React Component Child Selector Implementation Utilizing Regular Expression

Nadhif Radityo Nugroho (13523045)[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523045@mahasiswa.itb.ac.id, [2]nadhifradityo@gmail.com

***Abstract*—React has been the most popular JavaScript library used for building user interfaces, when fast and dynamic updates to the user interface without reloading the entire page are needed. The user interface is built using components, which are reusable pieces of code that return React elements. Along with declarative syntax, it describes what the user interface should look like, and React takes care of updating the DOM when the data change. And finally, it has an extension called JSX that lets us write HTML-like code in JavaScript. The problem arises when a component needs business logic, such as type checking, interface, or other custom logic, run against the children on which it is passed on. Though React has built-in children iterator and assertion, it still suffers from lack of features and requires developer intervention to acquire such simple logic as DOM query selector. This paper provides alternatives such as selecting react child nodes, property checking, grouping, and wildcard using regular expressions. Not only is this alternative elegant, but also runs on JavaScript's optimized regular expression machine, bypassing JavaScript code iterator and recursor.**

***Index Terms*—Regular expression, String matching, React**

I. Introduction

React has been a major framework used in large applications. Its declarative syntax, along with the JSX extension, allows developers to write self-explanatory code that can be easily understood. Pairing declarative syntax with procedural code also allows developers to express logic with greater ease than ever before. It also enables developers to create complex and interactive user interfaces by composing reusable components. Each component is a self-contained unit that defines how a portion of the user interface should appear based on a given state or data model. React's virtual DOM and diffing algorithm ensure minimal and optimized updates to the real DOM, improving performance and responsiveness across applications.

Each component can accept other nested components that allows developers to utilize the composition pattern. The component may also inspect and modify these children by using `React.Children.count`, `React.Children.forEach`, `React.Children.map`, `React.Children.only`, `React.Children.toArray`, and `React.isValidElement` utilities that React provides. Though these utilities allow for basic iteration and validation, these are often not enough for developers' more advanced use cases. Developers often need to resort to implementing custom logic for their own advanced use cases. These attempts not only result in more boilerplate code, but also compromise the nature of declarative syntax that React promotes. Furthermore, it becomes harder to maintain structural or behavioral constraints as these attempts often hard-coded for specific components requirements.

To make it clearer, this paper will explain real-world use cases that developers often implement for these problems and the alternatives provided in this paper. In short, these include defining the pattern with declarative syntax, type assertion against properties values and children primitives, filtering and grouping hierarchical children, and finally traversing nested structures. All in all, not only this solution is elegant, but also performant as it uses JavaScript's built-in regular expression machine implemented natively in a single pass. So, no JavaScript code will be the main overhead. Additionally, this paper also provides an outline on how to utilize the results of regular expression to select and extract specific children.

II. Theoretical Foundations

A. React Component Architecture

React is a library for expressing the HTML structure in an expressive way that aims to make building user interfaces easier. It allows developers to composite multiple components, allowing complex applications to be built with ease. It is maintained by Meta and a community of individual developers and companies. Many popular frameworks are built on top of React, such as Next.js and Remix. React also supports single-page, mobile, and server-rendered applications. A key advantage of React is that it only re-renders those parts of the page that have changed, avoiding unnecessary re-rendering of unchanged DOM elements. This works because React has an internal representation of DOM, called Virtual DOM nodes. Each state changes, and React will compute the before-and-after states against the diffing algorithm and render the changes accordingly.

In this paper, we will discuss one specific problem that is matching hierarchical virtual DOM nodes specifically when passing children to a component. Before that, here are the built-in React tools to inspect virtual DOM nodes:

- `React.Children.count`: Count the number of children in the 'children' data structure.

- `React.Children.forEach`: Run some code for each child in the 'children' data structure.
- `React.Children.map`: Map or transform each child in the 'children' data structure.
- `React.Children.only`: Assert that the children represent a single React element.
- `React.Children.toArray`: Create an array from the 'children' data structure.
- `React.isValidElement`: Checks whether a value is a React element.

B. Type Checking and Pattern Matching in UI Logic

While component system allows developers to create a complex application, developers often find themselves needing to check against the type of the children to give custom behaviors. Instead of statically checking every single invocation to a component matches with the expected types, this solution allows to dynamically check the type at runtime, avoiding cost of development and improving developer experience. Another big problem is structural validation, which cannot be easily solved by using conventional procedural code, as it only works for a specific case and not a generic one. This paper will provide solutions to these problems and additionally other features borrowed from regular expression.

C. Regular Expressions as a Pattern Matching Tool

Regular expression is a way to match string against a specific pattern. It is also sometimes referred to as a rational expression. Regular expression techniques are developed in theoretical computer science and formal language theory. The concept of regular expressions began in the 1950s, when the American mathematician Stephen Cole Kleene formalized the concept of a regular language.

III. Methodology

This paper takes advantage of practical methods as its methodology. This approach was chosen because it fit the need to prove the implementation of regular expression to match React child nodes. With that said, here are the main outlines of this paper.

- Patterns Definition and Parser: Expresses the pattern declaratively and compiles the pattern into regular expression.
- React Nodes Representation: Converts React's nodes to a text that regular expression can match against.
- Regular Expression Result Extraction: Maps the resulting text back to the React nodes.
- React Nodes Modification: An outline on how to use the matched React nodes and use it to modify the properties or its children.

IV. Implementation

Implementation of this paper basically consists of 4 steps. Starting with defining pattern and parser, representing React nodes into a text, regular expression result extraction, and finally an outline on how to modify the matched React nodes. Each step has their own unique problem that fits with a specific domain.

A. Patterns Definition and Parser

This step basically constructs a React pattern and then parses it into a regular expression pattern. Recognition to different data types that React nodes support is needed. This includes numbers, strings, arrays, and React nodes. Implementation of anchor, group, wildcard, count, look, and back reference is also important to match with the regular expression token specification.

Code 1 Pattern Parser Entry

```
protected static FlagToken: FlagToken;
protected static AnchorToken: AnchorToken;
protected static GroupToken: GroupToken;
protected static AnyToken: AnyToken;
protected static AlternateToken: AlternateToken;
protected static CountToken: CountToken;
protected static LookToken: LookToken;
protected static BackReferenceToken:
↪  BackReferenceToken;

// runtime state
protected knownReactTypes = new
↪  Map<string|React.JSXElementConstructor<any>,
↪  number>();
protected knownReactPropKeys = new Map<string,
↪  number>();
protected knownReactPropValues = new Map<any,
↪  number>();
protected groupIds: (BracketGroup|UserGroup)[] = [];

protected compiledPattern: string;
protected compiledRegex: RegExp;
public lastIndex = 0;

constructor(
  pattern: (tokens: {
    Flag: FlagToken;
    Anchor: AnchorToken;
    Group: GroupToken;
    Any: AnyToken;
    Alternate: AlternateToken;
    Count: CountToken;
    Look: LookToken;
    BackReference: BackReferenceToken;
  }) => React.ReactNode
) {
  // build raw JSX tree
  const rawTokens = pattern({
    Flag: ReactRegExp.FlagToken,
    Anchor: ReactRegExp.AnchorToken,
    Group: ReactRegExp.GroupToken,
    Any: ReactRegExp.AnyToken,
    Alternate: ReactRegExp.AlternateToken,
    Count: ReactRegExp.CountToken,
    Look: ReactRegExp.LookToken,
    BackReference: ReactRegExp.BackReferenceToken
  });

  // default flags
  const flags: Required<FlagProps> = {
    propsMatch: "includes",
    primitiveMatch: "sametype"
  };

  // helper to generate group IDs
  let userGroupCounter = 0;
  const backRefQueue: (string|number)[] = [];
```

```
const newGroupId = (type: "bracket"|"user",
↪  routine: React.ReactNode = null, name?:
↪  string) => {
  const id = `_${this.groupIds.length + 1}`;
  if (type === "bracket") {
    this.groupIds.push({ type: "bracket", id });
  } else {
    this.groupIds.push({ type: "user", id,
    ↪  routine, uid: userGroupCounter++, name });
  }
  return id;
};

// the recursive parser uses these methods:
const parseToken = (node: React.ReactNode):
↪  string => {
  if (isPrimitive(node)) return
  ↪  this.parsePrimitive(node, flags,
  ↪  newGroupId);
  if (isIterable(node)) return
  ↪  this.parseIterable(node, parseToken);
  if (ReactIs.isFragment(node) ||
  ↪  ReactIs.isPortal(node)) {
    return this.parseFragmentOrPortal(node,
    ↪  parseToken);
  }
  if (ReactIs.isElement(node)) {
    const type = node.type;
    switch (type) {
      case ReactRegExp.FlagToken:
        return this.parseFlag(node.props as
        ↪  FlagProps, parseToken, flags);
      case ReactRegExp.AnchorToken:
        return this.parseAnchor((node.props as
        ↪  AnchorProps).kind);
      case ReactRegExp.GroupToken:
        return this.parseGroup(node.props as
        ↪  GroupProps, parseToken, newGroupId);
      case ReactRegExp.AlternateToken:
        return this.parseAlternate();
      case ReactRegExp.CountToken:
        return this.parseCount(node.props as
        ↪  CountProps);
      case ReactRegExp.LookToken:
        return this.parseLook(node.props as
        ↪  LookProps, parseToken);
      case ReactRegExp.BackReferenceToken:
        return
        ↪  this.parseBackReference((node.props
        ↪  as BackReferenceProps).reference,
        ↪  backRefQueue);
      default:
        return this.parseElement(node,
        ↪  parseToken, flags, newGroupId);
    }
  }
  return "";
};

// wrap in a top-level capture group
let fullPattern = (() => {
  const rootId = newGroupId("user", rawTokens);
  return `(?<${rootId}>${parseToken(rawTokens)})`;
})();

// resolve back-refs
fullPattern = regexReplace(/<BR-(\d+)>/g,
↪  fullPattern, (m) => {
  const idx = +m[1], ref = backRefQueue[idx];
  const grp = this.groupIds.find(g => g.type ===
  ↪  "user" && ((typeof ref==="number"?g.uid:
  ↪  g.name) === ref)) as UserGroup;
  if (!grp) throw new Error(`Back reference
  ↪  '${ref}' not found`);
  if (grp.uid === 0) throw new Error(`Cannot
  ↪  back-reference root group`);
  const subPat = parseToken(grp.routine);
  // detect recursive
  if (/<BR-\d+>/.test(subPat))
    throw new Error(`Recursive back-reference in
    ↪  group '${ref}'`);
  return subPat;
});

this.compiledPattern = fullPattern;
this.compiledRegex = new RegExp(fullPattern, "g");
}
```

The pattern construction phase is initiated by interpreting a JSX-based description, composed of specialized components such as `<Group>`, `<Count>`, and `<Look>`, as an abstract syntax tree. `parseToken` then examines each node and delegates to dedicated handlers that emit corresponding regular-expression fragments. Primitive values are translated into bracketed tokens containing type and value identifiers; iterable collections are rendered as concatenated subpatterns; fragments and portals are unwrapped and their contents inlined; and flag components temporarily adjust matching parameters during recursive parsing. Anchor, alternation, quantifier, and lookaround components produce the familiar regex operators (`^`, `$`, `|`, `*`, `+`, `?`, `{m,n}`, `(?=...)`, `(?!...)`, `(?<=...)`, `(?<!...)`), while back-reference placeholders (`<BR-n>`) are resolved in a postprocessing step to ensure proper subgroup linkage. Standard React elements are converted into non-capturing groups of the form

```
/(?:\[\<typeId\>,(?<gid>\d+),\<propsRegex
\>\]\<childrenPattern\>\[\/\k<gid>\])/
```

thereby encoding element type, property constraints, and nested structure into a single composite pattern.

1) Token Generators: This code creates singletons for token elements that will be used within the React pattern construction.

Code 2 React Elements for Token Generators

```
const tokenGenerator = <P>(name: string, tag:
↪  string) => {
    const token = <P>(props: P) =>
        React.createElement(tag, props as any,
        ↪  (props as P & { children?:
        ↪  React.ReactNode }).children);
    token.displayName = name;
    return token;
};
ReactRegExp.FlagToken =
↪  tokenGenerator<FlagProps>("FlagComponent",
↪  "flag");
ReactRegExp.AnchorToken =
↪  tokenGenerator<AnchorProps>("AnchorComponent",
↪  "anchor");
ReactRegExp.GroupToken =
↪  tokenGenerator<GroupProps>("GroupComponent",
↪  "group");
ReactRegExp.AnyToken =
↪  tokenGenerator<AnyProps>("AnyComponent", "any");
ReactRegExp.AlternateToken = tokenGenerator<Alterna
↪  teProps>("AlternateComponent", "alternate");
```

```
ReactRegExp.CountToken =
↪  tokenGenerator<CountProps>("CountComponent",
↪  "count");
ReactRegExp.LookToken =
↪  tokenGenerator<LookProps>("LookComponent",
↪  "look");
ReactRegExp.BackReferenceToken = tokenGenerator<Bac
↪  kReferenceProps>("BackReferenceComponent",
↪  "backreference");
```

2) Primitive Parser: Handles primitive values (`string`, `number`, `boolean`, `null`, `undefined`) and returns a regex fragment based on the `primitiveMatch` flag.

Code 3 Primitive Parser

```
function isPrimitive(n: any): n is
↪  string|number|boolean|null|undefined {
  const t = typeof n;
  return n == null || t==="string"||t==="number"||t
  ↪  ==="boolean"||t==="undefined";
}
private parsePrimitive(
  node: React.ReactNode,
  flags: Required<FlagProps>,
  newGroupId: (t: any, r?: any, n?: any) => string
): string {
  const type = typeof node;
  const symbol = `PRIMITIVE_${node == null ? "null"
  ↪  : type}`;
  let typeId = this.knownReactTypes.get(symbol);
  if (!typeId) {
    typeId = this.knownReactTypes.size + 1;
    this.knownReactTypes.set(symbol, typeId);
  }
  const gid = newGroupId("bracket");
  if (flags.primitiveMatch === "exact") {
    let valId = this.knownReactPropValues.get(node);
    if (!valId) {
      valId = this.knownReactPropValues.size + 1;
      this.knownReactPropValues.set(node, valId);
    }
    return `\\[${typeId},(?<${gid}>\\d+),${valId}\\
    ↪  ]\\[\\/\\k<${gid}>\\]`;
  }
  if (flags.primitiveMatch === "sametype")
    return `\\[${typeId},(?<${gid}>\\d+),\\d+\\]\\[
    ↪  \\/\\k<${gid}>\\]`;
  // ignore
  return `\\[\\d+,(?<${gid}>\\d+),\\d+\\]\\[\\/\\k<
  ↪  ${gid}>\\]`;
}
```

3) Iterable Parser: Parses an iterable node (like an array) by recursively parsing each item and joining the resulting regex fragments.

Code 4 Iterable Parser

```
function isIterable(n: any): n is
↪  Iterable<React.ReactNode> {
  return typeof n === "object" && n != null &&
  ↪  Symbol.iterator in n;
}
private parseIterable(
  node: React.ReactNode,
  parseToken: (n: React.ReactNode) => string
): string {
  const segs: string[] = [];
  let idxResult: IteratorResult<React.ReactNode>,
  ↪  it = (node as any)[Symbol.iterator]();
  while (!(idxResult = it.next()).done) {
    segs.push(parseToken(idxResult.value));
  }
  segs.push(parseToken(idxResult.value));
  return segs.join("");
}
```

4) Fragment and Portal Parser: Handles React fragments and portals, parsing their children recursively.

Code 5 Fragment and Portal Parser

```
private parseFragmentOrPortal(
  node: React.ReactNode,
  parseToken: (n: React.ReactNode) => string
): string {
  const children = ReactIs.isFragment(node)
    ? (node as any).props.children
    : (node as any).children;
  return parseToken(children);
}
```

5) Flag Parser: Handles the `<Flag>` component. Temporarily overrides matching flags (`propsMatch`, `primitiveMatch`) while parsing its children.

Code 6 Flag Parser

```
private parseFlag(
  props: FlagProps,
  parseToken: (n: React.ReactNode) => string,
  flags: Required<FlagProps>
): string {
  const original = { ...flags };
  Object.assign(flags, props);
  const result = parseToken((props as
  ↪  any).children);
  Object.assign(flags, original);
  return result;
}
```

6) Anchor Parser: Converts an anchor spec (`start` or `end`) into the regex equivalents `^` or `$`.

Code 7 Anchor Parser

```
private parseAnchor(kind: "start"|"end"): string {
  return kind === "start" ? "^" : kind === "end" ?
  ↪  "$" : "";
}
```

7) Group Parser: Parses a `<Group>` component into a capture, non-capture, or named group regex, depending on `props.kind`.

Code 8 Group Parser

```
private parseGroup(
  props: GroupProps,
  parseToken: (n: React.ReactNode) => string,
  newGroupId: (t: any, r?: any, n?: any) => string
): string {
  const { kind="capture", id, children } = props;
  if (kind === "capture") {
    const gid = newGroupId("user", children);
    return `(?<${gid}>${parseToken(children)})`;
  }
  if (kind === "non-capture") {
    return `(?:${parseToken(children)})`;
  }
  // named
  const gid = newGroupId("user", children, id);
  return `(?<${gid}>${parseToken(children)})`;
}
```

8) Alternate Parser: Returns a simple regex alternation operator: `|`.

Code 9 Alternate Parser

```
private parseAlternate(): string {
  return "|";
}
```

9) Count Parser: Builds regex quantifiers like `*`, `+`, `?`, `{min,max}`, including their lazy variants when `kind="lazy"`.

Code 10 Count Parser

```
private parseCount({ kind="greedy", min, max }:
↪  CountProps): string {
  // greedy
  if (kind === "greedy") {
    if (min===0 && !isFinite(max)) return "*";
    if (min===1 && !isFinite(max)) return "+";
    if (min===0 && max===1) return "?";
    if (min===max) return `{${min}}`;
    return `{${min},${max}}`;
  }
  // lazy
  if (min===0 && !isFinite(max)) return "*?";
  if (min===1 && !isFinite(max)) return "+?";
  if (min===0 && max===1) return "??";
  if (min===max) return `{${min}}?`;
  return `{${min},${max}}?`;
}
```

10) Look Parser: Parses a lookahead or lookbehind assertion (positive or negative), wrapping the child pattern accordingly.

Code 11 Look Parser

```
private parseLook(
  { kind, polarity, children }: LookProps,
  parseToken: (n: React.ReactNode) => string
): string {
  if (kind==="ahead" && polarity==="positive")
  ↪  return `(?=${parseToken(children)})`;
  if (kind==="ahead" && polarity==="negative")
  ↪  return `(?!${parseToken(children)})`;
  if (kind==="behind" && polarity==="positive")
  ↪  return `(?<=${parseToken(children)})`;
  if (kind==="behind" && polarity==="negative")
  ↪  return `(?<!${parseToken(children)})`;
  return "";
}
```

11) Back Reference Parser: Adds a backreference (e.g. `\1` or `\k<name>`) placeholder into the regex using a queue for deferred resolution.

Code 12 Back Reference Parser

```
private parseBackReference(
  reference: string|number,
  queue: (string|number)[]
): string {
  const idx = queue.push(reference) - 1;
  return `<BR-${idx}>`;
}
```

12) Element Parser: Handles general React element matching: extracts the element's type and props and builds a regex that matches elements based on type, props (depending on `propsMatch`), and children.

Code 13 Element Parser

```
private parseElement(
  node: React.ReactElement,
  parseToken: (n: React.ReactNode) => string,
  flags: Required<FlagProps>,
  newGroupId: (t: any, r?: any, n?: any) => string
): string {
  // assign/get typeId
  const typeId =
  ↪  this.knownReactTypes.get(node.type) ??
    (() => {
      const id = this.knownReactTypes.size+1;
      this.knownReactTypes.set(node.type, id);
      return id;
    })();

  // build children pattern
  const childrenPattern =
  ↪  parseToken(node.props.children);

  // build props-regex
  const entries =
  ↪  Object.entries(node.props).filter(([k]) =>
  ↪  k!=="children");
  let propsPattern = "";
  if (flags.propsMatch === "exact") {
    propsPattern = entries
      .sort(([a],[b]) => a.localeCompare(b))
      .map(([k,v]) => {
        const keyId =
        ↪  this.knownReactPropKeys.get(k) ?? (()
        ↪  => {
          const id=this.knownReactPropKeys.size+1;
          this.knownReactPropKeys.set(k,id);
          return id;
        })();
        const valId =
        ↪  this.knownReactPropValues.get(v) ?? (()
        ↪  => {
          const id=this.knownReactPropValues.size+1;
          this.knownReactPropValues.set(v,id);
          return id;
        })();
        return `${keyId}=${valId}`;
      }).join(",");
  } else if (flags.propsMatch === "includes") {
    if (entries.length===0) {
      propsPattern =
      ↪  node.type===ReactRegExp.AnyToken ?
      ↪  "[\\d=,]*?" :
      ↪  "(?:(?:\\d+=\\d+,)*(?:\\d+=\\d+))?";
    } else {
      propsPattern = "(?:\\d+=\\d+,)*" + entries
        .sort(([a],[b]) => a.localeCompare(b))
        .map(([k,v]) => {
          const keyId =
          ↪  this.knownReactPropKeys.get(k) ?? (()
          ↪  => {
            const id=this.knownReactPropKeys.size+1;
            this.knownReactPropKeys.set(k,id);
            return id;
          })();
          const valId =
          ↪  this.knownReactPropValues.get(v) ??
          ↪  (() => {
            const
            ↪  id=this.knownReactPropValues.size+1;
            this.knownReactPropValues.set(v,id);
            return id;
          })();
          return `${keyId}=${valId}`;
        })
        .join("(?:,(?:\\d+=\\d+,)*)") +
        ↪  "(?:,(?:\\d+=\\d+,)*(?:\\d+=\\d+))?";
```

```
    }
  } else {
    propsPattern =
    ↪  "(?:(?:\\d+=\\d+,)*(?:\\d+=\\d+))?";
  }

  // wrap entire element
  const gid = newGroupId("bracket");
  return `(?:\\[${typeId},(?<${gid}>\\d+),${propsPa
  ↪  ttern}\\]${childrenPattern}\\[\\/\\k<${gid}>\
  ↪  \])`;
}
```

B. React Nodes Representation

After converting the declarative React pattern into a regex pattern, we still need the nodes converted into a string, too. This conversion must conform to the specification we define in the regular expression previously.

Code 14 React Nodes to String Conversion

```
protected compileNodes(nodes: React.ReactNode) {
  let str = "";
  const infos: ReactRegExpExecAdditionalInfo[] = [];
  const recurse = (node: React.ReactNode, parent:
  ↪  React.ReactNode, childId: number) => {
    // primitive case
    if (isPrimitive(node)) {
      const symbol = `PRIMITIVE_${node == null ?
      ↪  "null" : typeof node}`;
      const typeId =
      ↪  this.knownReactTypes.get(symbol) ?? 0;
      const valId =
      ↪  this.knownReactPropValues.get(node) ?? 0;
      const info = { node, parent, childId } as
      ↪  ReactRegExpExecAdditionalInfo;
      const idx = infos.push(info);
      info.patternStart = str.length;
      str += `[${typeId},${idx},${valId}][/${idx}]`;
      info.patternFragment =
      ↪  str.slice(info.patternStart);
      return;
    }
    // iterable
    const it = (node as any)[Symbol.iterator]?.();
    if (it) {
      let idx = 0, res;
      while (!(res = it.next()).done) {
        recurse(res.value, node, idx++);
      }
      recurse(res.value, node, idx++);
      return;
    }
    // fragment/portal
    if (ReactIs.isFragment(node)) {
      recurse((node as any).props.children, node,
      ↪  0);
      return;
    }
    if (ReactIs.isPortal(node)) {
      recurse((node as any).children, node, 0);
      return;
    }
    // element
    if (ReactIs.isElement(node)) {
      const { children, ...props } = node.props;
      const typeId =
      ↪  this.knownReactTypes.get(node.type) ?? 0;
      const propIds = Object.entries(props)
        .sort(([a],[b]) => a.localeCompare(b))
        .map(([k,v]) => {
          const pk = this.knownReactPropKeys.get(k)
          ↪  ?? 0;
          const pv =
          ↪  this.knownReactPropValues.get(v) ?? 0;
          return `${pk}=${pv}`;
        })
        .join(",");
      const info = { node, parent, childId } as
      ↪  ReactRegExpExecAdditionalInfo;
      const idx = infos.push(info);
      info.patternStart = str.length;
      str += `[${typeId},${idx},${propIds}]`;
      recurse(children, node, 0);
      str += `[/${idx}]`;
      info.patternFragment =
      ↪  str.slice(info.patternStart);
    }
  };

  recurse(nodes, null, 0);
  return { knownReactElements: infos,
  ↪  compiledNodes: str };
}
```

During execution, the target React node tree undergoes a parallel traversal that serializes each node into a flat string of identical bracketed tokens, complete with unique numeric indices that correspond to an auxiliary metadata array. The precompiled regular expression, constructed with the global flag, is then applied against this serialized representation. And by the congruent token syntax on both pattern and subject sides, matching precisely identifies sequences of component types, property configurations, and textual or element content according to the originally specified grouping, repetition, and look-around.

The serialized representation is emitted as a bracket-delimited token, indexed by a unique integer. Two token forms recur throughout:

1) Element Tokens: Element tokens are groups that encode the structure and properties of React components in a uniform, bracket-delimited syntax. Each element token begins with an opening bracket of the form

```
[<typeId>,<elementIndex>,<propsIdList>]
<childrenTokens>[/<elementIndex>]
```

- `<typeId>` is a small integer assigned to the component's constructor (or to the special internal tokens)
- `<elementIndex>` is a unique, sequential index for that specific node instance
- `<propsIdList>` is a comma-separated encoding of the node's prop-keys and prop-values (each mapped to an integer), or an empty-match pattern if props are ignored
- `<childrenTokens>` is the concatenation of whatever tokens its children produce
- The closing tag `[/<elementIndex>]` ensures proper nesting and allows the regex engine to identify the span of that element's subtree

2) Primitive Tokens: Primitive tokens mirror the element-token structure but capture simple values (strings, numbers, booleans, null, or undefined) in a bracketed form. Each primitive token is emitted as

[<primitiveTypeId>,<primitiveIndex>, <valueId>][/primitiveIndex]

- <primitiveTypeId> encodes "string", "number", "boolean", or "null/undefined"
- <primitiveIndex> matches the same unique index concept as for elements
- <valueId> is an integer representing the exact primitive value (used only under "exact" matching), or a wildcard placeholder under "sametype" or "ignore" modes

C. Regular Expression Result Extraction

The step is responsible for running the compiled regular expression against a serialized representation of a React-node tree and then reconstructing a structured, node-based match result.

Code 15 Regular Expression Result Extraction

```
public exec(nodes: React.ReactNode):
↪  ReactRegExpExecArray | null {
  const { compiledNodes, knownReactElements } =
  ↪  this.compileNodes(nodes);
  if (this.lastIndex >= knownReactElements.length)
  ↪  this.lastIndex = 0;
  this.compiledRegex.lastIndex = knownReactElements
  ↪  [this.lastIndex]?.patternStart ?? 0;

  const m = this.compiledRegex.exec(compiledNodes);
  if (!m) { this.lastIndex =
  ↪  knownReactElements.length; return null; }

  // compute next lastIndex
  const suffix = compiledNodes.slice(m.index +
  ↪  m[0].length);
  const nextId = +(/\[(\d+),/.exec(suffix)?.[1] ??
  ↪  NaN) - 1;
  this.lastIndex = Number.isNaN(nextId) ?
  ↪  knownReactElements.length : nextId;

  // rebuild match array
  const userGroups = this.groupIds.filter(g =>
  ↪  g.type === "user") as UserGroup[];
  const additionalInfo = userGroups.map(ug =>
  ↪  this.extractGroup(m.groups![ug.id],
  ↪  knownReactElements));
  const result = additionalInfo.map(arr =>
  ↪  arr?.map(i => i.node) ?? null) as
  ↪  ReactRegExpExecArray;

  result.index = +(/\[(\d+),/.exec(m[0])?.[1] ??
  ↪  -1) - 1;
  result.input = nodes;
  result.groups = userGroups.filter(ug =>
  ↪  ug.name).reduce((acc, ug, i) => {
    if (m.groups![ug.id]) acc[ug.name!] = result[i];
    return acc;
  }, {} as Record<string, React.ReactNode[]>);
  result.additionalInfo = additionalInfo;
  return result;
}

public test(nodes: React.ReactNode): boolean {
  const { compiledNodes } =
  ↪  this.compileNodes(nodes);
  return this.compiledRegex.test(compiledNodes);
}
```

The code begins by traversing the supplied React-node tree and serializing each node into a flat string of bracketed tokens, accompanied by an array of metadata objects that record node identity, parent linkage, and the string offset at which each token begins. The method then uses the stored `lastIndex` to set the starting offset for the next match attempt, allowing repeated invocations to continue from the end of the previous match. If `lastIndex` exceeds the number of serialized tokens, it is reset to zero, ensuring wrap-around behavior.

Upon execution of the compiled regular expression, the matching result is examined for named capture groups corresponding to user-defined <Group> components. For each such group, the substring matched by its named group is remapped via the metadata array back to the original React nodes, producing an array of `ReactNode` values for every capture. The code then assembles a `ReactRegExpExecArray`, setting its index to the starting node index of the match, its input to the original tree, and its groups property to any explicitly named groups. An `additionalInfo` field preserves the full metadata for further requirements.

D. React Nodes Modification

Code 16 React Nodes Modification

```
function reactRegexExtract(
  matcher: ReactRegExpExecArray
): ReactRegExpExtractArray {
  // Recursively wrap each matched node (or array
  ↪  of nodes) in a "clone" function
  function buildExtractor(
    original: React.ReactNode | React.ReactNode[],
    path: string
  ): ReactRegExpExtractComponentClone {
    // Determine children for arrays vs. single
    ↪  nodes
    const childNodes = Array.isArray(original)
      ? original
      : ReactIs.isElement(original) ||
      ↪  ReactIs.isPortal(original) ||
      ↪  ReactIs.isFragment(original)
      ? React.Children.toArray((original as
      ↪  any).props.children ?? (original as
      ↪  any).children)
      : [];

    // Recursively build extractors for each child
    const childExtractors = childNodes.map((n, i) =>
      buildExtractor(n, `${path}.${i}`)
    );

    // The actual "clone" function returned to the
    ↪  user
    const cloneFn = ((addedProps: any) => {
      if (addedProps &&
      ↪  Object.keys(addedProps).length > 0)
        throw new Error("Cannot add props to this
        ↪  node");
      // Elements get cloned, others get wrapped in
      ↪  a Fragment
      if (ReactIs.isElement(original)) {
        return React.cloneElement(
          original,
          { key: path, ...original.props },
          addedProps?.children ??
          ↪  original.props.children
        );
      } else {
        return React.createElement(
          React.Fragment,
```

```
          { key: path },
          original
        );
      }
    }) as ReactRegExpExtractComponentClone;

    // Attach metadata
    cloneFn.node = original;
    cloneFn.props = ReactIs.isElement(original) ?
    ↪  (original as any).props : null;
    cloneFn.children = childExtractors;
    childExtractors.forEach((ch, i) => {
      cloneFn[`c${i}`] = ch;
    });
    (cloneFn as any)[componentCloneSymbol] = true;

    return cloneFn;
  }

  // Build extractors for each top-level match
  const extractors = matcher.map((node, i) =>
    buildExtractor(node, `ROOT.${i}`)
  ) as ReactRegExpExtractArray;

  // .matches = dictionary of all by index
  extractors.matches = Object.fromEntries(
    extractors.map((fn, i) => [`c${i}`, fn])
  );

  // .groups = named groups, mapped into the
  ↪  extractor array
  if (matcher.groups) {
    extractors.groups = Object.fromEntries(
      Object.entries(matcher.groups).map(([name,
      ↪  nodes]) => {
        if (!nodes) return [name, undefined];
        // find first node in that group
        const idx =
        ↪  matcher.indexOf(Array.isArray(nodes) ?
        ↪  nodes[0] : nodes);
        return [name, extractors[idx]];
      })
    );
  }

  return extractors;
}
```

The `reactRegexExtract` function transforms the raw `ReactRegExpExecArray`, which is simply an ordered list of matched React nodes and optional named groups, into a hierarchical set of "extractor" functions that each wrap a specific match, preserve its original node and props, and recursively expose its children as further extractors. Each extractor is a callable clone factory (no additional props are accepted) tagged with metadata (`.node`, `.props`, `.children`, and indexed `.cN` properties) that allows reconstruction or modification of the matched subtree. Finally, the top-level array is augmented with a `.matches` object for direct, index-keyed access to each extractor and a `.groups` object mapping any named capture groups to their corresponding extractors. All in all, it provides a convenient API for post-match inspection and manipulation.

V. Results and Discussion

To make the implementation more clear, let's take a look at one example. Consider this use case, a navigation component that requires a specific layout for its child to render and style accordingly, whether the children are logo, legal notice, links, or social links.

Code 17 Navigation Component Use Case

```
const elementChildrenRegex = new ReactRegExp(({
↪  Anchor, Group, Any, Count }) => (
  <>
    <Anchor kind="start" />
    <Group kind="named" id="logo">
      <a navigation-role="logo">
        <Any /><Count min={0} max={1} />
      </a>
    </Group>

    <Group kind="named" id="links">
      <ul navigation-role="links">
        <li>
          <Any /><Count min={0} max={Infinity} />
        </li>
        <Count min={0} max={Infinity} />
      </ul>
    </Group>

    <Group kind="named" id="contacts">
      <ul navigation-role="contacts">
        <li>
          <Any /><Count min={0} max={Infinity} />
        </li>
        <Count min={0} max={Infinity} />
      </ul>
    </Group>

    <Group kind="named" id="legalNotice">
      <div navigation-role="legal-notice">
        <Any /><Count min={0} max={Infinity} />
      </div>
    </Group>
    <Anchor kind="end" />
  </>
));

export default function NavigationElement(props:
↪  Props) {
  const { children: childrenNode, ...htmlProps } =
  ↪  { ...props };
  elementChildrenRegex.lastIndex = 0;
  const elementRef = useRef<HTMLElement>();
  const element = useElementSelector(elementRef,
  ↪  selector);
  const children =
  ↪  useReactRegex(elementChildrenRegex,
  ↪  childrenNode);

  const Tag = "x-navigation" as any;
  return (
    <Tag
      ref={elementRef}
      {...{ ...htmlProps, className: null }}
      class={style`navigation
↪  ${visibility.matches("shown") ||
↪  visibility.matches("transitioningShow") ?
↪  "active" : ""} /${props.className}/`}
    >
      <div className={style`bar`}>
        <div className={style`content`}>
          <div className={style`floating left`}>
            {children!.groups!.logo!.c0(Logo =>
            ↪  (<Logo className={style`logo
            ↪  /${Logo.props.className}/`} />))}
          </div>
          <div className={style`floating right`}>
            <button
              className={style`hamburger`}
```

```
            title={visibility.matches("shown") ||
↪  visibility.matches("transitioningShow") ? "Open
↪  Navigation" : "Close Navigation"}
            onClick={() => sendVisibility({ type:
↪  "TOGGLE" })}
          />
        </div>
      </div>
    </div>
    <div className={style`full`}>
      <div className={style`background`}>
        <div></div>
        <div></div>
        <div></div>
      </div>
      <div className={style`content`}>
        <div className={style`links`}>
          {children!.groups!.links!.c0(Links =>
          ↪  (<Links />))}
        </div>
        <div className={style`contacts`}>
          {children!.groups!.contacts!.c0(Contact
          ↪  s => (<Contacts />))}
        </div>
        <div className={style`legal-notice`}>
          {children!.groups!.legalNotice!.c0(Lega
          ↪  lNotice => (<LegalNotice />))}
        </div>
        <div className={style`overlays`}></div>
      </div>
    </div>
  </Tag>
 );
}
```

The resulting regular expression pattern is as follows.

```
(?<_1>^ (?<_2>(?:\[1,(?<_4>\d+),(?:\d+=\d+,)*1=1(?:,(?:\d+=\d+,)*(?:\d+=\d+))?\](?:\[\d+,(?<_3>\d+),[\d=,]*?\](?:\[\/\d+\]|\[\d+,\d+,[\d=,]*?\])*?\[\/\k<_3>\])?\[\/\k<_4>\])) (?<_5>(?:\[2,(?<_8>\d+),(?:\d+=\d+,)*1=2(?:,(?:\d+=\d+,)*(?:\d+=\d+))?\](?:\[3,(?<_7>\d+),(?:(?:\d+=\d+,)*(?:\d+=\d+))?\](?:\[\d+,(?<_6>\d+),[\d=,]*?\](?:\[\/\d+\]|\[\d+,\d+,[\d=,]*?\])*?\[\/\k<_6>\])*\[\/\k<_7>\])*\[\/\k<_8>\])) (?<_9>(?:\[2,(?<_12>\d+),(?:\d+=\d+,)*1=3(?:,(?:\d+=\d+,)*(?:\d+=\d+))?\](?:\[3,(?<_11>\d+),(?:(?:\d+=\d+,)*(?:\d+=\d+))?\](?:\[\d+,(?<_10>\d+),[\d=,]*?\](?:\[\/\d+\]|\[\d+,\d+,[\d=,]*?\])*?\[\/\k<_10>\])*\[\/\k<_11>\])*\[\/\k<_12>\])) (?<_13>(?:\[4,(?<_15>\d+),(?:\d+=\d+,)*1=4(?:,(?:\d+=\d+,)*(?:\d+=\d+))?\](?:\[\d+,(?<_14>\d+),[\d=,]*?\](?:\[\/\d+\]|\[\d+,\d+,[\d=,]*?\])*?\[\/\k<_14>\])*\[\/\k<_15>\])) $)
```

The sections in regular expression match with the `logo`, `links`, `contacts`, and `legalNotice` groups, respectively.

Code 18 Navigation Component Invocation

```
<Navigation>
  <a navigation-role="logo" href="/"><Picture
  ↪  src={logo} priority /></a>
  <ul navigation-role="links">
    <li><a href="/">Home</a></li>
    <li><a href="/about-us">About Us</a></li>
    <li><a href="/archive">Archive</a></li>
    <li><a href="/contact-us">Contact Us</a></li>
  </ul>
  <ul navigation-role="contacts">
    <li><a href="https://facebook.com/"
    ↪  target="_blank_socials_facebook_osispk"
    ↪  aria-label="Facebook">facebook</a></li>
    <li><a href="https://instagram.com/"
    ↪  target="_blank_socials_instagram_osispk"
    ↪  aria-label="Instagram">instagram</a></li>
  </ul>
  <div navigation-role="legal-notice">
    <p>
      This website uses cookies to improve your
      ↪  experience.
      By accessing this website you consent to the
      ↪  use of cookies.
    </p>
  </div>
</Navigation>
```

When tested against the previous code, the generated flatten React node string representation is:

```
[1,1,0=0,1=1] [0,2,0=0,0=0] [/2] [/1] [2,3,1=2] [3,4, ] [1,5,0=0] [0,6,0] [/6] [/5] [/4] [3,7, ] [1,8,0=0] [0,9,0] [/9] [/8] [/7] [3,1 0, ] [1,1 1,0=0] [0,1 2,0] [/1 2] [/1 1] [/1 0] [3,1 3, ] [1,1 4,0=0] [0,1 5,0] [/1 5] [/1 4] [/1 3] [/3] [2,1 6,1=3] [3,1 7, ] [1,1 8,0=0,0=0,0=0] [0,1 9,0] [/1 9] [/1 8] [/1 7] [3,20,] [1,21,0=0,0=0,0=0] [0,22,0] [/22] [/21] [/20] [/16] [4,23,1=4] [0,24,] [0,25,0] [/25] [/24] [/23]
```

As shown above, the regular expression successfully matches the text that represents the correct hierarchy (each capture group is colored appropriately). This includes property matching, type checking, quantifiers, and more. The use case also provides an outline on how to modify the children it is passed on.

VI. Conclusion

The process consists of three stages. First, a concise JSX "pattern" composed of components such as `<Group>`, `<Count>`, `<Look>`, etc. is converted into a single comprehensive regular expression by dedicated parsers for primitives, arrays, fragments, anchors, alternations, quantifiers, look-arounds, back-references, and standard React elements. Second, the target React component tree is traversed and serialized into a bracket-delimited token string that aligns exactly with the generated regex. Third, the regular expression is executed against this serialized representation and the resulting matches are mapped back to the original React nodes. Each matched node and its subtree are exposed through a small extractor function, preserving original props, keys, and children, while `.matches` and `.groups` lookup objects provide convenient access to all captured fragments.

APPENDIX

The code implementation for the methods and experiments discussed in this paper can be found in the following GitHub repository: https://github.com/NadhifRadityo/react-regex.

Please explore the repository for a deeper understanding of the implementation details and for any potential extensions of the methodology. For issues or questions regarding the repository, please refer to the provided documentation or contact the repository maintainer directly.

ACKNOWLEDGMENT

The author expresses profound gratitude to Dr. Nur Ulfa Maulidevi, S.T, M.Sc., Dr. Ir. Rinaldi Munir, M.T., and Monterico Adrian, S.T., M.T. for his invaluable guidance and insights as the IF2211 Algorithm Strategies course lecturer, which greatly contributed to the development of this paper.

The author also expresses an apologies for any shortcomings that may remain in this work. It is sincerely hoped that this paper will serve as a useful reference for future studies and research purposes.

REFERENCES

[1] Jurafsky, D., Martin J. H. 2000. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.

[2] Aho Alfred V., Ullman, Jeffrey D. 1992. Chapter 10. Patterns, Automata, and Regular Expressions.

[3] Cox Russ. 2007. Regular Expression Matching Can Be Simple and Fast.

[4] Gruber H., Holzer M. 2008. Finite Automata, Digraph Connectivity, and Regular Expression Size. doi:10.1007/978-3-540-70583-3_4. ISBN 978-3-540-70582-6.

[5] Meta. 2025. React 19.

[6] Aggarwal, S. 2018. Modern Web-Development using ReactJS. International Journal of Recent Research Aspects. pp. 133–137.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 21 Juni 2025

Nadhif Radityo N. (13523045)