply regular expressions [2]. This paper aims to present an approach specially for parsing complex input recursively using PEG approach. The rest of this document describes an easy to configure and understand interpreter based on regular expressions over characters, tokens and schemas.

The goal is to parse an input text - being an ordinary sequence of characters, like a `.txt` file - into a tree of nodes known as Abstract Syntax Tree (AST from now on). In this process the input text does not get modified, but its parts get associated with their interpreted meaning, like when parsing a program-code, then the "`class`" word within the input text might be interpreted as a keyword, being part of a class definition.

The resulting AST can be used to obtain information from the raw input text, like in case of input files such as configuration files or more often the AST of raw program-code text is used by the compiler to compile the code. The software and approach presented in this document aims to deliver the ability, so users can set up an easy to understand, but efficient enough parsing workflow, which can be executed on any input text to obtain its AST.

Although regex can be transformed to equivalent PEG [3], the presented parser uses PEG with recursive algorithm and which does not operate on the input text characters directly, but (after tokenization) on matched tokens and schemas (consisting of token and/or sub-schema matches).

II. Related Work

There are many different approaches for parsing. One of the primary categorization is whether a parser is bottom-to-top/bottom-up or top-to-bottom/top-down. A bottom-up parser first finds the smallest, atomic parts, which hold the details of the information, then gradually merges these small parts into larger concepts. A top-down parser does the opposite; finds the bounds of the largest concepts first, then knowing them, will parse the inner part of these larger concepts, searching for more detailed information on it. [4]

As an example of the usual use-case, the top-down parser of program code might find the concept of the class first, between its "`class`" keyword, name of class, opening bracket until its ending bracket. What is included between the brackets is not important yet and will parsed afterwards. Then, after the class concept has been recognized, its inner part gets parsed, searching for field definitions and methods. After then within methods: for statements, within statements: values and operators... On the other hand, a bottom-up parser intents to find the values and operators first, then the statements enclosing them, then their enclosing functions and neighboring fields, then the class enclosing the found fields and functions.

Usually the most efficient parser strategy is to use DFA - deterministic finite automaton. Basically using a rigid state-diagram of how the parts can follow each other. Such mathematical approach can result in fast parsers performing well, on the other hand, they might very well be hard to construct, maintain and extend by hand. For that reason, the DFA many times gets generated from declarative definitions given by the programmer, which the parser generator uses in order to construct the DFA and output the source code of a parser based on the given DFA. The downside of this approach is that it expects a valid sequence of input. In case part of the code is syntactically wrong, then it can mess up all the code below it (further parts of the input sequence) for the compiler, hence one user error results in tons of errors from the compiler - this can experienced while compiling C++ code. The reason for this issue is, that we - programmers - do not conceptualize or understand code as a sequence, but rather a tree of separate definitions, like classes in a file, functions

in a class, statements in a function, etc. Hence, a human approach of "manual compilation" would highly differ from the described DFA approach.

Parser generators are considered to be the professional way of doing parsers and which are considered superior to parsers based on regular expressions, mainly because of nesting and unexpected parts [5]. Meanwhile, there are existing parser libraries - similar to what is presented in this paper - mostly configured through some programming API [6]. Examples of such libraries: Myna, Parsimmon, and Chevrotain. Some consider writing a regex-based parser as an initial step before proceeding to use non-deterministic finite automata and then deterministic finite automata [7], [8].

Parsing via PCRE (regex implementation with additional features) is considered to be very useful at times. Although handled with skepticism for more complex tasks, it is indeed usable for the purpose [9].

III. Methodology

The parser described in this paper aims to work more alike how a human being would try to understand a program-code. As of usage and implementation, it should be parameterized up fast, does not require any output code generation, compilation, nor graphs or state-diagrams to work. On the other hand, letting the user give the definitions in a declarative way and expect the parser to follow the rules enhances usability greatly.

For this reason, the parser first loads its definitions from definition files runtime on start-up, buffers them as definition objects and uses these objects to parse the input text. It is one process, runtime. This leads to some concern about performance, but gives the programmer/user the ability to modify, add, or generate definitions runtime from anything, anytime. Also, these definitions does not parameterize a parser-generator, but the parser itself. Such definitions given declaratively are the token system and the schema system.

A. Token System

The lexer, which tokenizes the input text into recognized tokens, works based on token type definitions. Each token type definition is named uniquely, but can have one or more token matchers associated with, whose role is to find matching parts of the input text for recognition for the defined token type. Such matchers can be simply of exact characters, some characters with extra criteria of word boundaries around or an ordinary regular expression on character sequence. These defined token type definitions form a token system for the token parser to work with. Furthermore, token matcher of ranges (with beginning and ending parts, like quoted strings) is also getting added soon. Such token matcher functionality can be optimized and further extended later by choice.

B. Schema System

Then, from tokens we parse schemas. Schemas in this context is a pattern of tokens and/or other schemas followings one another as defined. Similarly to the token system [10], the schema parser, which parses from the recognized tokens, work from a schema system of schema type definitions. Schema type definitions are somewhat similar to token type definitions; are uniquely named and associated with some patterns. One particular schema definition is either a range or a sequential schema definition, which two are different in mechanics. A sequential one matches a sequence of tokens and/or schemas to form a new concept, but which concepts are complete in the sense, that they are considered fully parsed downwards and can be used upwards only in containing schemas. Range schema definitions on the other hand has a beginning and ending pattern, which breaks the bottom-to-top concept, being a top-to-bottom one. It's not a problem, rather a tool to write the grammar more freely and more similar to human understanding.

A schema type definition matches a pattern of tokens/schemas of a regular expression - consisting of tokens and schemas, not characters -, hence the name of the project. Since a custom regular matcher had to be implemented, it does not support all variety of regex functionality, although most are planned. For now, it supports sequence of items, groups, selection (OR) within groups, group quantifiers and lock-aheads. For the near future, capturing groups are also planned and might take a role in the parsing as well. With quantifiers, optional groups are also possible as *at least zero* `(0..)` quantifier.

Range schemas can also be defined, which consist of a beginning and an ending pattern and matches if the two are matched in the mentioned order. In order to handle nesting properly, only the innermost matches are matched within a cycle. To do this, the ending pattern must find a match and it gets associated with the closest beginning match preceding it. An interesting feature of this parser is that when a range - which might be a code block, for example - is matched, it does not make any statement about the content of the matched range. This content is considered a not-yet-used parsing area with no associated rule-set (being a set of schema definitions). When another schema makes use of this range schema match, then it can specify the rule-set for the content of the range - and with the rule-set enqueue that parsing region for parsing. This allows the grammar to separate the definitions of the blocks themselves and the definitions of the contents of the blocks. Such approach was inspired by the custom highlighting definitions of Visual Studio Code.

C. Implementation

The loading of the definition systems is the least interesting part - an indention-based parser for a YAML-ish, custom syntax was implemented. Plain YAML and JSON support are to be added later on. What is more interesting is the actual parsing mechanism, that parses based on the given definitions. First of all, we know, that token types and schema types all have their own matchers, which can be used.

D. Matchers

All these matchers are capable of the same thing essentially; when given a sequence of *items*, then it can indicate whether

it matches some part of it, and if yes, then from which index to which other index - *items* being text characters for token matchers and token/schema matches for schema matchers.

The regular expression matching is implemented in a left-most match manner, with an always eager capturing, with backtrack fallbacks when allowed by group quantifier. Lazy quantifiers are neither supported currently nor planned, but might be added later if in demand. The whole regular expression matching is based on the individual matchers of the definitions - allowing any kind of, even custom matchers to take part of the matching their own way. The only requirement is to provide the schema matcher interface, essentially to provide a match if can from an input sequence.

E. Priorities while Parsing

There is an additional feature to both token and schema definitions, which remains to be described here, because it is closely related to the parsing mechanism itself. In this parser, there is no priority defined between definitions other than via this mentioned feature; sections. Sectioning can be used to separate definition into separate sections, which are kept in the order of definition and which order is chronologically considered while parsing. In other words, definitions within sections coming sooner in a file (more up in it) will take place sooner in parsing as well, while definitions given in later sections will take part in parsing later. This causes the sooner sections to have some kind of priority over the later sections. There are two kinds of separators: hard and soft. Hard separator separates its before and after part into separate hard sections. Hard sections can be further separated into soft sections using soft separators.

1) Hard Sections: Hard sections are taken in sequential order and execute one-by-one. This means, that while parsing the first hard section, the content of the second hard section is completely irrelevant. When definitions of one hard section can find no more matches, the hard section is considered exhausted and the parser proceeds to the next hard section. Exhausted hard sections are also not relevant while parsing later hard sections. Hence, these sections are "separated hard". The parsing of the hard section ends and the hard section is considered exhausted when none of its soft sections find any new matches. For this reason, parsing a hard section can be considered recursive.

2) Soft Sections: Soft sections are within hard sections, so each hard section consists of one or more soft sections. Soft sections do still enjoy priority over each other in definition order, but their parsing takes place in the same exhaustive loop. This means, that soft sections are parsed as if they were hard sections, after each other, dominating each other, but then after the last soft section the first soft section of the hard section is taken again, whether it finds new matches from the matches found in later soft sections of the hard section. Hence, the clearest difference between hard and soft sections is, that a definition within a hard section cannot refer to a definition, which is parsed in later hard sections, while definitions in soft sections can refer to matches of definitions matched in later soft sections (within the same hard section).

F. Interpretation Algorithm

Interpretation is usually achieved through tokenization and parsing - so the interpreter consists of a lexer capable of tokenization and a parser. The lexer converts the input text into tokens, and the parser parses the tokens to an AST. In this algorithm, the AST is not directly generated from tokens, but through recursively matching schemas from tokens and other already matched schemas.

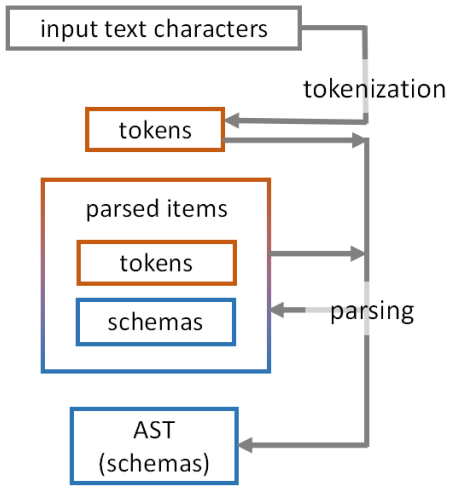

Fig. 1. Process of Interpretation

This figure simplifies the algorithm significantly, but at the highest abstraction level, it looks like this; the parsing runs in a loop, repeating until there are no more schemas found. At that point, the AST is considered complete and returned as the result of parsing. The real algorithm, of course, must consider the previously described sectioning and that the content of found range-schema matches must be parsed too.

```
interpret(config, inputText): Sequence<TokenMatch|SchemaMatch> {
    tokenSystem = loadTokenDefinitions(config)
    items = tokenize(inputText, tokenSystem)
    schemaSystem = loadSchemaDefinitions(config)
    parseAreas.add(items, scopes.baseScope)
    do {
       parseAreas.forEachAsync(parseArea -> {
           runHardSectionsOf(parseArea)
           parseAreas.remove(parseArea)
       })
    } while not (parseAreas.isEmpty?)
    return items
}
Where:
- config: the set of input resources, like definition files
- inputText: Sequence<Character>
- tokenSystem: TokenSystem
- items: Sequence<TokenMatch|SchemaMatch>
- schemaSystem: SchemaSystem
- parseAreas: List<Sequence<TokenMatch|SchemaMatch>,Scope>
```

Fig. 2. Pseudo-Code: Interpret

The attached pseudo-codes are still a bit simplified relative to actual code, but they describe how sections and ranges are

included in the parsing process; the tokenization happens only once, yielding all found tokens.

```
tokenize(input, tokenSystem): Sequence<TokenMatch> {
    tokenSystem.hardSections.forEachInOrder(hSection -> {
        do {
            foundNew? = false
            hardSection.softSections.forEachInOrder(sSection -> {
                markers = sSection.definitions.forEach(find)
                foundNew? |= addLeftMostTokenMatchesBasedOn(markers)
            })
        } while (foundNew?)
    })
    return input.filterByType(TokenMatch)
}
Where:
- input: Sequence<Character|TokenMatch>
- foundNew?: bool
Where addLeftMostTokenMatchesBasedOn(markers):
  - replaces left-most matches with token matches
```

Fig. 3. Pseudo-Code: Tokenize

Then, all these tokens are considered one big region of parsing - being now the only 1 parsing region active -, using the default schema definition rule-set from the schema system. The parsing then begins, executing all hard-sections of the rule-set after each other. After completion of the last hard-section, it checks if there has been a new parsing area added. If so, it was added with a rule-set associated with it, time to execute the hard-sections of that rule-set on the given added area.

```
runHardSectionsOf(parseArea): void {
    parseArea.scope.hardSections.forEachInOrder(hardSection -> {
        runHardSection(parseArea, hardSection)
    })
}
Where:
- parseArea: Sequence<TokenMatch|SchemaMatch>,Scope
```

Fig. 4. Pseudo-Code: Run Hard-Sections

This repeats until there are no more parsing areas. Parsing areas are being added during the execution of hard-sections, soft-sections are run and within them, schemas are being found. In case the found schema contains a range schema associating it with a rule-set, it adds the content of the range schema match with the rule-set to the parsing areas waiting for parsing. This also gives place to some serious optimization; the parsing areas are stored in a set essentially, which, if made thread-safe supports parallel parsing on multiple threads.

```
runHardSection(parseArea, hardSection): void {
    do {
        foundNewInCycle? = false
        hardSection.softSections.forEach(softSection -> {
            markers = definitionsInSection.forEach(find)
            foundNewInCycle? |= addLeftMostSchemaMatchesBasedOn(markers)
        })
    } while (foundNewInCycle?)
}
  Where:
- parseArea: Sequence<TokenMatch|SchemaMatch>,Scope
- hardSection: Sequence<SoftSection>
- foundNewInCycle?: bool
  Where addLeftMostSchemaMatchesBasedOn(markers):
  - replaces left-most matches with schema matches
  - possibly adds new items to parseAreas of interpret
```

Fig. 5. Pseudo-Code: Run Hard-Section

IV. Evaluation

A. Usability

The configuration of the described parser is far simpler than mentioned other similar applications and is done from configuration files in a declarative manner, not by code imperatively. This can make it work even as a language-independent command-line tool, not only as a code library. Also, since all required input files are textual in nature, they can enjoy all benefits of common version control systems like Git.

Since the parser is based primarily on the concept of regular expressions, their advantages are gained as well; regular expressions are powerful tool for matching against sequences of items. Also, regex is such a tool most programmers met during their career and can be learned under short periods of time. This parser makes use of the fact, that regular expressions are commonly defined and well known, further lessening the effort needed to use it.

B. Output

Schemas can be described as regular expressions which do backtrack, but which behavior is limited only the matcher of the given schema - the parser algorithm itself does not backtrack, but incrementally builds parts of the resulting AST instead. The already matched parts will not change meaning once matched, so the result can be observed while parsing, even before the parsing ends.

Testing the correctness of the concept and implementation is done via many unit tests and integration tests. Unit tests assure the correct functionality of most individual elements of the software. Integration tests run scenarios, like parsing given input text file to AST - using all elements of the software necessary - then comparing the resulting AST to the expected AST.

C. Performance

Performance was not yet measured or compared as implementation is still in progress. On the other hand, the algorithm opens up significant opportunities of parallelization; the interpretation process yields code blocks as new parsing areas, whose interpretation can be done on a separate thread, and their result injected into the resulting AST later, or progressively in a concurrent way.

Matchers can search for matches concurrently as well, although the implementation effort and overhead of parallelization might out-weight what is gained. If performance is of key importance, then matchers could be reworked and optimized, possibly into a DFA, improving the process significantly. However, optimizing the matching process of multiple regular expressions at once on the same input can be of similar or higher complexity than the algorithm presented in this paper.

Our modern integrated development environments optimize compilations in yet another way; they generally reuse the results of previous compilations. When a programmer does change only a portion of the source code files, then it must be considered not to recompile everything again - only those

parts, which are affected by the applied changes. This optimization is dependent on the way of source code structuring associated with the given parsed programming language, but most languages are modular enough to apply this strategy of compiler output reuse. As the presented algorithm recognizes blocks of code, this strategy can be used on the file content as well; for example, when a function gets changed, then unless the signature is changed, usually there is no need to recompile any of the other surrounding functions.

V. Conclusions

This paper presented an easily usable interpreter based PEG and regular expressions over matched items. Compared to usual parsers, the algorithm itself can be demonstrated and understood easily. Most importantly, it can be learned, set up and used with little effort and can be modified without significant effort.

We also saw, that regular expressions can be used not only on characters of text, but also on a series of arbitrarily defined objects and can be incorporated with custom patterns and matchers of such arbitrary objects (being token and schema matches in the presented parser) and be used for program code parsing.

Future plans include integrating the implementation into Visual Studio Code for code highlight and possibly for compilation using the LLVM pipeline.

Acknowledgment

The authors would like to thank both the GPGPU Programming Research Group of Obuda University and the Hungarian National Talent Program (NTPHHTDK-20) for their valuable support.

References

[1] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 111–122. [Online]. Available: https://doi.org/10.1145/964001.964011

[2] M. Oikawa, R. Ierusalimschy, and A. Moura, "Converting regexes to parsing expression grammars," in *Proceedings of the 14th Brazilian Symposium on Programming Languages, SBLP*, vol. 10, 2010.

[3] S. Medeiros, F. Mascarenhas, and R. Ierusalimschy, "From regular expressions to parsing expression grammars," in *Brazilian Symposium on Programming Languages*, 2011.

[4] Alfred Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley Publishing Company, 2006.

[5] A. Olsson. (2019). [Online]. Available: https://medium.com/storyteltech/parsing-text-input-without-regular-expressions-3e8de68a79a7

[6] C. Diggins. (2018) Beyond regular expressions: An introduction to parsing context-free grammars. [Online]. Available: https://www.freecodecamp.org/news/beyond-regular-expressions-an-introduction-to-parsing-context-free-grammars-ee77bdab5a92/

[7] Parsing regular expressions with recursive descent. [Online]. Available: http://matt.might.net/articles/parsing-regex-with-recursive-descent/

[8] B. Kordic, M. Popovic, and S. Ghilezan, "Formal verification of python software transactional memory based on timed automata," *Acta Polytechnica Hungarica*, vol. 16, no. 7, 2019.

[9] M. Lenz, *Parsing with Perl 6 Regexes and Grammars: A Recursive Descent Into Parsing*. Apress, 2017, page 3.

[10] I. Batyrshin, "Constructing correlation coefficients from similarity and dissimilarity functions," *Acta Polytechnica Hungarica*, vol. 16, pp. 191–204, 04 2019.