Abstract Syntax Tree

Beyond Theory

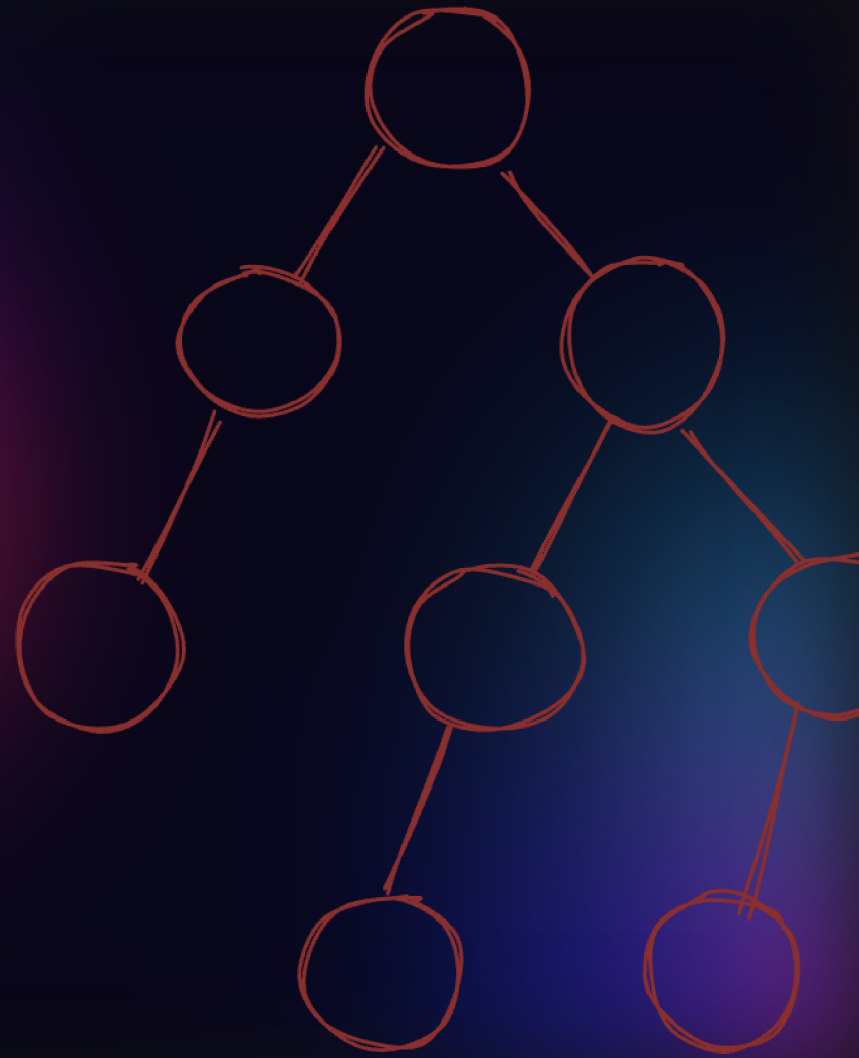
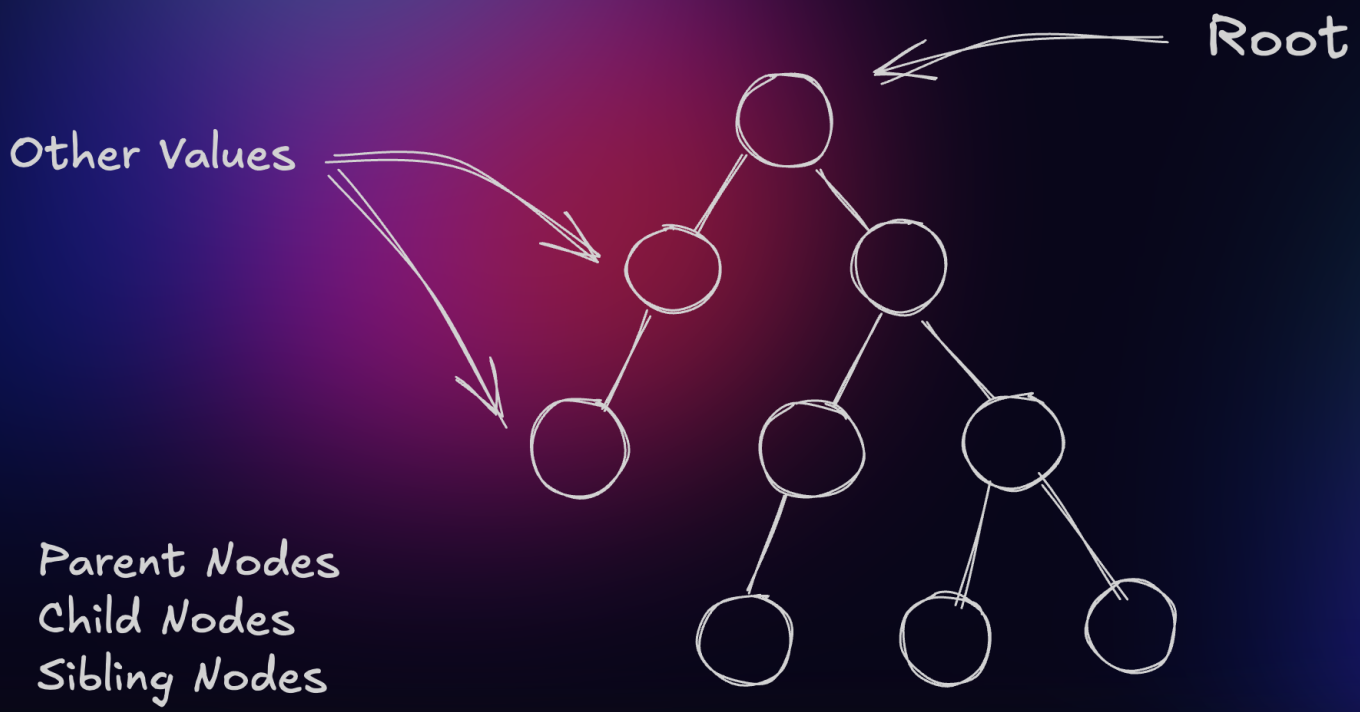


Table of Contents

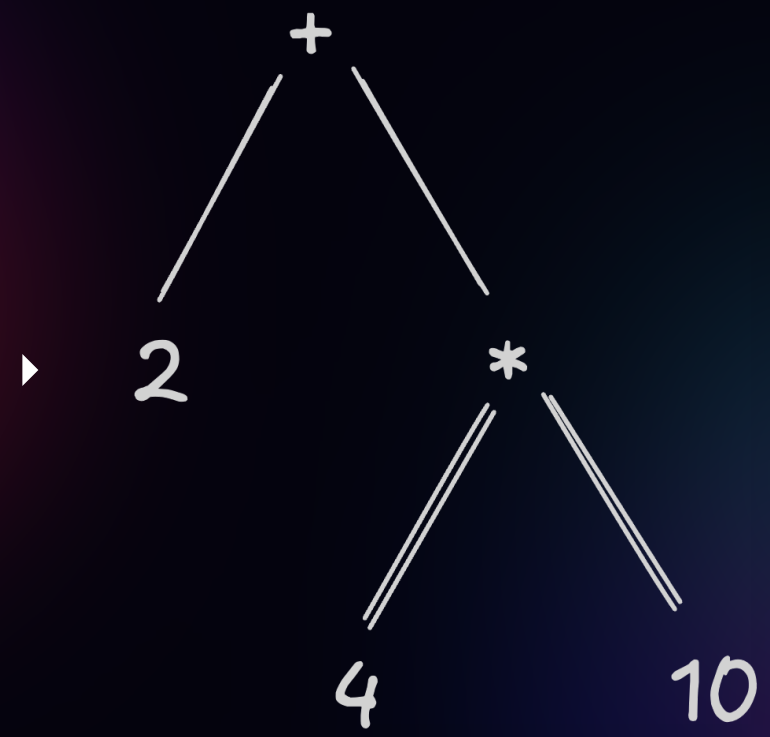
What is AST?

A tree representation of the syntactic structure of sourcecode



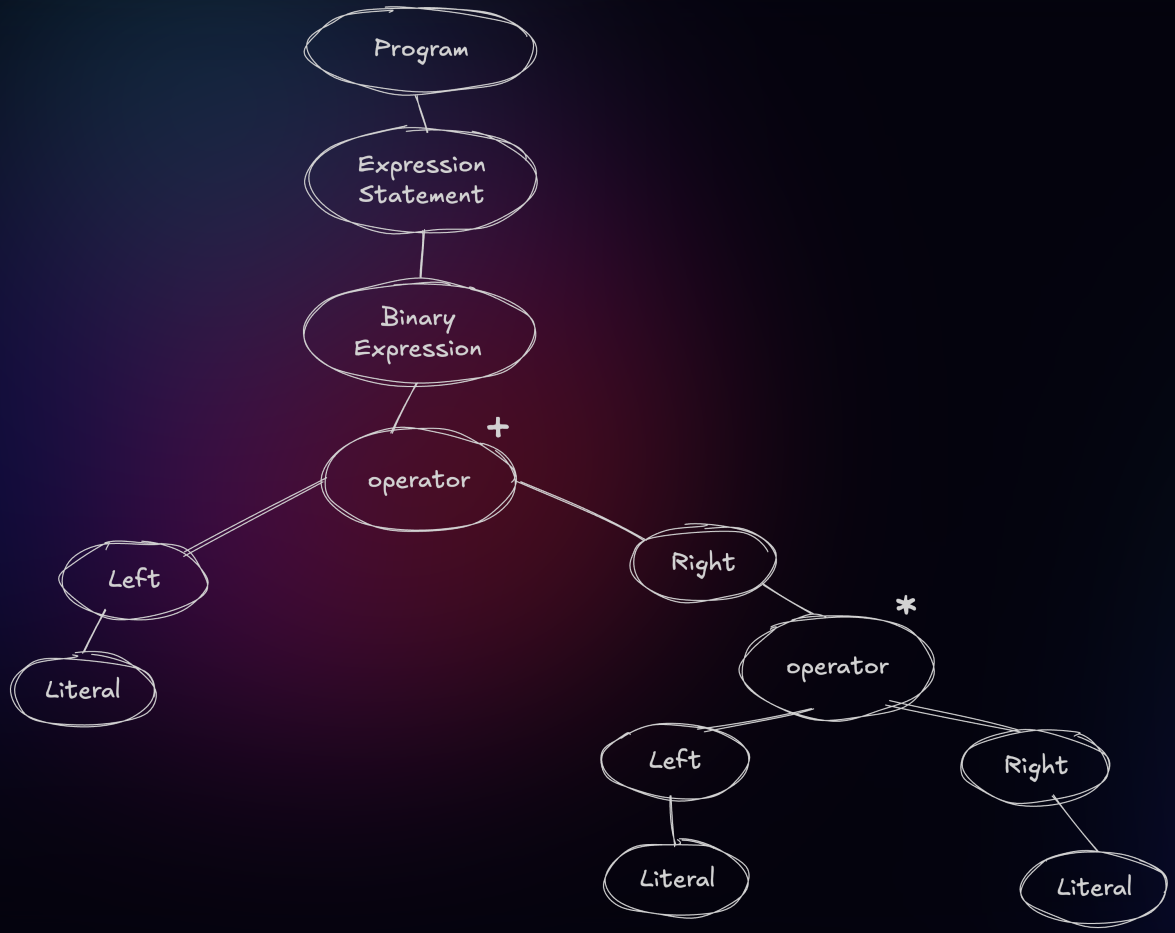
Quick Example

2 + (4 * 10)



```
2 + ( 4 * 10 )
```

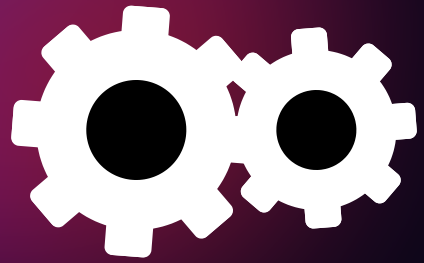
```
{
  "type": "BinaryExpression",
  "left": {
    "type": "NumericLiteral",
    "value": 2
  }
}
```


Program
Expression Statement
Binary Expression
operator
+
Left
Literal
Right
operator
*
Left
Literal
Right
Literal

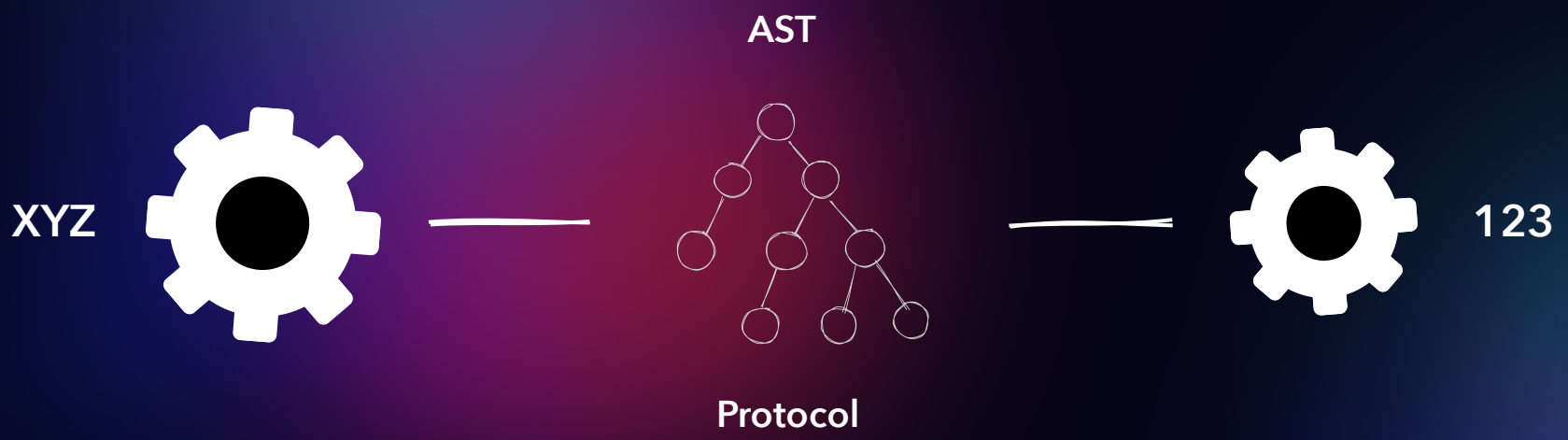
Why AST Matters?

Compilers

High Level Languages



Low Level Languages


AST
XYZ
123
Protocol

Linting

Formatting

Transpiling

JSCODESHIFT

Transforming

Exploring AST

How does code get transformed to AST?

// Code: let x = 42;

LEXICAL ANALYSIS

```
{ type: 'keyword', value: 'let' },
{ type: 'identifier', value: 'x' },
{ type: 'operator', value: '=' },
{ type: 'number', value: '42' },
{ type: 'punctuation', value: ';' }
```

SYNTACTIC ANALYSIS

Converts tokens into AST
Validates syntax
Creates tree structure

GENERATE AST


```
{
   "type":"BinaryExpression",
   "left":{
      "type":"Literal",
      "value":2,
      "raw":"2"
   },
}
```

Common Node Types

```
// Literal Nodes
"hello" -> { type: 'StringLiteral', value: 'hello' }
42 -> { type: 'NumericLiteral', value: 42 }
```

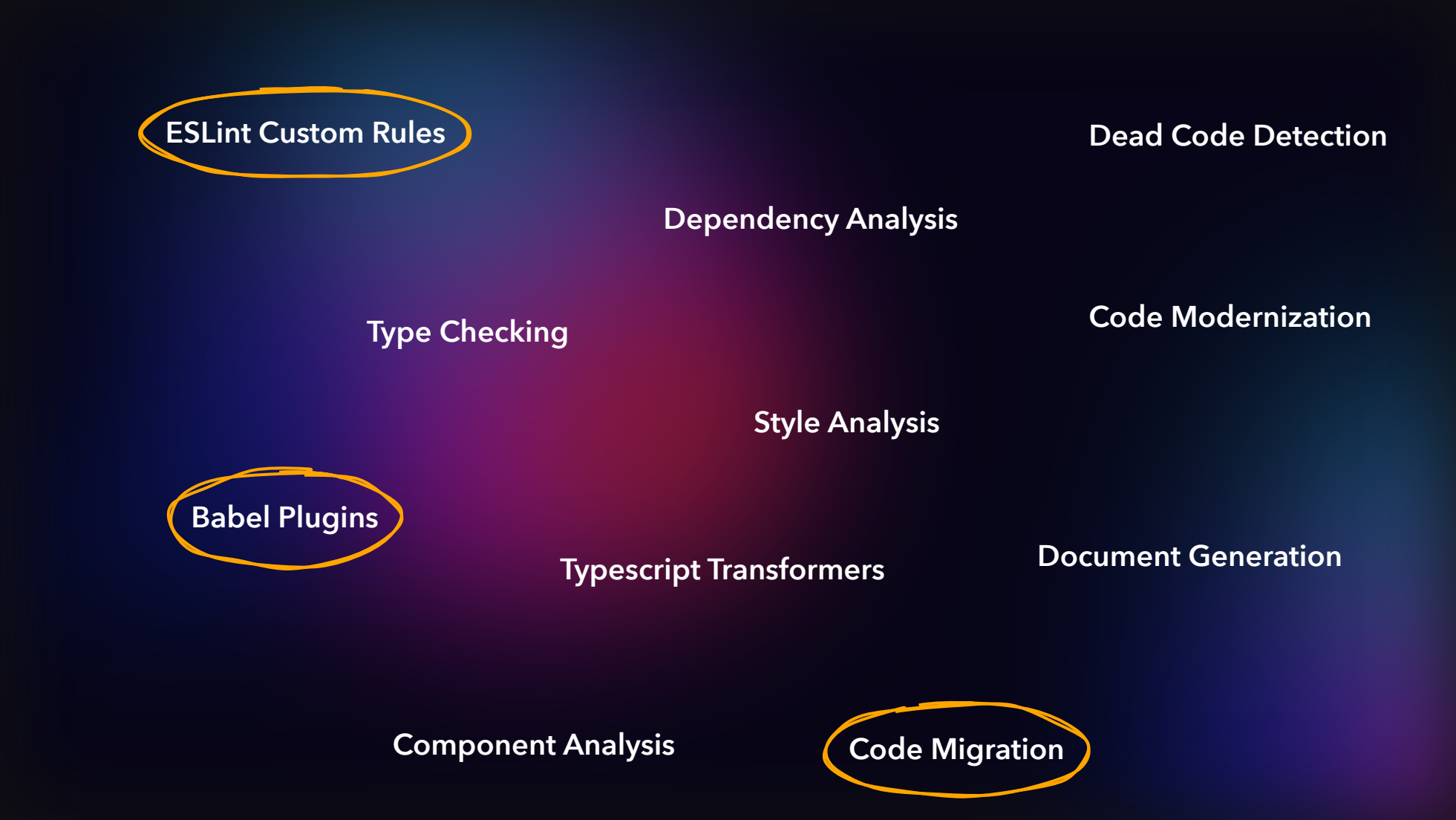
Tools to Explore AST

astexplorer.net

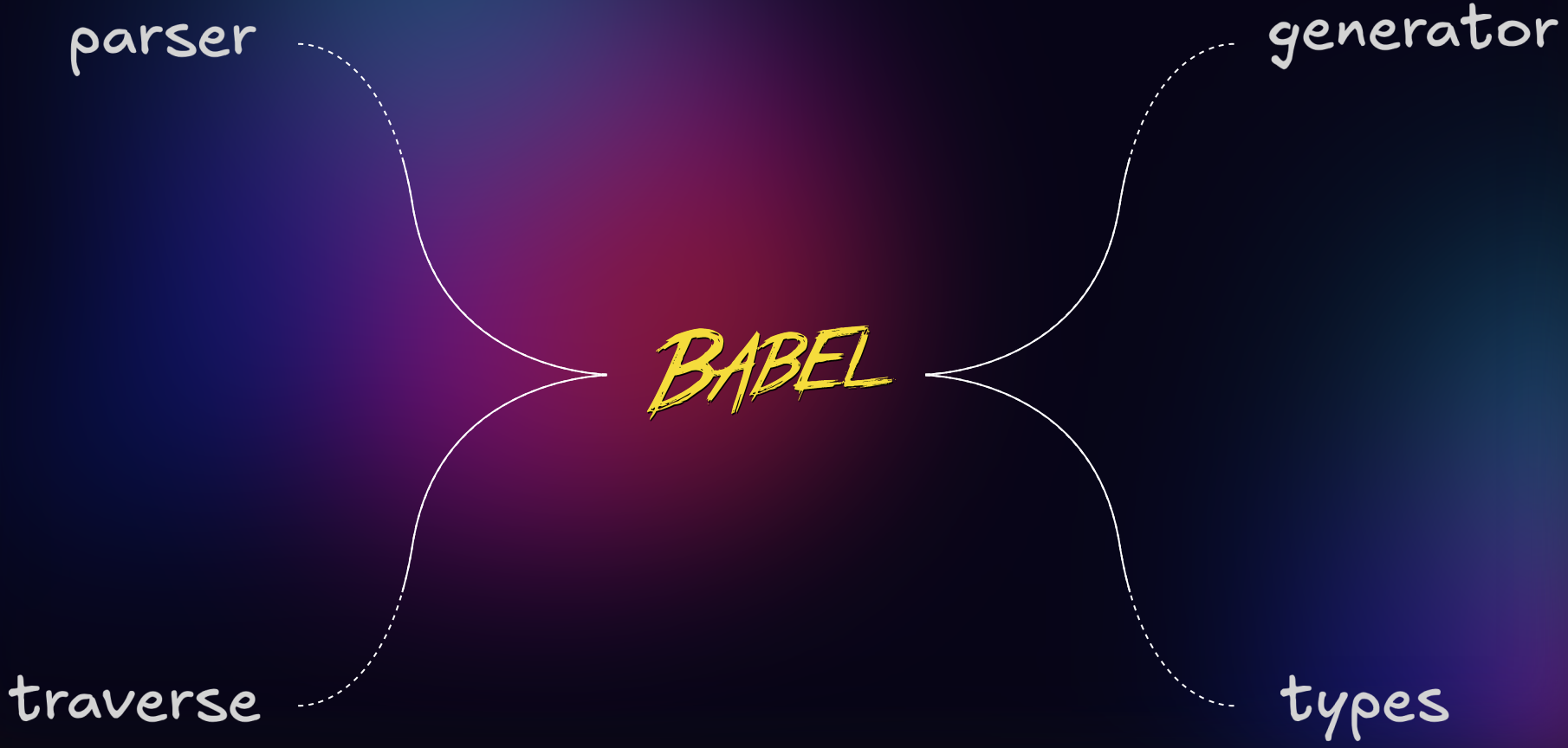
typescript ast viewer

github.com/estree

AST in Practice


ESLint Custom Rules
Dead Code Detection
Dependency Analysis
Type Checking
Code Modernization
Style Analysis
Babel Plugins
Typescript Transformers
Document Generation
Component Analysis
Code Migration

Working with javascript AST



@babel/parser

```
1  import fs from 'fs';
2  import babelParser from '@babel/parser';
```

@babel/traverse

```
import babelTraverse from '@babel/traverse';
const { default: traverse } = babelTraverse;
```

@babel/types

```
1  import * as t from '@babel/types';
```

@babel/generator

```
1  import babelGenerator from '@babel/generator';
2  const { default: generate } = babelGenerator;
```

Case Study

Styled Components Migration & Post-Transformation

Preface

How do we handle RWD ?

```
const GENERAL_WIDTH = 375;

// landing會先換算，讓 1rem = 1vw
export const px2Unit = (px) => {
  const unit = (px / GENERAL_WIDTH) * 100;
  return `${unit}rem`;
}
```

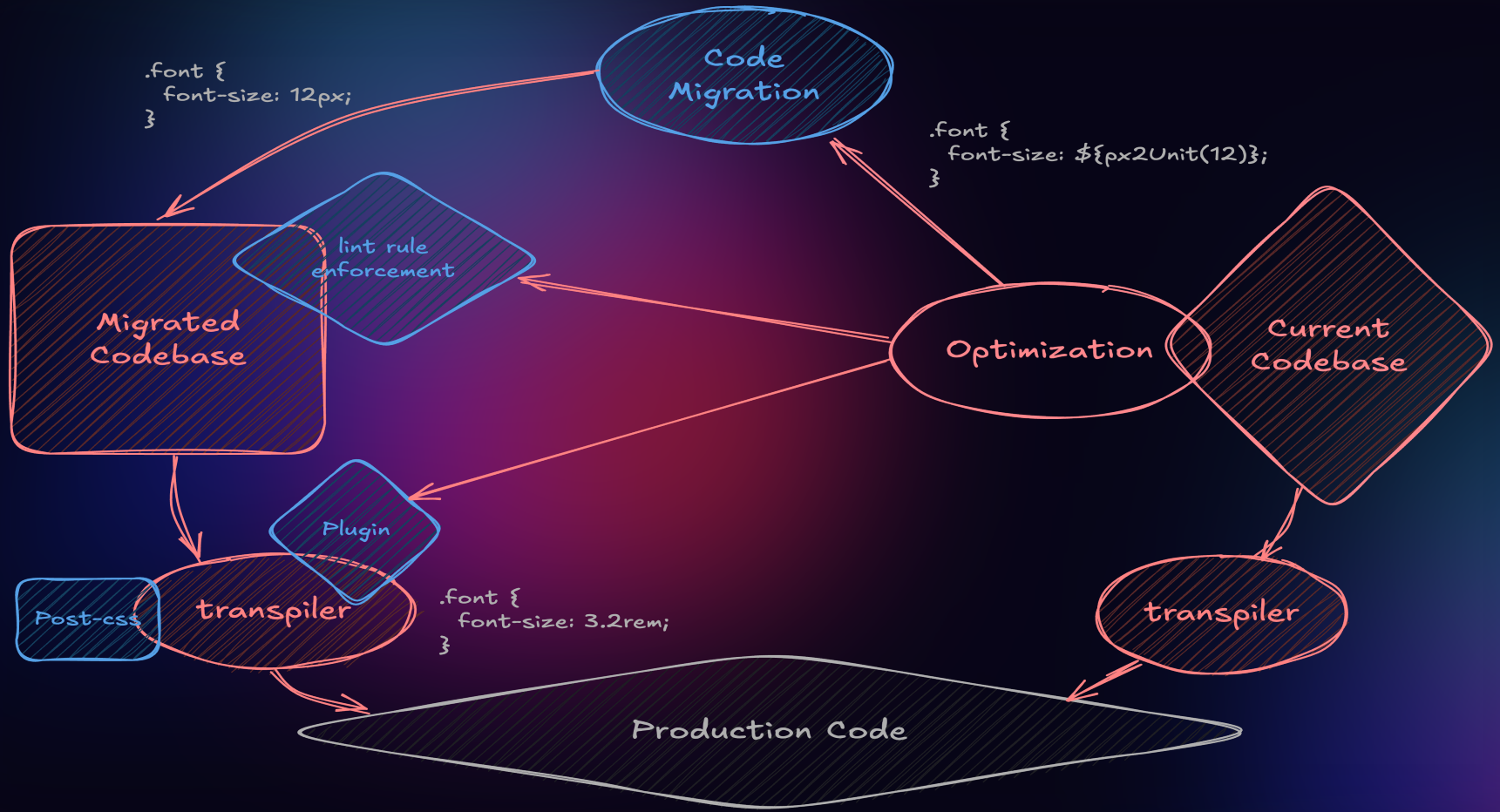
```
import styled from 'styled-components';
import { px2Unit } from './utils';

const StyledComponent = styled.div`
  width: ${px2Unit(100)};
  height: ${px2Unit(100)};

  text-shadow:
    ${px2Unit(-1)} ${px2Unit(-1)} 0 #fff,
    ${px2Unit(1)} ${px2Unit(-1)} 0 #fff,
    ${px2Unit(-1)} ${px2Unit(1)} 0 #fff,
    ${px2Unit(1)} ${px2Unit(1)} 0 #fff;
`;
```

Why is this Optimizable?

1. Runtime Performance Issue
   - Cost might be high due to excessive and repeated calculation
2. Bad Developer Experience
   - Manually add px2Unit everytime can be verbose and time-consuming
3. Maintenance Challenges
   - Allowing scattered call to px2Unit and can be hard to maintain

Solution

1. Runtime to Buildtime transformation
2. Current codebase migration
3. Future Usage Enforcement


.font {
font-size: 12px;
}
Code Migration
.font {
font-size: ${px2Unit(12)};
}
lint rule enforcement
Migrated Codebase
Optimization
Current Codebase
Plugin
Post-css
transpiler
.font {
font-size: 3.2rem;
}
transpiler
Production Code

Enough talk–let's dive into it

AST DEMO | SWC_PLUGIN

How do we do the migration?

1. Identify all styled-components.
2. Find all px2Unit call inside styled-components.
3. Remove all px2Unit call inside styled-components
4. Find all JSX expression container that contains px2Unit call
5. Remove all px2Unit call inside JSX expression container
6. Remove unused px2Unit imports

Identifying All Styled-Components

How Many valid ways to declare a styled-components?

```
1  // Basic Tagged Template Literal
```

```
1  // styled object method + Tagged Template Literal
```

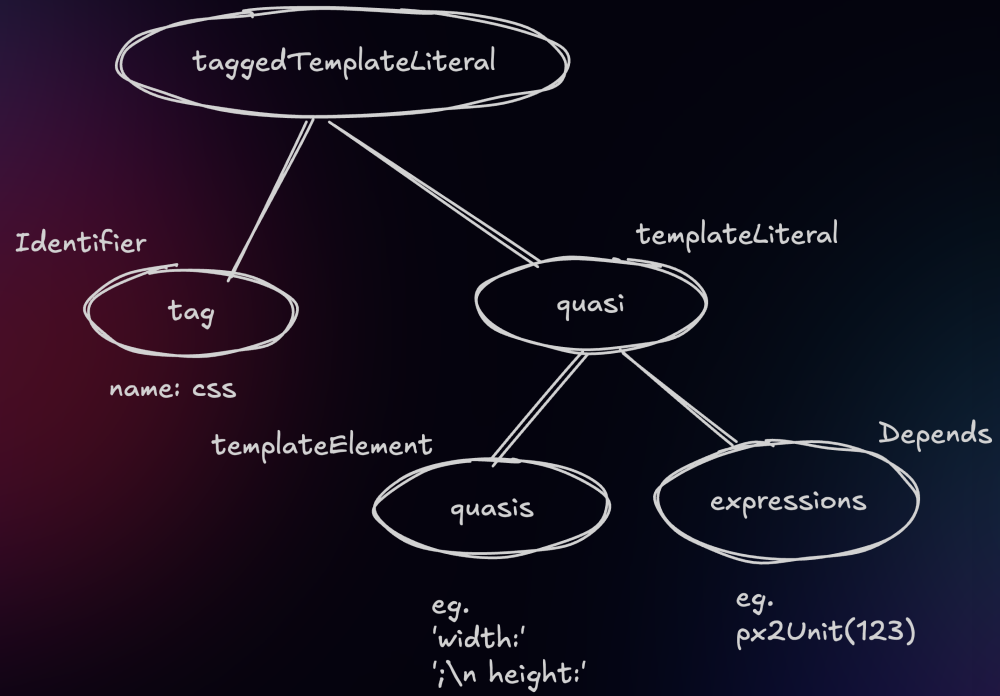
```
1  // styled function call + Tagged Template Literal
```

```
1  // styled object method function call
```

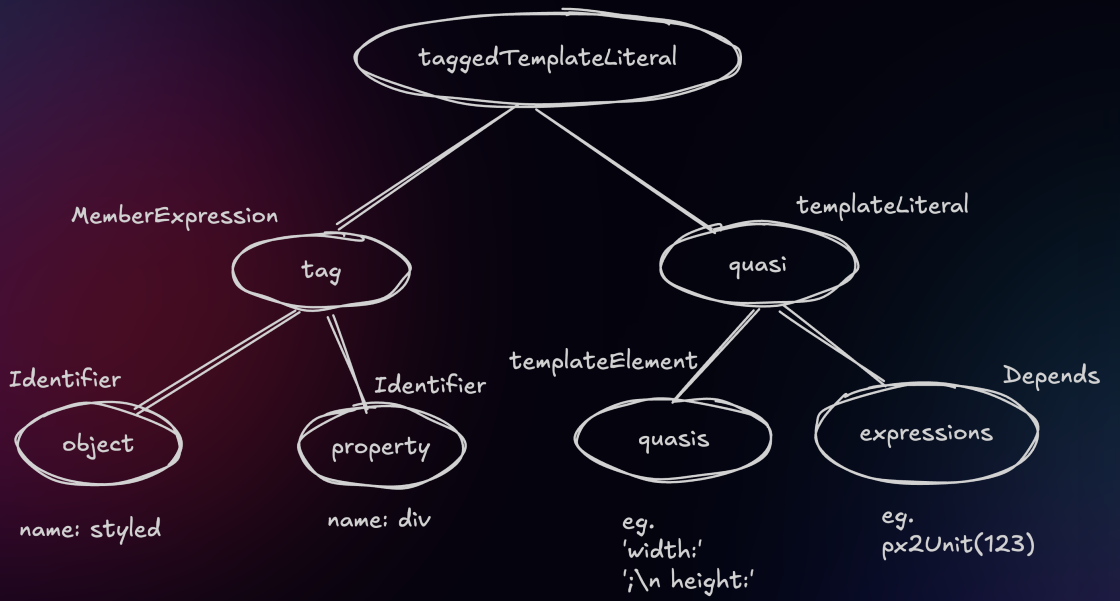
```
1  // attributes
```

```
1  // attributes function call
```

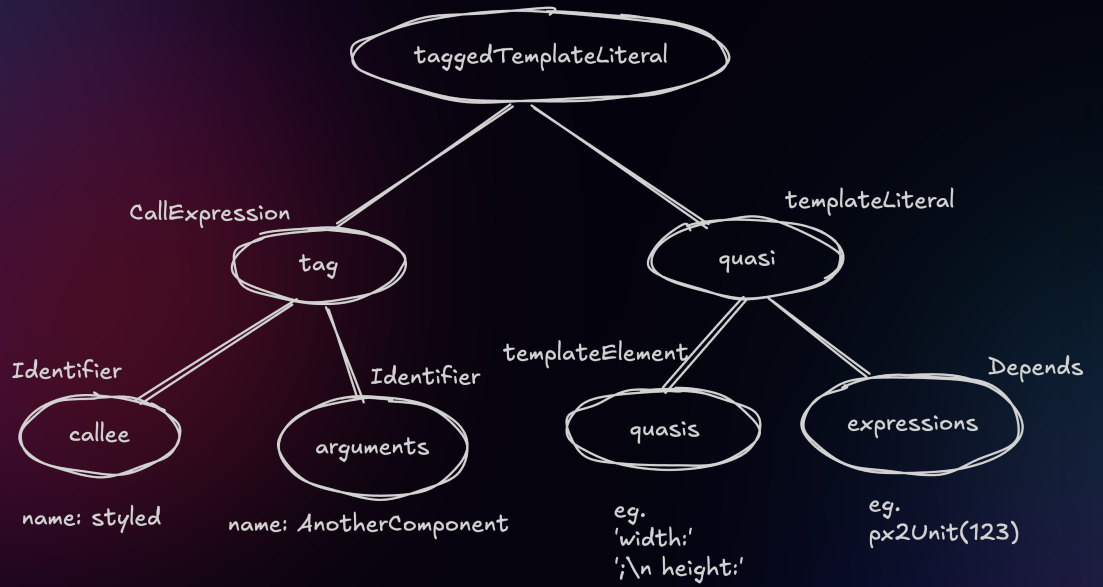
```
// Basic Tagged Template Literal
const Component = css``;
```



```
// styled object method
const Component = styled.div``;
```



```
// styled function call
const Component = styled(AnotherComponent)``;
```



```
// styled object method function call
const Component = styled.div.attrs()``;
```

```
1  import { isIdentifier, isMemberExpression, isCallExpression } from '@babel/types';
```

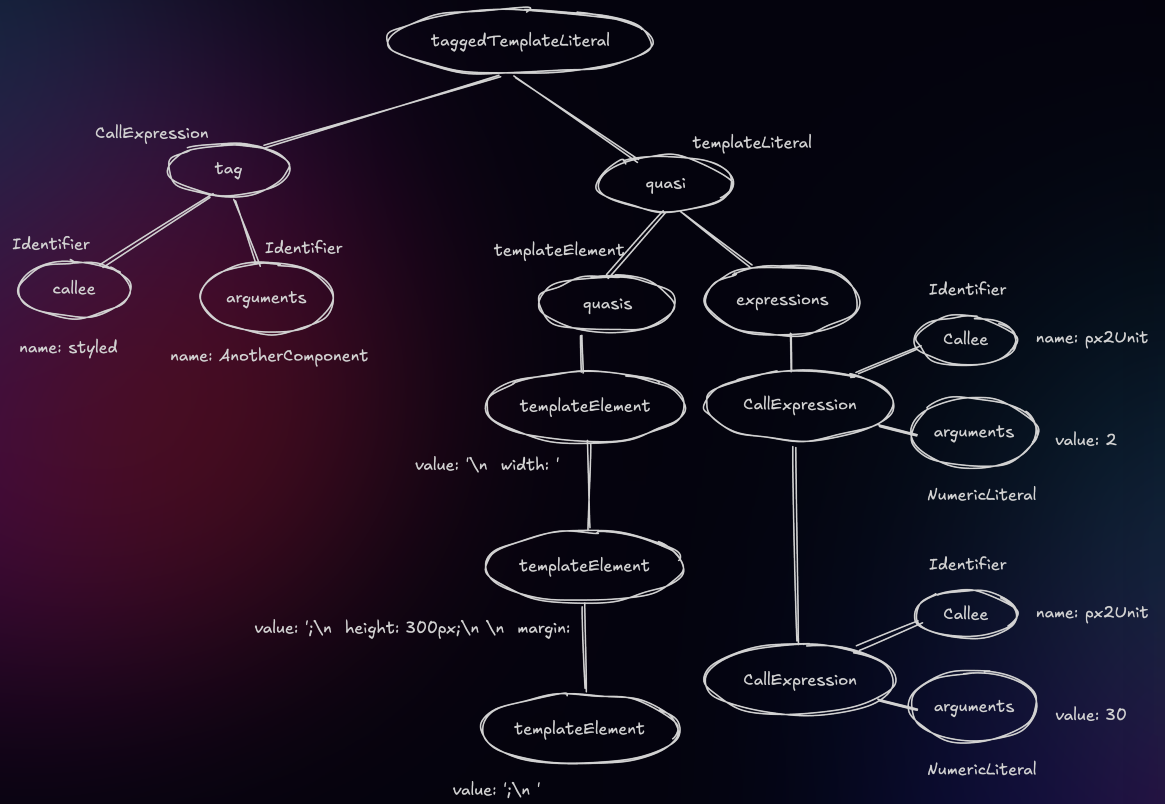
Finding all px2Unit call inside styled-components

How Many ways to use px2Unit inside styled-components?

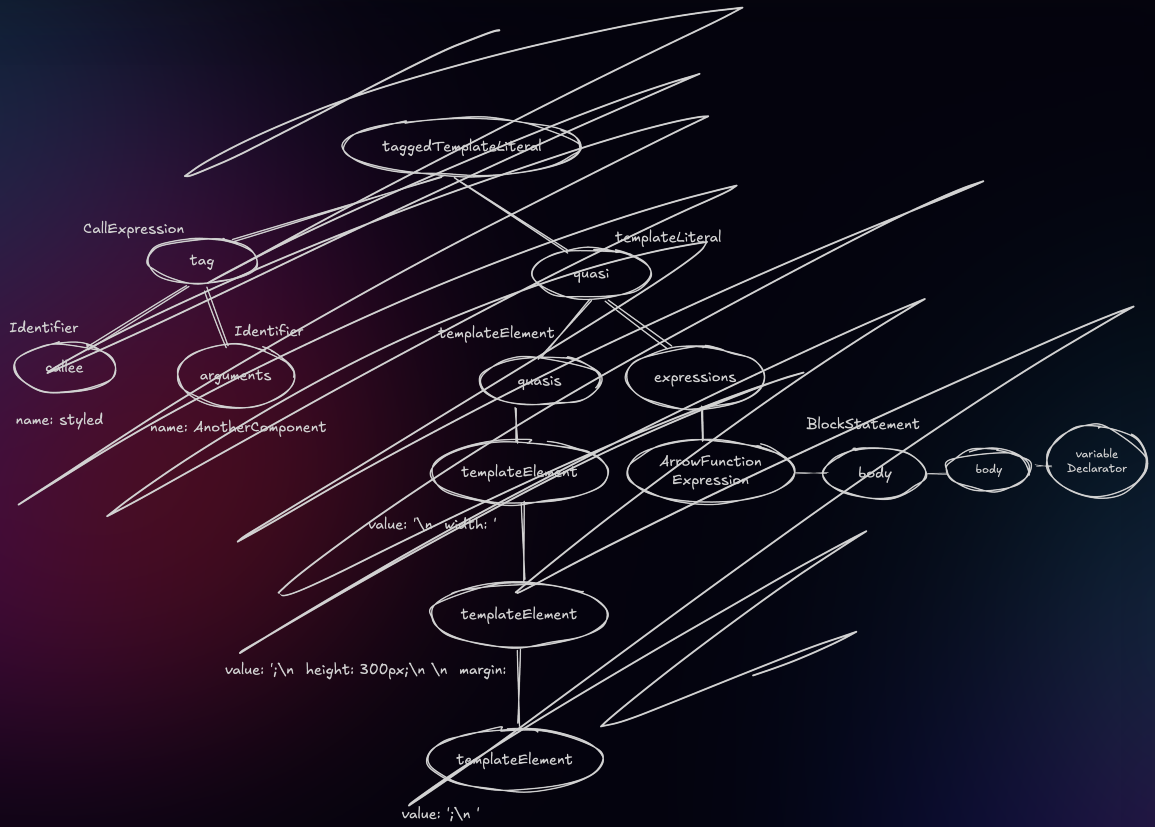
```
1  // Direct call
```

```
1  // Anything except direct call
```

```
// TaggedTemplateLiteral
export default styled(AnotherComponent)`
  width: ${px2Unit(2)};
  height: 300px;
  margin: ${px2Unit(30)};
`
```



```
// TaggedTemplateLiteral
export default styled(AnotherComponent)`
  width: ${() => {
    const width = px2Unit(2)
    return width
  }};
  height: 300px;
  margin: ${px2Unit(30)};
`
```



```
{
  TaggedTemplateExpression: () => {}
}
```

Removing all px2Unit call from styled-components

```
1  const transformDirectCall = () => {}
```

```
1  const transformDynamicCall = () => {}
```

```
import { isCallExpression } from '@babel/types';
const isPx2UnitCall = (expression) =>
  expression.callee?.name === "px2Unit";

{
  TaggedTemplateExpression: (path) => {
    if (!isStyledTag(path)) return;
    const templateLiteralPath = path.get("quasi");
    const { quasis, expressions } = templateLiteralPath.node;

    expressions.forEach((expressionPath, i) => {
      if (isCallExpression(expressionPath) && isPx2UnitCall(expressionPath)) {
        // Replace the expression with px2Unit value
      } else {
        expressionPath.traverse({
          CallExpression: (callExpressionPath) => {
            if (isPx2UnitCall(callExpressionPath)) {
              // Replace the expression with px2Unit value
            }
          }
        })
      }
    })
  }
}
```

Finding All JSX expression container that contains px2Unit call

```
{
  JSXExpressionContainer: () => {}
}
```

Remove all JSX expression container that contains px2Unit call

```
const transformJSXContainer = (jsxContainerPath, arg) => {
  // Handle pure literals cases: eg. string or numeric literals
  // transform  width={px2Unit(10)} to width="10px"
  if (isPureLiteral(arg)) {
    jsxContainerPath.replaceWith(stringLiteral(`${arg.value}px`));
    return;
  }
};
```

```
const transformTemplate = (templateLiteralPath) => {
  const { quasis, expressions } = templateLiteralPath.node;

  expressions.forEach((expr, i) => {
    const exprPath = templateLiteralPath.get(`expressions.${i}`);
    const quasiPath = templateLiteralPath.get(`quasis.${i + 1}`);

    if (isCallExpression(expr) && isPx2UnitCall(expr)) {
      transformDirectPx2UnitCall(exprPath, quasiPath, quasis[i], quasi);
    } else {
      transformDynamicCall(exprPath);
    }
  })
};
```

```
const transformDynamicCall = (exprPath) => {
  const getReplaceNode = (arg) =>
    isNumericLiteral(arg)
      ? stringLiteral(`${arg.value}px`)
      : createTemplateLiteral(arg);

  exprPath.traverse({
    CallExpression: (callExprPath) => {
      if (!isPx2UnitCall(callExprPath.node)) return;

      const [arg] = callExprPath.node.arguments;
      // Replace px2Unit call with either:
      // 1. A string literal (e.g. "10px")
      // 2. Dynamic values (e.g. `${width}px`)
      callExprPath.replaceWith(getReplaceNode(arg));
    },
  });
}
```

```
import { isNumericLiteral, isStringLiteral, stringLiteral } from '@babel/types';
const isPureLiteral = (expression) => isNumericLiteral(expression) || isStringLiteral(expression);
const isPx2UnitCall = (expression) => expression.callee?.name === "px2Unit";

{
  JSXExpressionContainer: (jsxExpressionPath) => {
    jsxExpressionPath.traverse({
      CallExpression: (callExpressionPath) => {
        if (isPx2UnitCall(jsxContainerPath.node.expression)) {
          // handle direct px2Unit call eg. width={px2Unit(10)}
        } else if (isTemplateLiteral(jsxContainerPath.node.expression)) {
          // handle px2Unit call inside template literal eg. margin={`${px2Unit(10)} ${px2Unit(20)}`}
        } else {
          // handle all other complex expressions
        }
      }
    })
  }
}
```

CONCLUSION

AST can be intimidating and tricky as you first sees it,
but it can be extemely powerful if you know how to harness its power.

Thanks!

Q&A