```
"declarations": [
  {
    "type": "VariableDeclarator",
    "id": {
      "type": "Identifier",
      "name": "foo"
    },
    "init": {
      "type": "CallExpression",
      "callee": {
        "type": "Identifier",
        "name": "require"
      },
      "arguments": [
        {
          "type": "Literal",
          "foo",
```

Matching patterns in syntax trees

While developing Lebab (https://lebab.io/), major part of the work is detecting patterns in source code. After having detected the part of Abstract Syntax Tree (AST) that interests us, it's usually fairly simple to apply a transformation on it.

Say, we'd like to transform CommonJS `require()` calls to ES6 import syntax. Given the following code:

```
var foo = require('foo');
```

Esprima (http://esprima.org/) produces the following AST:

```
{
    "type": "VariableDeclaration",
    "declarations": [
        {
            "type": "VariableDeclarator",
            "id": {
                "type": "Identifier",
                "name": "foo"
            },
            "init": {
                "type": "CallExpression",
                "callee": {
                    "type": "Identifier",
                    "name": "require"
                },
                "arguments": [
                    {
                        "type": "Literal",
                        "value": "foo",
                        "raw": "'foo'"
                    }
                ]
            }
        }
    ],
    "kind": "var"
}
```

To detect this `require()` pattern, we used to write code like this:

```
function isRequire(node) {
    return node.type === 'VariableDeclaration' &&
        node.kind === 'var' &&
        node.declarations.length === 1 &&
        node.declarations[0].id.type === 'Identifier' &&
        node.declarations[0].init &&
        node.declarations[0].init.type === 'CallExpression' &&
        node.declarations[0].init.callee.type === 'Identifier' &&
        node.declarations[0].init.callee.name === 'require' &&
        node.declarations[0].init.arguments.length === 1 &&
        node.declarations[0].init.arguments[0].type === 'Literal' &&
        typeof node.declarations[0].init.arguments[0].value === 'string';
}
```

That's straightforward to write, but damn hard to read. Even while looking at the AST above, it's hard to tell whether it's matching it correctly.

Pattern matching to the rescue

What if we could just use the AST that Esprima produces as a pattern to match against? Then it would all come down to a fairly simple pattern-matching operation.

It shouldn't be hard to write a function that does a deep comparison of two objects. Actually we don't even have to write one by ourselves. The popular Lodash (https://lodash.com/) library provides a function `_.matches` (https://lodash.com/docs#matches) that does exactly this.

We can give it our AST pattern and it creates a function that matches against it:

```
const isRequire = _.matches({
    "type": "VariableDeclaration",
    "declarations": [
        {
            "type": "VariableDeclarator",
            "id": {
                "type": "Identifier",
                // "name": <any value>
            },
            "init": {
                "type": "CallExpression",
                "callee": {
                    "type": "Identifier",
                    "name": "require"
                },
                "arguments": [
                    {
                        "type": "Literal",
                        // "value": <any string>
                    }
                ]
            }
        }
    ],
    "kind": "var"
});
```

With all the conditional logic gone, it's now straightforward to understand what this function matches.

Pattern matching extended

However `_.matches()` is too limited for our needs. We'd also like to check that `require()` takes just one string argument, and that's not achievable by plain pattern matching.

What if we could write custom assertions inside our AST pattern:

```
{
    "type": "Literal",
    "value": (v) => typeof v === "string"
}
```

Then we could extend the pattern matching with conditional logic where needed.

Turns out, that's really easy to achieve as Lodash provides `_.matches()` extension mechanism. We can use `_.isMatchWith()` (https://lodash.com/docs#isMatchWith) to add custom comparison logic. Here's our custom `matches()` function:

```
function matches(pattern) {
    return (ast) => {
        return _.isMatchWith(ast, pattern, (value, matcher) => {
            // When comparing against function, execute the function
            if (typeof matcher === 'function') {
                return matcher(value);
            }
            // Otherwise fall back to built-in comparison logic
        });
    };
}
```

Not only can we now write simple assertions like the one above, but we can also compose it with further call to `matches()` :

```
const isRequire = matches({
    "type": "VariableDeclaration",
    "declarations": (decs) => decs.length === 1 && matches([
        {
            "type": "VariableDeclarator",
            "id": {
                "type": "Identifier",
                // "name": <any value>
            },
            "init": {
                "type": "CallExpression",
                "callee": {
                    "type": "Identifier",
                    "name": "require"
                },
                "arguments": (args) => args.length === 1 && matches([
                    {
                        "type": "Literal",
                        "value": (v) => typeof v === 'string'
                    }
                ])(args)
            }
        }
    ])(decs),
    "kind": "var"
});
```

Depending your inclination, you might consider the above code really neat or really horrible. But we can alternatively extract it to several functions:

```
const isStringLiteral = matches({
    "type": "Literal",
    "value": (v) => typeof v === 'string'
});

const isRequireDeclarator = matches({
    "type": "VariableDeclarator",
    "id": {
        "type": "Identifier",
        // "name": <any value>
    },
    "init": {
        "type": "CallExpression",
        "callee": {
            "type": "Identifier",
            "name": "require"
        },
        "arguments": (args) => args.length === 1 && isStringLiteral(args[0])
    }
});

const isRequire = matches({
    "type": "VariableDeclaration",
    "declarations": (decs) => decs.length === 1 && isRequireDeclarator(decs[0]
    "kind": "var"
});
```

Extracting data

With `isRequire()` implemented, we can easily detect the pattern and return new AST for ES6 import declaration:

```
if (isRequire(node)) {
    return {
        "type": "ImportDeclaration",
        "specifiers": [
            {
                "type": "ImportDefaultSpecifier",
                "local": node.declarations[0].id
            }
        ],
        "source": node.declarations[0].init.arguments[0]
    }
}
```

But wait… These `node.declarations[0].init.arguments[0]` expressions are just like the ones we tried to eliminate. It's not quite as bad as the initial matcher function, but we're partly repeating the AST that we already specified in the matcher function.

What if we could simply label the things we need to extract within our matcher function:

```
const matchRequire = matches({
    "type": "VariableDeclarator",
    "id": {
        "type": "Identifier",
        "name": extract("name")
    },
    "init": extract("init")
});
```

and instead of returning true, our matcher would return us an object with extracted values:

```
const {name, init} = matchRequire(node);
```

Turns out, it's also fairly easy to implement. Our `extract()` will simply produce a function that creates and object with a single match:

```
function extract(fieldName) {
    return (ast) => ({[fieldName]: ast});
}
```

Within `matches()` we then combine all these singular matches into a single object:

```
function matches(pattern) {
    return (ast) => {
        const extractedFields = {};

        const matches = _.isMatchWith(ast, pattern, (value, matcher) => {
            // When comparing against function, execute the function
            if (typeof matcher === 'function') {
                const result = matcher(value);
                // When extracted fields returned,
                // combine them with other extracted fields
                if (typeof result === 'object') {
                    Object.assign(extractedFields, result);
                }
                return result;
            }
            // Otherwise fall back to built-in comparison logic
        });

        return matches ? extractedFields : false;
    };
}
```

Extracting extended

The nice thing about this implementation is that `extract()` can be composed with further conditional logic, even with additional `matches()`:

```
const isRequireDeclarator = matches({
    "type": "VariableDeclarator",
    "id": (id) => matches({
        "type": "Identifier",
    })(id) && extract("local")(id),
});
```

The above pattern is common enough to improve our `extract()` function with an optional second parameter to specify the nested pattern:

```
const isRequireDeclarator = matches({
    "type": "VariableDeclarator",
    "id": extract("local", {
        "type": "Identifier",
    }),
});
```

The implementation of this is somewhat similar to `matches()` . This time we'll need to combine extracted fields from nested matches:

```
function extract(fieldName, matcher) {
  return (ast) => {
    const extractedFields = {[fieldName]: ast};

    // Convert plain pattern into matcher function
    if (typeof matcher === 'object') {
        matcher = matches(matcher);
    }

    if (typeof matcher === 'function') {
        const result = matcher(ast);
        // When the nested match also contains extracted fields,
        // combine them with current field
        if (typeof result === 'object') {
            return Object.assign(extractedFields, result);
        }
        // When the nested match failed, the whole matching failed.
        if (!result) {
            return false;
        }
    }

    return extractedFields;
  };
}
```

In the end the full implementation of our import transform reads like so:

```
const isStringLiteral = extract("local", {
    "type": "Literal",
    "value": (v) => typeof v === 'string'
});

const isRequireDeclarator = matches({
    "type": "VariableDeclarator",
    "id": extract("local", {
        "type": "Identifier",
    }),
    "init": {
        "type": "CallExpression",
        "callee": {
            "type": "Identifier",
            "name": "require"
        },
        "arguments": (args) => args.length === 1 && isStringLiteral(args[0])
    }
});

const isRequire = matches({
    "type": "VariableDeclaration",
    "declarations": (decs) => decs.length === 1 && isRequireDeclarator(decs[0]
    "kind": "var"
});

function createImportDeclaration({local, source}) {
    return {
        "type": "ImportDeclaration",
        "specifiers": [
            {
                "type": "ImportDefaultSpecifier",
                "local": local
            }
        ],
        "source": source
    }
}

export default function doTransform(ast) {
    estraverse.replace(ast, {
        enter(node) {
            const match = matchRequire(node);
            if (match) {
                return createImportDeclaration(match);
            }
        }
    });
}
```

Happy days.

With this knowledge under your belt, go implement your favorite ES5-to-ES6 transform and send pull request to Lebab (https://github.com/mohebifar/lebab).

Rene Saarsoo

Väike alatu progeja.

Twitter (http://twitter.com/renku)

Facebook (http://facebook.com/rene.saarsoo)

Github (http://github.com/nene)

Matching patterns in syntax trees on kirjutatud April 02, 2016 ja viimati muudetud April 02, 2016. Autor: Rene Saarsoo (http://nene.github.io/about).

SIND VÕIB LISAKS HUVITADA

(KÕIK POSTITUSED (HTTP://NENE.GITHUB.IO/KIRJUTISED))

- Why the ZWO format sucks (http://nene.github.io/2021/01/14/zwo-sucks)
- Code review checklist (http://nene.github.io/2016/01/21/code-review)
- Facebooki häkk (http://nene.github.io/2014/04/03/Facebooki-hakk)
