Introduction to Regular Expressions

What a Regular Expression Is

A regular expression (regex) is a pattern that describes a set of strings.

- Defines what local patterns match
- Does not understand global structure
- Answers: "Does this substring match this pattern?"

Examples:

- Email addresses
- Phone numbers
- Whitespace
- Numeric literals
- Operator symbols

Regex vs Grammar

Both describe languages, but at different scales:

Regex:

- Matches local patterns (single token)
- Cannot enforce balanced structure like `(((...)))`
- Useful for tokenization

Grammar (EBNF):

- Describes global structure
- Can enforce balanced parentheses
- Operates on tokens

In the pipeline: **Regex → Tokens → Grammar → AST**

Basic Literal Matching

The simplest regex is a literal string.

Pattern: `hello`

Matches: `hello`

Does not match: `Hello`, `hell`, `hello world` (without anchors)

Character Classes

`[...]` matches any single character inside the brackets.

```
[abc]       matches 'a', 'b', or 'c'
[a-z]       matches any lowercase letter
[0-9]       matches any digit
[A-Za-z0-9] matches any letter or digit
```

Negation:

```
[^abc]      matches any character except 'a', 'b', 'c'
[^0-9]      matches any non-digit
```

Common Escape Sequences

```
`.`  – any character (except newline by default)
`\d` – digit `[0-9]`
`\w` – word character `[a-zA-Z0-9_]`
`\b` – word boundary (to make sure you only match a word)
`\s` – whitespace (space, tab, newline, etc.)
`\n` – newline
`\t` – tab
```

Negation:

```
`\D` – non-digit
`\W` – non-word character
`\S` – non-whitespace
```

Quantifiers

Quantifiers specify how many times a pattern repeats.

```
x?   – zero or one (optional)
x*   – zero or more
x+   – one or more
x{n} – exactly n times
x{n,m} – between n and m times
```

Examples:

```
\d+       — one or more digits (matches: 42, 3, 1000)
\d*       — zero or more digits (matches: '', 0, 123)
[a-z]{2,4} — 2 to 4 lowercase letters
colou?r   — 'color' or 'colour'
```

Anchors

Anchors specify where in the string a match must occur.

```
^   — start of string
$   — end of string
```

Examples:

Pattern: `^hello$`

Matches: `hello`

Does not match: `hello`, `hello `, `hello world`

Pattern: `^[0-9]+$`

Matches: `42`, `0`, `123456789`

Does not match: `42x`, `x42`, `42`

Grouping and Alternation

`(...)` groups patterns.

`|` matches any alternative.

```
(cat|dog)    — matches 'cat' or 'dog'
(ab)+        — one or more 'ab' (matches: ab, abab, ababab)
([0-9]{1,3}\.){3}[0-9]{1,3}  — pattern for IP address (simplified)
```

Regex in Tokenization

A tokenizer uses regexes to find tokens in source code.

Example rules:

```
patterns = [
    (r"\s+", "whitespace"),
    (r"\d+", "number"),
    (r"\+", "+"),
    (r"\-", "-"),
    (r"\/", "/"),
    (r"\*", "*"),
    (r"\(", "("),
    (r"\)", ")"),
    (r".", "error"),
]
```

Scanner algorithm:

1. Start at position 0
2. Try each rule in order
3. The first match wins
4. Emit a token and advance
5. Repeat until end of input

Greedy vs Non-Greedy

By default, quantifiers are **greedy** (match as much as possible).

Pattern: `a.*b`

String: `axxxbyyybzzb`

Match: `axxxbyyybzzb` (greedy — matches to the last `b`)

Non-greedy syntax (in some languages):

Pattern: `a.*?b`

Match: `axxxb` (non-greedy — matches to the first `b`)

Common Pitfalls

Regex order matters:

Rules:

```
    (r"if", "if"),
    (r"[a-zA-Z_][A-Za-z0-9_]*", "identifier"),
```

Input: `if`

Result: Matches `"if"` (first rule), correctly identified as keyword.

But if order were reversed:

Input: `if`

Result: Matches as an `identifier` and is tagged as an identifier.

Solution: Order rules from most specific to most general.

Limitations

Regex cannot match:

- Balanced elements: `((...))`
- Nested structures
- Recursive patterns

These require grammar (EBNF and a parser).

Regex answers: "What is this token?"

Grammar answers: "Is this program valid?"

Why Regex Matters

Regex is the frontend of language processing:

- Quick pattern matching
- Easy to specify
- Efficient to implement
- Integrates with almost all languages

Combined with grammar and evaluation:

- Tokenizer (regex) → Parser (grammar) → Evaluator (logic)

Understanding regex is essential for building language tools.