

Exploring Regular Expressions Through Practical Examples

Exploring Regular Expressions Through Practical Examples.

By Henrik Vestermark (hve@hvks.com)

Abstract

This paper provides practical examples of regular expressions relevant to creating a lexical checker for a programming language. It assumes that readers have prior knowledge of regular expressions and focus on matching patterns rather than extraction techniques. The examples are drawn from the author's extensive experience developing an interpreter for a modern programming language with features like complex arithmetic and arbitrary precision. The paper uses JavaScript syntax for regular expressions but can be adapted to other languages with minor adjustments.

Introduction

"Some people, when confronted with a programming problem, think, 'I know, I'll use regular expressions to fix it.' Now they have two problems."

This saying humorously highlights the power and potential pitfalls of using regular expressions. Regular expressions, as powerful tools for pattern matching and text processing, can be challenging to write and debug. However, they can empower you to solve complex problems with careful use.

In this paper, I will present practical examples of regular expressions you would encounter if you were making a lexical checker for a programming language. These examples will engage you and provide a hands-on understanding of how regular expressions work in real-world scenarios. This paper is not a tutorial or introduction to regular expressions; the readers are expected to have the knowledge and practical use of regular expressions. Another limitation is that we will only demonstrate how the regular expression example can match or not match a given input. Otherwise, we will not discuss adding or selecting an element from the input example using grouping, non-grouping, or name grouping.

This project was born out of a need to develop an interpreter for a modern language of my own, similar to JavaScript, C, and C++, but with added support for complex arithmetic, Interval arithmetic, and arbitrary precision arithmetic. To achieve this, I built the entire interpreter, including my IDE, in JavaScript and used regular expressions for the lexical scanning of the source file. For that reason, I needed the following detection.

- Integers (decimal, octal, Hexadecimal, or binary)
- Floating-point in the usual notations
- String, using the "" delimiter
- Complex numbers

Exploring Regular Expressions Through Practical Examples

- Interval numbers
- And arbitrary precision integer and float numbers
- All the other terminal symbols like
`<, >, >=, <=, ==, !=, [,], (, ~, !, %, *, +, -, /, :, ;, //, /*, */` etc.

In this paper, I will use the JavaScript version and style of defining a regular expression in all examples. However, if you use a different programming language, rest assured that it will be easy to port these examples to your favorite system with only minor adjustments.

Change Log

11 October 2024. Initial version

Exploring Regular Expressions Through Practical Examples

Contents

Exploring Regular Expressions Through Practical Examples.....	1
Abstract.....	1
Introduction.....	1
Change Log.....	2
A Brief History of Regular Expression.....	4
Application for Regular Expression.....	4
Basics of Regular Expressions.....	5
A Regular expression for integer constants.....	5
A Regular expression for floating-point constants.....	9
A Regular expression for string constants.....	10
Intermediate Regular Expressions.....	11
A Regular expression for interval numbers.....	11
A Regular expression for Complex numbers.....	12
A regular expression for comments.....	13
Advanced Regular Expressions.....	14
Example:.....	14
Overall Result.....	16
Conclusion.....	16
Useful online tools.....	16
Regex101.....	16
RegExr.....	17
RegexBuddy.....	17
Regex Pal.....	17
Regex Tester.....	17
HVE Regular expression Tester.....	18
Reference.....	18

Exploring Regular Expressions Through Practical Examples

A Brief History of Regular Expression

Regular expressions (regex) originated from the mathematical concept of "regular sets" proposed by Stephen Kleene in the 1950s. Kleene, a mathematician, introduced these as part of his research on automata theory, describing a method to define regular languages using what he called "regular events." He introduced the Kleene Star (*), a fundamental component that denotes zero or more repetitions of the previous element.

In the 1960s, Ken Thompson incorporated Kleene's notation into computing through the search utilities of the QED and later Unix text editors. Thompson implemented a method to convert regular expressions into finite automata, enabling efficient text searching and manipulation.

Through the 1970s and 1980s, regular expressions became integral to Unix, significantly enhancing the capabilities of tools like grep (Global Regular Expression Print), sed (stream editor), and awk. These tools popularized regular expressions by providing powerful text-processing capabilities.

Further developments occurred in the 1980s and 1990s, especially with Perl's powerful regex capabilities. Designed by Larry Wall, Perl's regex included advanced features like non-greedy matching and assertions, influencing later standards.

Today, regular expressions are supported in nearly all programming languages and text editors, reflecting their importance in software development. They are especially pivotal in languages like JavaScript, which uses regular expressions extensively for web development tasks such as validating input and searching text.

JavaScript integrates regular expressions in ways essential for web developers, enabling complex text manipulations directly in the browser. Enhancements to JavaScript's regex capabilities have included features like named capture groups and Unicode property escapes, enriching its pattern-matching capabilities.

This historical progression from a theoretical concept to a fundamental programming tool underscores the practical adoption and continuous enhancement of regular expressions across computing disciplines.

Application for Regular Expression

Regular expressions (regex) are powerful in programming, data processing, and text manipulation. Below are typical applications:

- **Data Validation**
Regex validates user input, such as emails and phone numbers, ensuring correct formats. It also cleans data by removing unwanted characters.

Exploring Regular Expressions Through Practical Examples

- Text Processing
They are used in text editors for find-and-replace operations based on patterns, such as extracting dates or names from documents.
- Programming
Regex modifies code across multiple files and is used by editors to format keywords, strings, and comments.
- Log File Analysis
Regex helps find errors and security issues in log files for debugging and monitoring.
- Web Scraping
It extracts specific data from web pages, like prices or contact details.
- Networking
Regex parses network traffic or decomposes URLs for routing or security checks.
- Bioinformatics
Regex identifies genetic patterns in gene sequence analysis.
- Natural Language Processing (NLP)
Regex breaks down text into tokens and identifies linguistic patterns.
- Database Querying
Regex-based querying selects records matching patterns in textual fields.
- Automated Testing
Regex verifies system outputs against expected patterns for testing.

Regular expressions are efficient for manipulating text and data, making them essential across many fields.

Basics of Regular Expressions

As mentioned, the readers should be familiar with and expected to have previous experience using regular expressions.

Let's start with some easy examples and work our way to more advanced examples and the use of regular expressions.

A Regular expression for integer constants.

Scanning for integers is properly the simple way of using regular expressions by simply defining a regular expression like:

```
/[0-9]+/
```

This means a decimal number consists of one or more decimal digits (0 to 9). For example, using the above regular expression, you get this

Input	Match
1	1
12	12

Exploring Regular Expressions Through Practical Examples

123	123
1234	1234
12345	12345
123456	123456

If, for some reason, you only accept a minimum and maximum number of digits, you can use a regular expression like:

```
/[0-9]{3,5}/
```

Indicating that you will only match it if it has at least three digits and only up to the first five digits. Example.

Input	Match
1	
12	
123	123
1234	1234
12345	12345
123456	12345

If you omit the upper bound:

```
/[0-9]{3,}/
```

You match 3 or more digits. The classic:

```
/[0-9]+/
```

It could also have been written as:

```
/[0-9]{1,}/
```

Now, matching integers is so common that regular expressions understand a shortcut for decimal digits 'd'. The two above examples could be written as:

```
/\d+/ and /\d{3,5}/
```

Let's look into how to match the C++ programming language for integers. The BNF syntax could be similar to the syntax below.

```
<integer-constant> ::= <decimal-literal> <integer-suffix-optional>  
                    | <octal-literal> <integer-suffix-optional>  
                    | <hexadecimal-literal> <integer-suffix-optional>  
                    | <binary-literal> <integer-suffix-optional>
```

```
<decimal-literal> ::= <nonzero-digit> <digits-optional>
```

```
                    | <nonzero-digit>
```

```
<octal-literal> ::= '0' <octal-digits-optional>
```

Exploring Regular Expressions Through Practical Examples

```
    | '0'  
<hexadecimal-literal> ::= <hexadecimal-prefix> <hexadecimal-digits>  
<binary-literal> ::= <binary-prefix> <binary-digits>  
  
<digits-optional> ::= <digit> <digits-optional>  
    | ε  
<octal-digits-optional> ::= <octal-digit> <octal-digits-optional>  
    | ε  
<hexadecimal-digits> ::= <hexadecimal-digit> <hexadecimal-digits>  
    | <hexadecimal-digit>  
<binary-digits> ::= <binary-digit> <binary-digits>  
    | <binary-digit>  
  
<digit> ::= '0' | <nonzero-digit>  
<nonzero-digit> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'  
<octal-digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'  
<hexadecimal-digit> ::= <digit> | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' |  
'D' | 'E' | 'F'  
<binary-digit> ::= '0' | '1'  
  
<hexadecimal-prefix> ::= '0x' | '0X'  
<binary-prefix> ::= '0b' | '0B'  
  
<integer-suffix-optional> ::= <integer-suffix>  
    | ε  
<integer-suffix> ::= <unsigned-suffix> <long-suffix>  
    | <long-suffix> <unsigned-suffix>  
    | <unsigned-suffix>  
    | <long-suffix>  
<unsigned-suffix> ::= 'u' | 'U'  
<long-suffix> ::= 'l' | 'L' | 'll' | 'LL'
```

We notice that we need to handle binary, octal, decimal, and hexadecimal versions of an integer plus some suffixes that specify unsigned and the size of the integer type (e.g., 4 bytes, 8 bytes, etc.).

To distinguish between hexadecimal, octal, and binary numbers, they are prefixed with

Prefix	
0x or 0X	Hexadecimal prefix
0	Octal prefix
0b or 0B	Binary prefix

If there is no prefix, it is assumed that it is a decimal number.

0177, 0b10101, 0x12ABC, and 345 are all valid numbers. We have already established the regular expression for decimal digits. For binary numbers, you can use:

```
\0[bB][01]+\
```

For the Octal number, you can use:

```
\0[0-7]*\
```

And for Hexadecimal number:

Exploring Regular Expressions Through Practical Examples

```
\0[xX][0-9a-zA-F]+\
```

Notice that the C++ syntax required at least one digit after the prefix '0x' and '0b'
Now we need to combine them all with the regular expression | operator (or operator) you get:

```
/0[xX][0-9a-zA-F]+|0[0-7]*|0[bB][0-1]+\|d+/\
```

Input	Match
12	12
0x23ab	0x23ab
0b10101	0
0777	0777

Surprisingly, we didn't get a match on the binary integer number. Instead, it returns two other matches: 0 as an octal number match. This is a classic case of the regular expression pitfall.

Let's break down the provided pattern and its behavior with the input '0b10101'.

First Alternative: 0[xX][0-9a-zA-Z]+ does not match since x is missing in the input

Second Alternative: 0[0-7]*:

- Matches a '0' followed by zero or more octal digits (0-7).
- The input '0b10101' matches the '0' at the beginning since '[0-7]*' can match zero occurrences of '[0-7]'.
- These leaves 'b10101' as the remaining unmatched string.

Third Alternative: 0[bB][0-1]+:

- Matches a '0' followed by 'b' or 'B' and one or more binary digits (0 or 1).
- The remaining input, 'b10101', does not match since the leading character is 'b' and is not part of a valid binary sequence starting with '0'.

Fourth Alternative [0-9]+:

- Matches one or more decimal digits (0-9).
- For the remaining input 'b10101', the initial 'b' also disqualifies this part.

The Regex engine starts with the first alternative, finds a match for '0' on the second alternative, and does not proceed to try the other options comprehensively.

To achieve a match that correctly handles the binary number, we need to adjust the Regex or apply additional logic to prioritize the binary pattern correctly.

To get it right, we need to ensure that we try the binary number before trying the octal pattern. The revised regular expression becomes:

```
/0[xX][0-9a-zA-F]+ |0[bB][0-1]+ |0[0-7]*|\d+/\
```

Input	Match
-------	-------

Exploring Regular Expressions Through Practical Examples

12	12
0x23ab	0x23ab
0b10101	0b10101
0777	0777

Now, we correctly match all the different variables in the base of an integer constant. If we try it with an input of 0, we notice that it will match up with the octal patterns. However, that is acceptable since a single 0 in base 8 and 10 is the same.

Lastly, we need to add the optional suffix. Since it all is optional, we can add this to the end of our current pattern, and we get this:

```
/(\0[xX][\0-9a-fA-F]+ |\0[bB][\0-1]+ |\0[\0-7]*|\d+)([uU]?[lL]{0,2}|[lL]{0,2}[uU]?)/
```

Notice that u, l, and ll can appear in any order. We can now match any C++ syntax for integer constants. To be complete, we could also add an optional sign as the prefix to the integer constants, and we get our final regular expression:

```
/[+-]?(\0[xX][\0-9a-fA-F]+ |\0[bB][\0-1]+ |\0[\0-7]*|\d+)([uU]?[lL]{0,2}|[lL]{0,2}[uU]?)/
```

Even for such a simple pattern matching of integer constant, we still end up with the above intimidating regular expression.

A Regular expression for floating-point constants.

Unfortunately, there is no shortcut for floating-point numbers, as we saw with the `\d` for integers only. We would have to build up the regular expression for floating-point numbers from scratch.

The BNF syntax for floating-point constants is:

```
<floating-point> ::= <mantissa> <exponent-part-optional> <floating-point-suffix-optional>
<mantissa> ::= <digits> '.' <digits>
              | <digits> '.'
              | '.' <digits>
              | <digits>
<floating-point-suffix-optional> ::= 'f' | 'F' | 'l' | 'L' | ε
<digits> ::= <digit> <digits> | <digit>
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<exponent-part-optional> ::= <exponent-part> | ε
<exponent-part> ::= ('E' | 'e') <optional-sign> <digits>
<optional-sign> ::= '+' | '-' | ε
```

Exploring Regular Expressions Through Practical Examples

We can easily create the corresponding regular expression using the above grammatical rules for floating-point numbers.

Mantissa grammar

Floating-point grammar	Regular expression	Optimized
<digits> '.' <digits>	\d+\.\d+	\d+\.\d*
<digits> '.'	\d+\.	
'.' <digits>	\.\d+	
<digits>	\d+	

\d+\.\d+ and \d+\. => \d+\.\d*

And \d+\.\d* can be combined with \d+ => \d+(\.\d+)?

The Exponential part is straightforward;

```
[Ee][+-]?\d+
```

And combining it all, you get the regular expression for floating-point:

```
(\d+('.\d*)?|\.\d+)([Ee][+-]?\d+)?
```

Now, the C++ standard specifies that the valid suffix to a float is f or F for a 32-bit float and l and L for a long double. Add this suffix together with an optional sign of + or -, and you get the final version:

```
[+-]?(\d+('.\d*)?|\.\d+)([Ee][+-]?\d+)?[fFLL]?
```

Input	Match
123.	123.
123.4	123.4
.12	.12
-123f	-123f
1E4	1E4
.1e4L	.1e4L
2.4e-5	2.4e-5

A Regular expression for string constants.

Below is a string constant that also can handle escape quote \" in the string:

```
"\"(\\. | [^\"\\])*\\"
```

At first glance, it looks odd, but breaking it down makes it easier to digest.

\": Matches the opening quote.

Exploring Regular Expressions Through Practical Examples

`(\\.[^"\\\])*`: Matches any sequence of:

`\\`: Any escaped character, including an escaped quote `"`. The backslash must be escaped in the Regex pattern, hence `\\`.

`[^"\\\]`: Any character without a quote or a backslash.

`"`: Matches the closing quote.

This pattern accounts for any character that is either escaped (anything immediately following a backslash, including a double quote) or not a backslash or a quote. Using `[^"\\\]` ensures the Regex does not prematurely end when hitting an escaped quote.

String text	Matched Text
"text"	"text"
"more text \" and even more"	"more text \" and even more"
" text \" \" "	" text \" \" "
" even more text \"	" even more text \"

Intermediate Regular Expressions

We now turn to more advanced regular expressions. In the previous section, we showed how to deal with floating-point numbers; however, since C14, you can now add a separator: ‘to read the number easily.

The floating point for regular expression is:

```
[+-]?(\d+(\.'\d*)?|\.\d+)([Ee][+-]?\d+)?[fFLL]?
```

The only rule is that a digit needs to be in front of the separator ‘. This can be accomplished by adding `(\d+)*` after every `d+` and altering the `\d*` to `(\d+)*`. For handling the separator.

```
[+-]?(\d+(\.'\d+)*(\.\d+(\.'\d+)*)*?)|\.\d+(\.'\d+)*)([eE][+-]?\d+)?[fFLL]?
```

It is much harder to read, but it will do the job. A more straightforward way to handle separators is to remove them before matching up against the floating-point regular expressions. This is OK since a separator does not alter the number at hand, and at some point, you need to convert it from the decimal representation to the binary IEEE754 format for floating-point.

A Regular expression for interval numbers.

Another helpful way of using regular expressions is to handle intervals in interval arithmetic. An interval consists of one or two floating-point numbers separated by a comma and enclosed with a square or round bracket indicating whether it has open or closed left and right values.

Exploring Regular Expressions Through Practical Examples

```
<optional-sign> ::= '+'  
                  | '-'  
                  | <empty>
```

We assume that <float> is the BNF for a floating-point number.

A regular expression that matches this is:

```
\((((([+-]?\d+(\.\d*)?)|\.\d+)([eE][+-]?\d+)?)?([+-]?\d+(\.\d*)?)|\.\d+)([eE][+-]?\d+)?[iI])?|[+-]?[iI])\)
```

As for the Interval number, the regular expression is long and very hard to read and maintain.

Input text	Matched text
(1)	(1)
(-2+2i)	(-2+2i)
(3i)	(3i)
(-4i)	(-4i)
(.12-.6i)	(.12-.6i)
(1i)	(1i)
(+2i)	(+2i)
(i)	(i)
()	()
(-i)	(-i)

A regular expression for comments

Before moving on to advanced regular expressions, we need to address the issue of matching comments in modern languages like JavaScript, C++, and others.

For this language, a comment can be a single-line comment starting with the double slash symbol ‘//’ or a multiline comment enclosed in the symbol of ‘/*’ and ‘*/.’

Since comments can span multiple input text, they require a particular regular expression.

The regular expression is:

```
(\\/*([^\n]|[\r\n])*(/*\n|[\r\n]))*\n| (\\/\/\.*)
```

It's not as complex as complex numbers or intervals, but it's also hard to read.

Input text	Matched text
// text comment	// text comment
/* Multi Line	/* Multi Line

Exploring Regular Expressions Through Practical Examples

Comment */	Comment */
---------------	---------------

Advanced Regular Expressions

As mentioned, I was developing a language, a mix of JavaScript and C++ (as an interpreter), with build types like complex numbers, interval numbers, and support of arbitrary precision. I'd like to know how efficiently I can make the lexical scanning using as much regular expression as possible. To provide sufficient error code in case of lexical or syntax errors, we must skip any comments but preserve the line numbers referring back to the source code. In other words, I need to replace multiline comments with empty newlines to avoid messing up the source line code. Furthermore, I needed to eliminate the single-line comments and replace them with a new line. The easiest way was to pass the source in two steps. Step one was to remove all comments described above, and step two was to use a regular expression to split up all the terminal symbols into tokens for subsequent grammar parsing. I ended up with a lengthy regular expression that would split the input source into tokens that could be passed on to the parser for syntax, semantics, and code generation.

```
/[+][+=]?|[-][-  
=]?|[*]{1,2}[=]?|[/]{1,2}[=]?|&[&=]?|[|]{1,3}[=]?|>{1,3}[=]?|<<{1,3}[=]?|[%^!~=  
][=]?|[() , ; : {} \ ] # @ | [0][xX][\da-fA-F_]+[sun]?(i)?|[0][oO][0-  
7_]+[sun]?(i)?|[0][bB][0-  
1_]+[sun]?(i)?|((([ \d_ ]+([ \. ] [ \d_ ]*)?))|([ \. ] [ \d_ ]+))([eE][+-]?[ \d ]+)?([su  
fn])?(i)?|[0-9_]+[sun]?(i)?|[_a-zA-  
Z][\w]*|\"(?:\\\"|.)*?(\"|$)|\'(?:\\\'|.)*?(\'|$)|[ \. ]([+%\-\-]|[*]{1,2}|[/]{  
1,2})|[.]{1,3}/
```

This is most likely an overuse of regular expression and is definitively hard to maintain.

Example:

A small C++ function calculates the error upper bound when evaluating a polynomial.

```
// Calculate an upper bound for the rounding errors performed in a  
// polynomial with complex coefficient a[] at a complex point z.  
// (Grant & Hitchins test)  
auto upperbound = [](const vector<complex<double>>& a, complex<double> z)  
{  
    const size_t n = a.size() - 1;  
    double nc, oc, nd, od, ng, og, nh, oh, t, u, v, w, e;  
    double tol = 0.5 * pow((double)_DBL_RADIX, -DBL_MANT_DIG + 1);  
  
    oc = a[0].real();  
    od = a[0].imag();  
    og = oh = 1.0;  
    t = fabs(z.real());  
    u = fabs(z.imag());  
    for (size_t i = 1; i <= n; i++)
```

Exploring Regular Expressions Through Practical Examples

```

{
    nc = z.real() * oc - z.imag() * od + a[i].real();
    nd = z.imag() * oc + z.real() * od + a[i].imag();
    v = og + fabs(oc);
    w = oh + fabs(od);
    ng = t * v + u * w + fabs(a[i].real()) + 2.0 * fabs(nc);
    nh = u * v + t * w + fabs(a[i].imag()) + 2.0 * fabs(nd);
    og = ng;
    oh = nh;
    oc = nc;
    od = nd;
}
e = abs(complex<double>(ng, nh)) * pow(1 + tol, 5 * n) * tol;
return e;
};

```

The corresponding terminal tokens are enclosed by || and listed per source line. Notice that the comments have been removed, but the lines have been preserved. \$ indicates the end of a line. I added no reserved words in the advanced Regex, which would have bloated the Regex. Instead, I only picked up identifiers and then added the reserved words to the name table, avoiding accidentally using a reserved word in a declaration.

```

Tokens: 490 tokens generated.
[ 1]$
[ 2]  $
[ 3]  $
[ 4]  | auto | | upperbound | | = | | [ | ] | ( | const | | vector | < |
complex | < | double | >> | & | | a | , | | complex | < | double | > | | z | )$
[ 5]  | {$
[ 6]  | const | | size_t | | n | | = | | a | . | size | ( | ) | | - | |
1 | ;$
[ 7]  | double | | nc | , | | oc | , | | nd | , | | od | , | | ng | , |
| og | , | | nh | , | | oh | , | | t | , | | u | , | | v | , | | w | , | | e
| ;$
[ 8]  | double | | tol | | = | | 0.5 | | * | | pow | ( | ( | double | ) |
- | DBL_RADIX | , | | - | DBL_MANT_DIG | | + | | 1 | ) | ;$
[ 9]  $
[ 10] | oc | | = | | a | [ | 0 | ] | . | real | ( | ) | ;$
[ 11] | od | | = | | a | [ | 0 | ] | . | imag | ( | ) | ;$
[ 12] | og | | = | | oh | | = | | 1.0 | ;$
[ 13] | t | | = | | fabs | ( | z | . | real | ( | ) | ) | ; | $
[ 14] | u | | = | | fabs | ( | z | . | imag | ( | ) | ) | ;$
[ 15] | for | | ( | size_t | | i | | = | | 1 | ; | | i | | <= | | n
| ; | | i | ++ | )$
[ 16] | {$
[ 17] | nc | | = | | z | . | real | ( | ) | | * | | oc | | - | | z
| . | imag | ( | ) | | * | | od | | + | | a | [ | i | ] | . | real | ( | ) | ;$
[ 18] | nd | | = | | z | . | imag | ( | ) | | * | | oc | | + | | z
| . | real | ( | ) | | * | | od | | + | | a | [ | i | ] | . | imag | ( | ) | ;$
[ 19] | v | | = | | og | | + | | fabs | ( | oc | ) | ; | $
[ 20] | w | | = | | oh | | + | | fabs | ( | od | ) | ;$
[ 21] | ng | | = | | t | * | | v | | + | | u | * | | w |
| + | | fabs | ( | a | [ | i | ] | . | real | ( | ) | ) | | + | | 2.0 | | * | |
fabs | ( | nc | ) | ;$
[ 22] | nh | | = | | u | * | | v | | + | | t | * | | w |
| + | | fabs | ( | a | [ | i | ] | . | imag | ( | ) | ) | | + | | 2.0 | | * | |
fabs | ( | nd | ) | ;$
[ 23] | og | | = | | ng | ; | $
[ 24] | oh | | = | | nh | ;$
[ 25] | oc | | = | | nc | ; | $
[ 26] | od | | = | | nd | ;$
[ 27] | }$
[ 28] | e | | = | | abs | ( | complex | < | double | > | ( | ng | , | | nh
| ) | ) | | * | | pow | ( | 1 | | + | | tol | , | | 5 | | * | | n | ) | | *
| | tol | ;$

```

Exploring Regular Expressions Through Practical Examples

```
[ 29]      | return | | e | ;$  
[ 30]      | } | ;$  
[ 31]$  
[ 32]$
```

As can be seen, lexical scanning can tokenize an input stream for any programming language with no or few adjustments to the regular expression.

Overall Result

After all the regular expressions had been built and tested, it was surprisingly easy to create a function to scan the entire source of my programming language lexically and parse the tokens one at a time to the compiler for the grammar.

Conclusion

This paper has demonstrated the practical application of regular expressions in constructing a lexical checker for a programming language, showcasing real-world scenarios where complex patterns are essential. This paper has explored regular expressions for common elements such as integers, floating-point numbers, strings, intervals, complex numbers, and comments. While regular expressions are a powerful tool, their complexity can often make them difficult to manage, especially when handling multiple data types and edge cases. However, with careful design and optimization, they remain an indispensable asset for parsing and analyzing source code in language interpreters.

Useful online tools

Several online tools are available for testing and debugging regular expressions. Here are some widely used Regex tools with relevant details.

Regex101

- **Website:** [Regex101](#)
- **Description:** Regex101 is a powerful online tool for testing and debugging regular expressions. It supports multiple programming languages, including JavaScript, Python, PHP, and Go. The tool provides a detailed explanation of each Regex part as you type, along with a quick reference guide and a library of user-submitted Regex patterns.
- **Key Features:**
 - Real-time Regex parsing and testing.
 - Detailed explanations of Regex constructions.
 - Code generator for various languages.
 - User pattern library.

Exploring Regular Expressions Through Practical Examples

RegExr

- **Website:** [RegExr](#)
- **Description:** RegExr is another popular online tool for learning, building, and testing regular expressions. It offers a clean interface and provides real-time visual feedback on your Regex pattern matching.
- **Key Features:**
 - Real-time results and highlighting.
 - Extensive community patterns and examples.
 - Detailed help and cheat sheets.
 - Provide a history of your Regex tests for easy backtracking.

RegexBuddy

- **Website:** [RegexBuddy](#)
- **Description:** RegexBuddy is a downloadable tool for Windows that acts as your Regex assistant. It helps you create and understand complex Regexes and implements them in source code.
- **Key Features:**
 - Detailed analysis of regular expressions.
 - Test Regexes against sample texts.
 - Integration with various programming environments.
 - Regex building blocks for easier assembly.

Regex Pal

- **Website:** [Regex Pal](#)
- **Description:** Regex Pal is a straightforward, web-based tool for quickly testing JavaScript regular expressions. It provides immediate visual feedback but is more straightforward and less feature-rich than Regex101 or RegExr.
- **Key Features:**
 - Quick testing with real-time highlighting.
 - Minimalistic and fast.
 - Sidebar with Regex tokens and short descriptions.

Regex Tester

- **Website:** [Regex Tester and Debugger Online - Javascript, PCRE, PHP](#)
- **Description:** Regex Tester from RegexPlanet supports testing and debugging regex for multiple programming languages, including Java, .NET, and Ruby. It offers a unique environment to test regex in different programming contexts.
- **Key Features:**
 - Support for several programming languages.
 - Regex library and community examples.
 - Advanced options for regex testing and results.

Exploring Regular Expressions Through Practical Examples

HVE Regular expression Tester

- **Website:** [Testing Regular expression for matching specific text patterns \(hvks.com\)](https://hvks.com)
- **Description:** Regular Expression Tester supports testing and debugging regex for JavaScript programming languages. It offers a unique environment to test regex in different programming contexts.
- **Key Features:**
 - Support for most programming languages.
 - Samples of regular expression for most common day coding problems
 - Offer decomposing of regular expressions for easier debugging and understanding.
 - Offer optimization of regular expressions
 - Offer multiline matching.
 - Can check and test two regular expressions simultaneously.
 - Offers printing and emailing of results.

Reference

1. J. Goyvaerts & S. Levithan, Regular Expression Cookbook, O'Reilly May 2009
2. Regular Expression Tester. [Testing Regular expression for matching specific text patterns \(hvks.com\)](https://hvks.com)