

Reverse React Notation (RRN): Simplifying React syntax with FORTH-like Reverse Polish Notation and Stack Machine Architecture

Liang Ng

April 2022

Conventional React Syntax:

```
<Route path=":invoiceId" element={<Invoice />} />
```

Reverse React Notation (RRN)

```
{ f( <Invoice />, ":invoiceId", "path:" ) }
```

- RRN Repository: <https://github.com/udexon/RRN>

Programming languages have evolved a long way since the days of Fortran and COBOL in the 1950s, with React being one of the latest and hottest programming languages bewildering young programmers' minds in 2022.

In this article, we attempt to demonstrate how React syntax can be simplified using FORTH-like Reverse Polish Notation and Stack Machine Architecture, hence the name "Reverse React Notation" (RRN).

We will use the bookkeeper example from the following websites:

- <https://stackblitz.com/edit/github-agqlf5?file=src/App.jsx>
- <https://reactrouter.com/docs/en/v6/getting-started/tutorial>

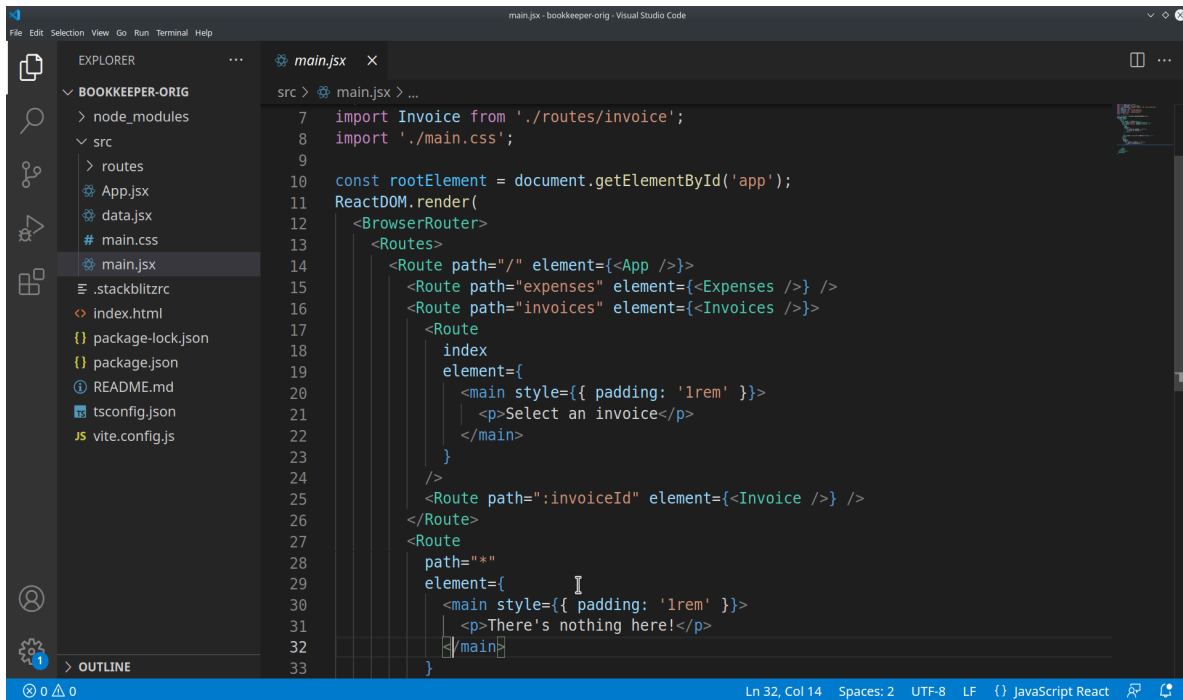


Figure 1

React can be confusing to beginners, as its high level main program (main.jsx) consists of routing code, which can be difficult to visualise. To complicate matters further, React code mixes HTML with JavaScript (or Typescript). Reverse React Notation (RRN) aims to solve these problems.

```

19 ReactDOM.render(
20   <BrowserRouter>
21     <Routes>
22       {
23         // r_App()
24         // f_children()
25         f_chld()
26       }
27       {
28         // r_App_child()
29         r( <App />, "/", "Route_ch:")
30       }
31     </Routes>
32   </BrowserRouter>,
33   rootElement
34 );
35

```

Figure 2

```

35
36 function f_chld() {
37   f( <Expenses />, "expenses", "Route:" )
38
39   f( 'Select an invoice 888', 'index:' )
40   f( <Invoice />, ":invoiceId", "Route:" )
41   f( 'merge:' )
42
43   f( 'Invoices:' ) // need to import Invoices in libdom
44   f( 'merge:' )
45
46   f( 'There is nothing 000', 'h_p: r_main: * Route:')
47   f( 'merge:' )
48 }
49

```

Figure 3

Figures 2 and 3 show the equivalent of the program in figure 1 written in Reverse React Notation. The compactness and indentation make RRN easier to understand and clearly show the structure.

Example 1: Route

Original React code (line 25):

```

<Route path=":invoiceId" element={<Invoice />} />

```

Figure 4

Reverse React Notation (RRN) code:

```

{ S.push( <Invoice /> ) }
{ S.push( ":invoiceId" ) }
{ S.push( <Route path={S.pop()} element={S.pop()} /> ) }

```

Figure 5

Let us explain the principles of Reverse React Notation (RRN) using Reverse Polish Notation (RPN).

One of the simplest Reverse Polish Notation (RPN) example would be addition, which is written as:

3 4 +

The operator is written after the operands, hence the name “reverse” (also known as “postfix”, in contrast with the “infix” convention where the operator is written in between operands:

3 + 4

Comparing "3 4 +" to RRN code above, the steps involved are:

```
S.push( 3 )  
S.push( 4 )  
S.push( S.pop() + S.pop() )
```

S, the data stack, is simply a JavaScript (Typescript) array initialised with:

```
var S = [ ]
```

Data tokens 3, 4 are pushed onto the stack (line 1 and 2). Upon encountering a function token + (plus), the required operands on the stack 3, 4 are popped, added and the result is pushed back onto the stack (line 3).

The same principles apply in figure 5 for RRN code.

Now some readers may be wondering how would 1 line of code in figure 2 be simplified as 3 lines of code in figure 3?

Be patient. This is only the first step. We just want to show you the fundamental principles of stack machine and reverse polish notation.

After some coding gymnastics, we may reduce the code in figure 5 into the following:

```
f( <Invoice />, ":invoiceId", "Route:" )
```

Figure 6

Figure 6 shows the syntax of Reverse Polish Notation, where:

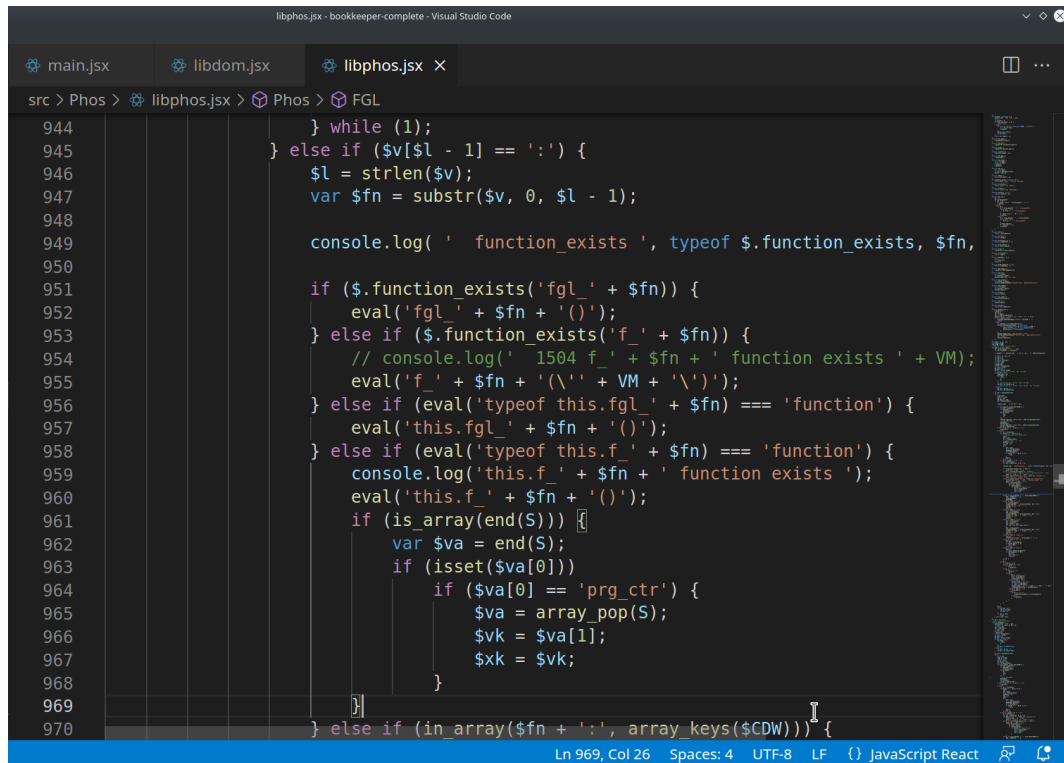
f(arg[0], arg[1], , arg[n-2], arg[n-1])

- All arguments from index 0 through [n-2] are data tokens (variables).
- arg[n-1] is a space delimited string, consisting of data tokens and function tokens.

```
function f_Route() {  
  var S=window.M.S  
  S.push(<Route path={S.pop()} element={S.pop()} />)  
}
```

Figure 7

"Route:" a function token (figure 6) is mapped to `f_Route()` (figure 7) in line 953 in stack machine function FGL() (figure 8).



```
944     } while (1);
945   } else if ($v[$l - 1] == ':') {
946     $l = strlen($v);
947     var $fn = substr($v, 0, $l - 1);
948
949     console.log( ' function_exists ', typeof $.function_exists, $fn,
950
951     if ($.function_exists('fgl_' + $fn)) {
952       eval('fgl_' + $fn + '()');
953     } else if ($.function_exists('f_' + $fn)) {
954       // console.log(' 1504 f_' + $fn + ' function exists ' + VM);
955       eval('f_' + $fn + '()' + VM + '()');
956     } else if (eval('typeof this.fgl_' + $fn) === 'function') {
957       eval('this.fgl_' + $fn + '()');
958     } else if (eval('typeof this.f_' + $fn) === 'function') {
959       console.log('this.f_' + $fn + ' function exists ');
960       eval('this.f_' + $fn + '()');
961       if (is_array(end(S))) {
962         var $va = end(S);
963         if (isset($va[0]))
964           if ($va[0] == 'prg_ctr') {
965             $va = array_pop(S);
966             $vk = $va[1];
967             $xk = $vk;
968           }
969       }
970     } else if (in_array($fn + ':', array_keys($CDW))) {
```

Figure 8

Line 37 and 46 in figure 2 (shown below for convenience) consists of examples similar Route: function.

```
f( <Expenses />, "expenses", "Route:" )

f( 'There is nothing here!', 'h_p: r_main: * Route:')
```

Figure 9

Line 46 (figure 2) maps to line 27 (figure 1), as shown below:

```
f( 'There is nothing here!', 'h_p: r_main: * Route:')
```

```

<Route
  path="*"
  element={
    <main style={{ padding: '1rem' }}>
      <p>There's nothing here!</p>
    </main>
  }
/>

```

Figure 10

1. 'There is nothing here!' is pushed onto the stack.
2. 'h_p:' maps to f_h_p(), which simply creates an HTML element <p> from the top of stack (TOS) element:

```

function f_h_p() { // html p
  var S = window.M.S;
  S.push(<p>{S.pop()}</p>);
}

```

Figure 10

A VERY important property in RPN (Reverse Polish Notation), RRN (Reverse React Notation) and other FORTH-like programming languages is CONCATENATION, i.e. tokens (data and function tokens) will be processed one after another when they are written as a space delimited string.

In this case, the input string is:

```
'h_p: r_main: * Route:'
```

So the tokens are:

1. h_p: (function token mapped to f_h_p())
2. r_main: (function token mapped to f_r_main())
3. * (a literal * character)
4. Route: (function token mapped to f_Route())

Functions will be executed consecutively, taking input from data items on the stack, pushing back results to the stack.

As each of the function tokens can represent something fairly complex, it can reduce something quite complex (figure 10) into a very compact expression i.e. a space delimited string.

Example 2: Attributes in Component

```
<Route
  index
  element={
    <main style={{ padding: '1rem' }}>
      <p>Select an invoice</p>
    </main>
  }
/>
```

Figure 11

Example 2 (figure 11) is similar to example 1 (figure 7), except instead of `path="*" we have index. So we map the word index: (a FORTH term meaning token) to:`

```
function f_index() {
  const props = {index: true, element: r('h_p: r_main:')} };
  window.M.S.push(<Route {...props} />);
}
```

Figure 12

As you can see, we can easily change the function name to whichever word that we think it is easier to understand and remember. In fact, aliases can be created simply by substitution and concatenation, e.g.

```
function f_element() {
  f_index()
}
```

Figure 13

This creates an alias for `index`: called `element`:

`sm`: is a concatenation of `swap`: `merge`:

```
function f_sm() { // swap: merge:
  f_swap(); f_merge();
}
```

Figure 14

```

18
19 ReactDOM.render(
20   <BrowserRouter>
21     <Routes>
22       {
23         // r_App()
24         // f_children()
25         f_chld()
26       }
27       {
28         // r_App_child()
29         r( <App />, "/", "Route_ch:")
30       }
31     </Routes>
32   </BrowserRouter>,
33   rootElement
34 );
35

```

Figure 2

```

35
36 function f_chld() {
37   f( <Expenses />, "expenses", "Route:" )
38
39   f( 'Select an invoice 888', 'index:' )
40   f( <Invoice />, ":invoiceId", "Route:" )
41   f( 'merge:' )
42
43   f( 'Invoices:' ) // need to import Invoices in libdom
44   f( 'merge:' )
45
46   f( 'There is nothing 000', 'h_p: r_main: * Route:')
47   f( 'merge:' )
48 }
49

```

Figure 3

Example 3: The Rest of It

Let us copy figure 2 and 3 here for convenience. We have covered all items except the following:

```

f( 'merge:' )
f( 'Invoices:' )
r( <App />, "/", "Route_ch:")

```

merge: is mapped to f_merge(), which is a necessary construct in React to combine multiple children into one node using '<>...</>', before it can be added to a parent node:

```

function f_merge() {
  var S=window.M.S
  S.push(<>{S.pop()} {S.pop()}</>)
}

```

Figure 15

Invoices: and Route_ch: are similar, as shown in f_Invoices() (figure 16) and f_Route_ch() (figure 17) below. In f_Invoices(), element is hard coded to <Invoices />, while in f_Route_ch(), element is assigned to the top of stack (TOS) item. f_Route_ch() and f_Route() are the same, except that f_Route_ch() contains a child node.

```
f( 'Invoices:' )  
r( <App />, "/", "Route_ch:")
```

```
function f_Invoices() {  
  var S=window.M.S  
  S.push(<Route path="invoices" element={<Invoices />}>  
    { S.pop() }  
  </Route>)  
}
```

Figure 16

```
function f_Route_ch() {  
  var S=window.M.S  
  S.push(<Route path={S.pop()} element={S.pop()}> {S.pop()} </Route>)  
}
```

Figure 17

Conclusions

Reverse React Notation (RRN) is a name in the tongue-in-cheek fashion, derived from Reverse Polish Notation. The biggest immediate contributions of RRN are to break the myths that React and other “modern” programming languages cannot be simplified, and that old programming languages like FORTH are obsolete.

These two achievements may sound trivial, but they remove two huge psychological barriers amongst younger programmers today, and will have a profound effect on their careers, as few realise that React and other “modern” programming languages are “designed” to “retire programmers by 35 years old”, and RRN, though with a simple demonstration above, may just save millions of “soon to be older” programmers’ jobs.

We shall leave such debates to other articles, as we hope this article is the beginning of an avalanche effect that may change the landscapes of programming languages.