Liyad aaabbb

Let's make your yet another DSL with Lisp S-expression!

Liyad (Lisp yet another DSL interpreter, or LIYAD is yum and delicious) is very small Lisp interpreter written in JavaScript.
You can easily start making your new DSL using Lisp and S-expression.

npm v0.2.4 release v0.2.4 build passing Fork 0 Star 10

bar

Install

from NPM:

```
$ npm install liyad --save
```

or download from release page.

Install CLI

See liyad-cli .

```
$ npm install -g liyad-cli
$ liyad
```

Playground

https://shellyln.github.io/liyad/playground.html

Features

- APIs to customize all operators and macros
- Builtin S-expression parser
- Builtin minimal Lisp interpreter
- Reference implementation of LSX (alternative JSX notation using Lisp)

Real world examples

- Ménneu
  Component-based extensible document processor
- mdne - Markdown Neo Edit
  A simple markdown and code editor powered by Markdown-it, Ace and Carlo.

What is LSX

LSX is an alternative JSX notation using Lisp.

LSX and Liyad advantages:

- No transpiler needed
    - Liyad uses ES6 template literal syntax.
      You don't pass the entire code to transpile and evaluate it.
      Save your coding times.
- Secure execution for untrusted contents
    - No host environment's symbols are accessible from evaluated user contents by default.
      Malicious codes can not make a serious attack.
- Simple and powerful
    - What you can do with JSX can be done with LSX.
      Plus, LSX itself is a complete data description format and is a complete programming language,
      so you can write more concise and powerful.

The LSX runtime directly calls `React.createElement` (or a JSX Factory function such as RedAgate, Vue.js, etc.) as a Lisp function,
Convert a Lisp list to a renderer component object tree.

In order to resolve the renderer component, you must register the object's constructor with the LSX runtime in advance.

All unresolved lisp function symbols are dispatched to `React.createElement('some_unresolved_name', ...)`.
You can declare HTML/XML standard tags.

As with JSX, LSX must always return a single component.
Using `Template` Lisp function instead of JSX `Fragment` tag will produce the same result.

Example:

```
lsx`
(Template
    (select (@ (style (display "inline-block")
                      (width "300px") )
               (className "foo bar baz")
               (onChange <span style="background-color: pink;"><strong>(</strong>e) => t
        ($=for <span style="background-color: pink;"><strong>e</strong>xampleCodes</span
            ($=if (== (% $index 2) 1)
                (option (@ (value $index)) ($concat "odd: " ($get $data "name"))) )
            ($=if (== (% $index 2) 0)
                (option (@ (value $index)) ($concat "even: " ($get $data "name"))) ))))`
```

See also:

Playground's source code is written in LSX.

Usage

Output S-expression into JSON:

```
import { S } from 'liyad';

console.log(
    JSON.stringify(S`
        ($list
            1 2 3 "a" "b" "C"
            ($list 4 5 6) <span style="background-color: pink;"><strong>"</strong>X"</sp
        // You can also parse by calling w/o template literal syntax as following:
        // S(' ... ')
    )
);
```

Output:

```
[{"symbol":"$list"},1,2,3,"a","b","C",[{"symbol":"$list"},4,5,6],{"value":"X"},{"value":
```

Run minimal Lisp interpreter:

```
import { lisp } from 'liyad';

console.log(
    JSON.stringify(lisp`
        ($defun fac (n)
            ($if (== n 0)
                1
                (* n ($self (- n 1))) ))
        ($list
            1 2 (fac 3) "a" "b" "c"
            ($list 4 5 (fac 6) <span style="background-color: pink;"><strong>"</strong>X

        // You can also evaluate by calling w/o template literal syntax as following:
        // lisp(' ... ')
    )
);
```

Output:

```
[1,2,6,"a","b","c",[4,5,720,"X",["Y","Z"]]]
```

Render web page with LSX:

```
import * as React    from 'react';
import * as ReactDOM from 'react-dom';
import { LSX }       from 'liyad';

var lsx = null;

const exampleCodes = [{
    name: "Example1: factorial",
    code: ` ... `
}, {
    name: "Example2: Hello, World!",
    code: ` ... `,
}];

class ExampleLoader extends React.Component {
    constructor(props, context) {
        super(props, context);
        this.state = {};
    }

    handleExampleSelected(i) {
        this.props.loadExample(i);
    }

    render() {
        return (lsx`
        (Template
            (select (@ (style (display "inline-block")
                              (width "300px") )
                       (onChange <span style="background-color: pink;"><strong>(</strong
                ($=for <span style="background-color: pink;"><strong>e</strong>xampleCod
                    (option (@ (value $index)) ($get $data "name")) )))`);
    }
}

class App extends React.Component {
    constructor(props, context) {
        super(props, context);
        this.state = {};
    }

    loadExample(i) {
        console.log(exampleCodes[i].code);
    }

    render() {
        return (lsx`
        (Template
            (div (@ (style (margin "4px")))
                (ExampleLoader  (@ (loadExample <span style="background-color: pink;"><s
    }
}

var lsx = LSX({
    jsx: React.createElement,
    jsxFlagment: React.Fragment,
    components: {
        ExampleLoader,
        App,
    },
});

ReactDOM.render(lsx`(App)`, document.getElementById('app'));
```

Build your new DSL:

```
import { SxFuncInfo,
         SxMacroInfo,
         SxSymbolInfo,
         SExpression,
         SxParserConfig,
         defaultConfig,
         installCore,
         installArithmetic,
         installSequence } from 'liyad';

const myOperators: SxFuncInfo[] = [{
    name: '$__defun',
    fn: (state: SxParserState, name: string) => (...args: any[]) => {
        // S expression: ($__defun 'name '(sym1 ... symN) 'expr ... 'expr)
        //  -> S expr  : fn
        const car: SxSymbol = $$first(...args);
        if (args.length < 3) {
            throw new Error(`[SX] $__defun: Invalid argument length: expected: <span sty
        }
        const fn = $__lambda(state, name)(...args.slice(1));
        state.funcMap.set(car.symbol, {
            name: car.symbol,
            fn: (st, nm) => fn
        });
        return fn;
    },
}];

const myMacros: SxMacroInfo[] = [{
    name: '$defun',
    fn: (state: SxParserState, name: string) => (list) => {
        // S expression: ($defun name (sym1 ... symN) expr ... expr)
        //  -> S expr  : ($__defun 'name '(sym1 ... symN) 'expr ... 'expr)
        return [{symbol: '$__defun'},
            ...(list.slice(1).map(x => quote(state, x))),
        ];
    },
}];

const mySymbols: SxSymbolInfo[] = [
    {name: '#t', fn: (state: SxParserState, name: string) => true}
];

export const MyDSL = (() => {
    let config: SxParserConfig = Object.assign({}, defaultConfig);

    config = installCore(config);
    config = installArithmetic(config);
    config = installSequence(config);
    config.stripComments = true;

    config.funcs = (config.funcs || []).concat(myOperators);
    config.macros = (config.macros || []).concat(myMacros);
    config.symbols = (config.symbols || []).concat(mySymbols);

    return SExpression(config);
})();


console.log(
    JSON.stringify(MyDSL`( ... )`)
);
```

Extended syntax

Here document:

```
"""
Hello, Liyad!
"""
```

is equivalent to:

```
(Template
"
Hello, Liyad!
"
)
```

Here document with variable substitution:

```
"""
Hello, %%%($get name)!
"""
```

is equivalent to:

```
(Template
"
Hello, " ($get name) "!
"
)
```

Here document with custom function:

```
"""div
Hello, %%%($get name)!
"""
```

is equivalent to:

```
(div
"
Hello, " ($get name) "!
"
)
```

Here document with custom function and LSX props:

```
"""div@{(id "123") (class "foo bar baz")}
Hello, %%%($get name)!
"""
```

is equivalent to:

```
(div (@ (id "123") (class "foo bar baz"))
"
Hello, " ($get name) "!
"
)
```

Spread operator

```
($list 1 2 ...($concat (3 4) (5 6)) 7 8)
```

is equivalent to:

```
($list 1 2 ($spread ($concat (3 4) (5 6))) 7 8)
```

and is to be:

```
[1,2,3,4,5,6,7,8]
```

`$spread` is NOT a macro. The list passed as a parameter is spliced after evaluation.

Splice macro

```
($list 1 2 3 4 ($splice (5 6 7 8)) 9 10)
```

is equivalent to:

```
($list 1 2 3 4 5 6 7 8 9 10)
```

```
(($splice ($call x add)) 5 7)
```

is equivalent to:

```
($call x add 5 7)
```

Shorthands

$set

```
(::foo:bar:baz= 7)
```

is equivalent to:

```
($set ("foo" "bar" "baz") 7)
```

$get

```
($list ::foo:bar:baz)
```

is equivalent to:

```
($list ($get "foo" "bar" "baz"))
```

$call

```
(::foo:bar@baz 3 5 7)
```

is equivalent to:

```
($call ($get "foo" "bar") baz 3 5 7)
```

Rest parameter

```
($defun f (x ...y)
    ($list x y) )

($list
    (f 1)
    (f 1 2)
    (f 1 2 3)
    (f 1 2 3 4)
    (f 1 2 3 4 5) )
```

is to be:

```
[
    [1,[]],
    [1,[2]],
    [1,[2,3]],
    [1,[2,3,4]],
    [1,[2,3,4,5]]
]
```

Verbatim string literal

Verbatim string literal

```
($last @"c:\documents\files\u0066.txt")
```

is to be:

```
"c:\\documents\\files\\u0066.txt"
```

Normal string literal

```
($last "c:\documents\files\u0066.txt")
```

is to be:

```
"c:documents\filesf.txt"
```

APIs

`SExpression`

Create a new DSL.

```
interface SxParserConfig {
    raiseOnUnresolvedSymbol: boolean;
    enableEvaluate: boolean;
    enableHereDoc: boolean;
    enableSpread: boolean;
    enableSplice: boolean;
    enableShorthands: boolean;
    enableVerbatimStringLiteral: boolean;
    enableTailCallOptimization: boolean;
    stripComments: boolean;
    wrapExternalValue: boolean;
    reservedNames: SxReservedNames;
    returnMultipleRoot: boolean;
    maxEvalCount: number;

    jsx?: (comp: any, props: any, ...children: any[]) => any;
    JsxFragment?: any;

    funcs: SxFuncInfo[];
    macros: SxMacroInfo[];
    symbols: SxSymbolInfo[];

    funcSymbolResolverFallback?: SxFunc;
    valueSymbolResolverFallback?: SxSymbolResolver;
}
```

```
function SExpression(config: SxParserConfig): (strings: TemplateStringsArray | string, .
```

- returns : Template literal function.
- `config` : Parser config.

`S`

Parse a S-expression.

```
function S(strings: TemplateStringsArray | string, ...values?: any[]): SxToken
```

- returns : S-expression parsing result as JSON object.
- `strings` : Template strings.
- `values` : values.

`lisp`

Evaluate a Lisp code.

```
function lisp(strings: TemplateStringsArray | string, ...values?: any[]): SxToken
```

- returns : Evalueting result value of Lisp code.
  - If input Lisp code has multiple top level parenthesis, result value is last one.
- `strings` : Template strings.
- `values` : values.

`lisp_async`

Evaluate a Lisp code.
(asynchronous features are enabled.)

```
function lisp_async(strings: TemplateStringsArray | string, ...values?: any[]): Promise<
```

- returns : Promise that evalueting result value of Lisp code.

    - If input Lisp code has multiple top level parenthesis,
      result value is last one.
- `strings` : Template strings.
- `values` : values.

`LM`

Evaluate a Lisp code (returns multiple value).

```
function LM(strings: TemplateStringsArray | string, ...values?: any[]): SxToken
```

- returns : Evalueting result value of lisp code.
    - If input Lisp code has multiple top level parenthesis,
      result value is array.
- `strings` : Template strings.
- `values` : values.

`LM_async`

Evaluate a Lisp code (returns multiple value).
(asynchronous features are enabled.)

```
function LM_async(strings: TemplateStringsArray | string, ...values?: any[]): Promise<Sx
```

- returns : Promise that evalueting result value of lisp code.
    - If input Lisp code has multiple top level parenthesis,
      result value is array.
- `strings` : Template strings.
- `values` : values.

`LSX`

Evaluate a Lisp code as LSX.

```
interface LsxConfig {
    jsx: (comp: any, props: any, ...children: any[]) => any;
    jsxFlagment: any;
    components: object;
```

```
}

function LSX<R = SxToken>(lsxConf: LsxConfig): (strings: TemplateStringsArray, ...values
```

- returns : Template literal function.
- `lsxConf` : LSX config.

`LSX_async`

Evaluate a Lisp code as LSX.
(asynchronous features are enabled.)

```
interface LsxConfig {
    jsx: (comp: any, props: any, ...children: any[]) => any;
    jsxFlagment: any;
    components: object;
}

function LSX_async<R = SxToken>(lsxConf: LsxConfig): (strings: TemplateStringsArray, ...
```

- returns : Template literal function.
- `lsxConf` : LSX config.

(`lisp` | `lisp_async` | `LM` | `LM_async` : SExpressionTemplateFn) methods

evaluateAST

```
evaluateAST(ast: SxToken[]): SxToken;
```

- returns : evaluation result value.
- `ast` : AST to evaluate.

repl

```
repl(): SExpressionTemplateFn;
```

- returns : Template literal function that will keep variables and states for each evaluation.

setGlobals

```
setGlobals(globals: object): SExpressionTemplateFn;
```

- returns : myself (template literal function).
- `globals` : Global variables to preset.

appendGlobals

```
appendGlobals(globals: object): SExpressionTemplateFn;
```

- returns : myself (template literal function).
- `globals` : Global variables to preset.

setStartup

```
setStartup(strings: TemplateStringsArray | string, ...values: any[]): SExpressionTemplat
```

- returns : myself (template literal function).
- `strings` : Startup code that evaluate before each evaluation of user code.

setStartupAST

```
setStartupAST(ast: SxToken[]): SExpressionTemplateFn;
```

- returns : myself (template literal function).
- `ast` : Startup code AST that evaluate before each evaluation of user code.

appendStartup

```
appendStartup(strings: TemplateStringsArray | string, ...values: any[]): SExpressionTemp
```

- returns : myself (template literal function).
- `strings` : Startup code that evaluate before each evaluation of user code.

appendStartupAST

```
appendStartupAST(ast: SxToken[]): SExpressionTemplateFn;
```

- returns : myself (template literal function).
- `ast` : Startup code AST that evaluate before each evaluation of user code.

install

```
install(installer: (config: SxParserConfig) => SxParserConfig): SExpressionTemplateFn;
```

- returns : myself (template literal function).
- `installer` : Installer function that register the operators, macros, constants to the `config` object.

`runScriptTags`

Run script tags.

```
function runScriptTags(
    lisp: SExpressionTemplateFn | SExpressionAsyncTemplateFn,
    globals?: object,
    contentType = 'text/lisp')
```

- returns : Evaluation result.
- `lisp` : Evaluater function.
- `globals` : Global variables.
- `contentType` : Content type attribute of script tags.

Usage:

```
<!DOCTYPE html>
<head>
    <meta charset="utf-8">
    <script type="text/lisp">
        ($local ((body (::document@querySelector "body")))
            ($set (body innerText) "Hello, Lisp! ") )
        ($local (c) ($capture (c)
            ($$defun tarai(x y z)
                ($set c (+ c 1))
                ($if (<= x y)
                    y
                    ($self ($self (- x 1) y z)
                        ($self (- y 1) z x)
                        ($self (- z 1) x y))))
        ($list ($datetime-to-iso-string ($now)) (tarai 13 6 0) c) ))
    </script>
    <script src="liyad.min.js"></script>
    <script>
        // Since the above lisp code refers to the body element,
        // you need to enclose the lisp evaluation with addEventListener.
        document.addEventListener('DOMContentLoaded', function(event) {
            const result = JSON.stringify(
                liyad.runScriptTags(liyad.lisp, {window, document}));
            const body = document.querySelector('body');
```

```
                setTimeout(() => body.innerText = body.innerText + result, 30);
            });
        </script>
    </head>
    <body></body>
```

Operators

See core, arithmetic, sequence, concurrent, JSX (LSX) operators.

License