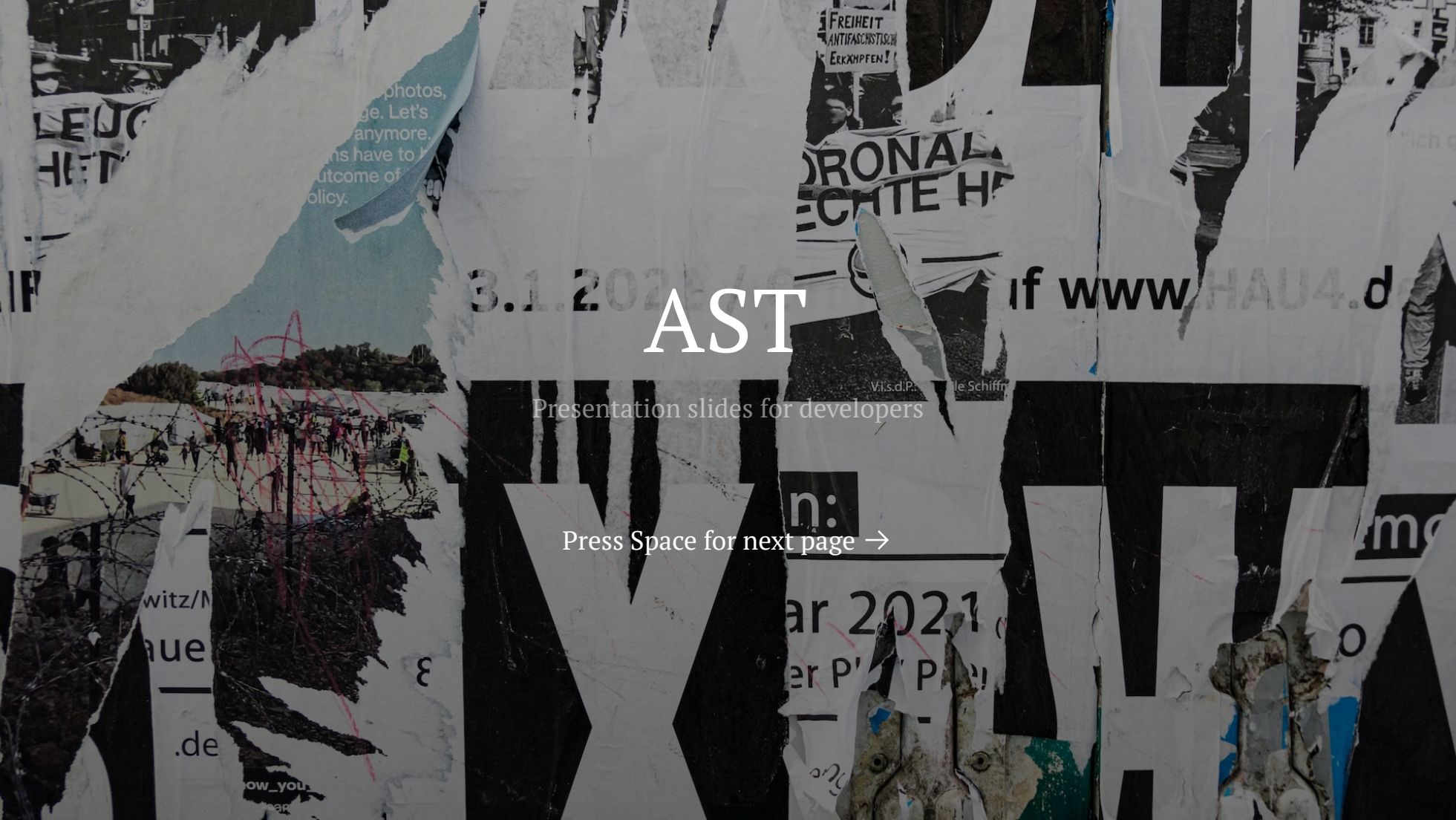


AST

Presentation slides for developers

Press Space for next page →

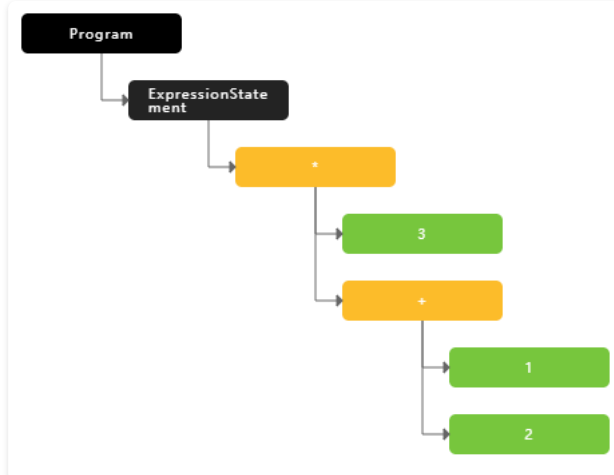
目录

简介

抽象语法树（Abstract Syntax Tree，AST）是源代码语法结构的一种抽象表示，它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。

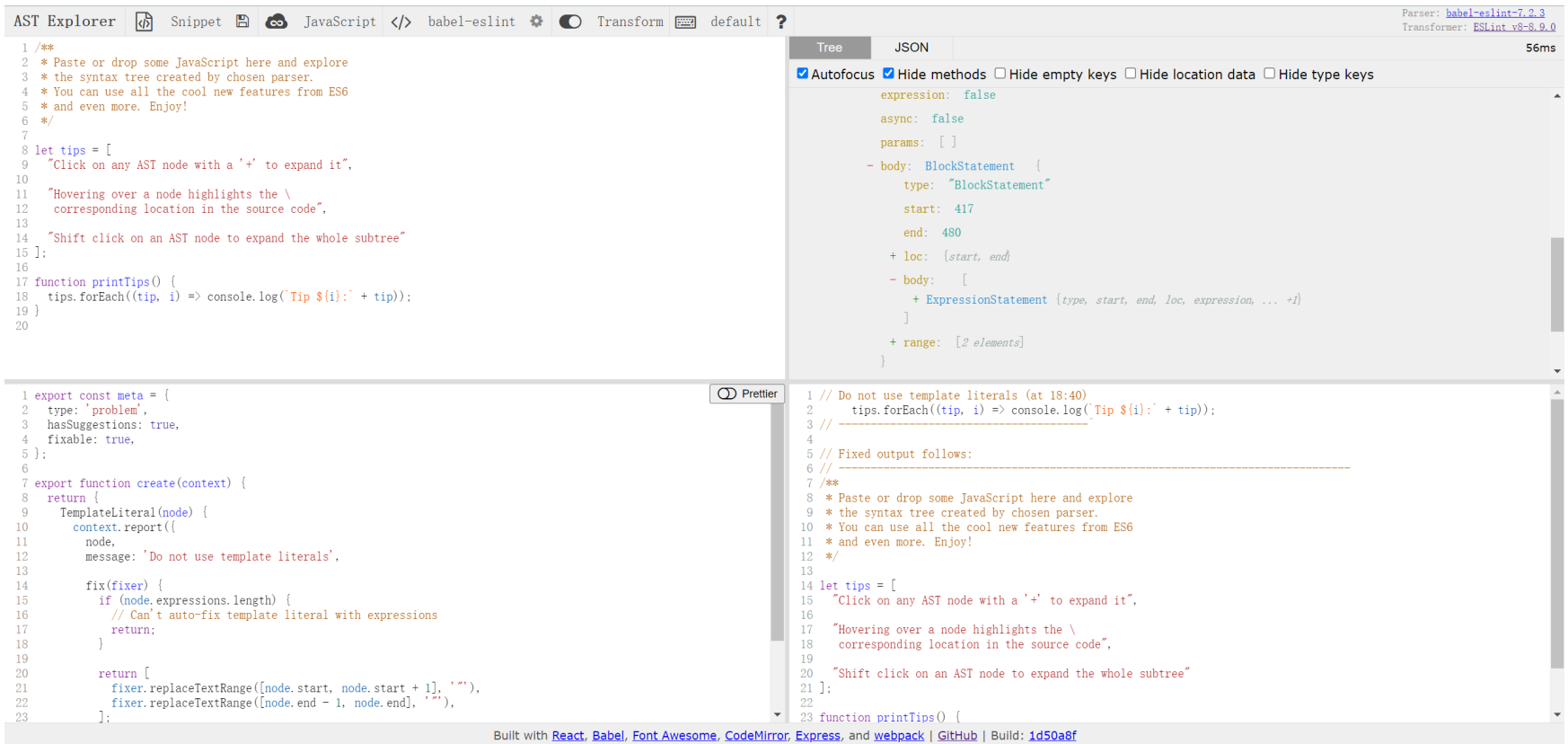
一个简单的案例

(1+2) * 3

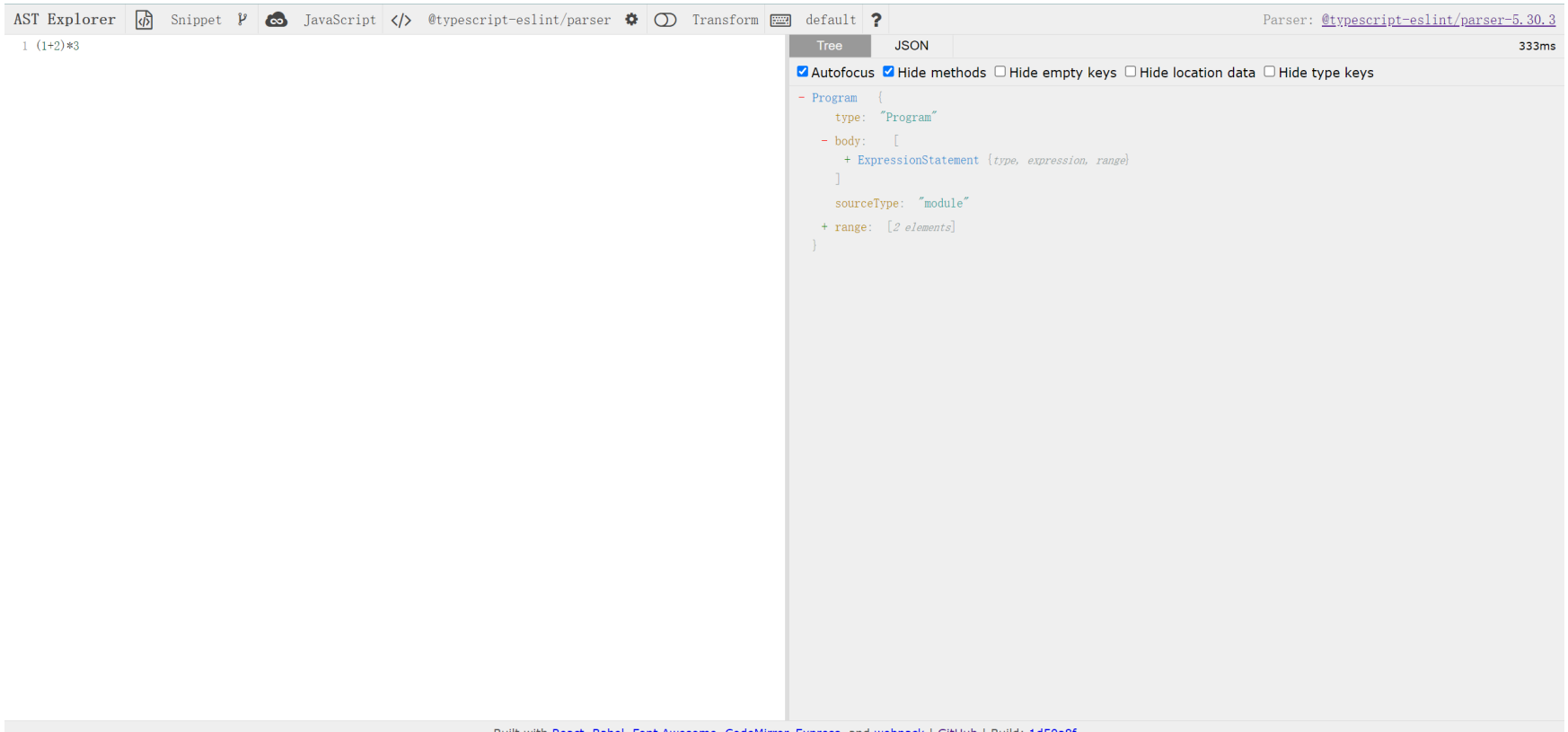


AST工具

AST explorer



Ast Explore: (1+2)*3



AST中的应用

1. 将 Javascript 代码进行格式化 (eslint/prettier)
2. 将 ES6+ 转化为 ES5 (babel)
3. 代码转换 (jscodeshift/gogocode)
4. 代码压缩 (terser)

AST与编译器

Vue的虚拟DOM，Babel的代码转换

1. Parse: 将字符串转为AST
2. Transform: 对AST进行操作
3. Code Generate: 生成新的代码

Parse

parse分为词法分析 和 语法分析

- 词法分析 将代码转为一组tokens， 每个token表示一个特有的语法

```
const a = 1

[
 {type: {}, value: 'const', start: 0, end: 5, loc: {}},
 {type: {}, value: 'a', start: 6, end: 7, loc: {}},
 {type: {}, value: '=', start: 8, end: 9, loc: {}},
 {type: {}, value: '1', start: 10, end: 11, loc: {}},
]
```

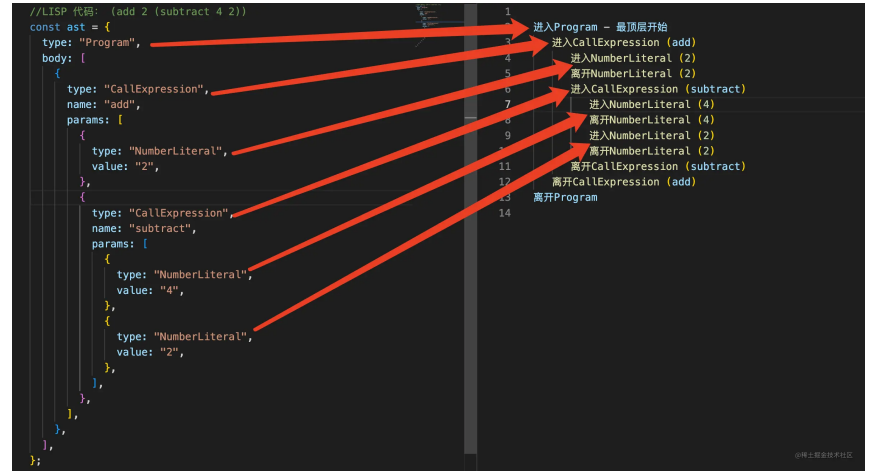
- 语法分析 将 Tokens 转化为结构化的 AST

Transform

根据当前AST（抽象语法树）生成一个新的AST（抽象语法树）

遍历AST（抽象语法树）的所有节点

```
{
  type: 'Program',
  body: [{
    type: 'CallExpression',
    name: 'add',
    params: [{
      type: 'NumberLiteral',
      value: '2'
    }, {
      type: 'CallExpression',
      name: 'subtract',
      params: [{
        type: 'NumberLiteral',
        value: '4'
      }, {
        type: 'NumberLiteral',
        value: '2'
      }]
    }]
```



Code Generate

代码生成阶段: 访问Transformation生成的AST(抽象语法树)或者再结合tokens，按照指定的规则，将“树”上的节点生成新的字符串

the-super-tiny-compiler

AST在JS体系中应用

- ESLint
- Babel

Start Code

```
// invalid
if(a) console.log(true)

// valid
if(!a){
 console.log(false)
}
```

```
import type { Rule } from 'eslint'

export default {
  meta: {
    docs: {
      description: 'Disallow IfStatement without blocks',
      category: 'Stylistic Issues',
      recommended: true,
    },
    fixable: 'code',
    // rule options
    schema: [],
  },
  create(context: Rule.RuleContext) {
    return {
      // visitor
    }
  },
}
```

Visitor

访问器最基本的思想是创建一个“访问器”对象，这个对象可以处理不同类型的节点函数

```
IfStatement(node) {
      if (!node.consequent)
        return
      if (node.consequent.type === 'BlockStatement')
        return

      context.report({
        node,
        message: 'IfStatement without blocks',
        fix(fixer) {
          return [
            fixer.insertTextBefore(node.consequent, '{'),
            fixer.insertTextAfter(node.consequent, '}'),
          ]
        },
      })
    }
```

Babel

Babel

- @babel/parser 可以把源码转换成`AST`
- @babel/traverse 用于对 `AST` 的遍历，维护了整棵树的状态，并且负责替换、移除和添加节点
- @babel/generator 可以把`AST`生成源码，同时生成`sourcemap`
- @babel/types 用于 `AST` 节点的 Lodash 式工具库, 它包含了构造、验证以及变换 `AST` 节点的方法，对编写处理 `AST` 逻辑非常有用
- @babel/core Babel 的编译器，核心 API 都在这里面，比如常见的 `transform`、`parse`，并实现了插件功能

Start Code

```
function testRegex(str){
  const reg = /regex/;
  return str.match(reg)
}
```

提升Regex

```
const _reg = /regex/;

function testRegex(str){
  return str.match(_reg)
}
```

步骤分解

1. 找到Program节点
2. 生成新的变量名来重命名，并用reg赋值
3. 生成新的AST

```
RegExpLiteral(path) {
        const program = path.findParent(types.isProgram)

        const name = path.parent.id.name
        const newIdentifier = path.scope.generateUidIdentifier(name)
        path.scope.rename(name, newIdentifier.name)
        const variableDeclaration = types.variableDeclaration('const', [types.variableDeclarator(newIdentifier, path.node)])

        program.node.body.unshift(variableDeclaration)
        path.parentPath.remove()
      },
```

AST在小布会议中的使用

Vue3中的setup

使用响应式 API 来声明响应式的状态，在 setup() 函数中返回的对象会暴露给模板和组件实例。

```
<script>
import { ref } from 'vue'

export default {
  setup() {
    const count = ref(0)

    // 返回值会暴露给模板和其他的选项式 API 钩子
    return {
      count
    }
  },

  mounted() {
    console.log(this.count) // 0
  }
}
</script>

<template>
  <button @click="count++">{{ count }}</button>
</template>
```

setup 语法糖

```
<script setup>
import { ref, onMounted } from 'vue'

const count = ref(0)

onMounted(()=>{
   console.log(count.value) // 0
})
</script>

<template>
  <button @click="count++">{{ count }}</button>
</template>
```

两种语法的对比

Vue3的语法 `Composition API: setup()` 和 `<script setup>`

`<script setup>` 是在单文件组件 (SFC) 中使用组合式 API 的编译时语法糖。当同时使用 SFC 与组合式 API 时该语法是默认推荐。相比于普通的 `<script>` 语法，它具有更多优势：

- 更少的样板内容，更简洁的代码。
- 能够使用纯 TypeScript 声明 props 和自定义事件。
- 更好的运行时性能 (其模板会被编译成同一作用域内的渲染函数，避免了渲染上下文代理对象)。
- 更好的 IDE 类型推导性能 (减少了语言服务器从代码中抽取类型的工作)。
- 方便调试

代码转换方法对比

原始的方法:

- 正则表达式

AST CodeMode工具

将函数的多个入参封装为一个对象传入，变成下面这样

```
import car from 'car';

const suv = car.factory('white', 'Kia', 'Sorento', 2010, 50000, r
const truck = car.factory(
  'silver',
  'Toyota',
  'Tacoma',
  2006,
  100000,
  true,
  true
);
```

```
import car from 'car';
const suv = car.factory({
  color: 'white',
  make: 'Kia',
  model: 'Sorento',
  year: 2010,
  miles: 50000,
  bedliner: null,
  alarm: true
});
const truck = car.factory({
  color: 'silver',
  make: 'Toyota',
  model: 'Tacoma',
  year: 2006,
  miles: 100000,
  bedliner: true,
  alarm: true
});
```

用 jscodeshift 实现

```
export default (fileInfo, api) => {
  const j = api.jscodeshift;
  const root = j(fileInfo.source);

  // find declaration for "car" import
  const importDeclaration = root.find(j.Im
    source: {
      type: 'Literal',
      value: 'car'
    }
  });

  // get the local name for the imported m
  const localName = importDeclaration.find

  // current order of arguments
  const argKeys = [
    'color',
    'make',
    'model',
    'year',
    'miles',
    'bedliner',
    'alarm'
```

```
// find where `.factory` is being called
 return (
   root
     .find(j.CallExpression, {
       callee: {
         type: 'MemberExpression',
         object: {
           name: localName
         },
         property: {
           name: 'factory'
         }
       }
     })
```

```
      .replaceWith(nodePath => {
        const { node } = nodePath;

        // use a builder to create the Obj
        const argumentsAsObject = j.objectE
          // map the arguments to an Array
          node.arguments.map((arg, i) =>
            j.property('init', j.identifie
          )
        );

        // replace the arguments with our
        node.arguments = [argumentsAsObjec

        return node;
      })

      // specify print options for recast
      .toSource({ quote: 'single', trailin
  );
```

用gogocode实现

```
const argKeys = [
  'color',
  'make',
  'model',
  'year',
  'miles',
  'bedliner',
  'alarm'
];
const argObj = {};
$(code)
  .find(`const $_$1 = car.factory($_$2);`)
  .each(item => {
    const variableName = item.match[1][0].value;
    item.match[2].forEach((match, j) => {
        argObj[argKeys[j]] = match.value;
    });
    item.replaceBy(
        $(`const ${variableName} = car.factory(${JSON.stringify(argObj)})`)
    );
  })
  .root()
  .generate()
```

Transform Setup

```vue
<script lang="ts">
import { defineComponent, reactive, ref } from "vue"
import Hello from 'hello.vue'
export default defineComponent({
  components: { Hello },
  props: {
    title: {
      type: String,
      default: '',
    },
  },
  emits: ['accepted'],
  setup(props, context) {
    const a = ref(1)
    const b = reactive([])

    function accept() {
      context.emit('accepted')
    }

    return {
      accept,
      a,
      b,
```

```vue
<script setup lang="ts">
import { reactive,ref,defineProps,defineEmits } from 'vue'
import Hello from 'hello.vue'
const props = defineProps({
  title: {
    type: String,
    default: '',
  },
})
const emit = defineEmit(['accepted'])
const a = ref(1)
const b = reactive([])

function accept() {
  emit('accepted')
}
</script>
```

转换步骤

GoGoCode PlayGround

1. 提取出setup中的给个Node节点，并且移除Setup节点

```
script.find('{ setup() {$$$0} }').match.$$$0.forEach((node) => {
    if (node.type !== 'ReturnStatement')
      script.after(node)
})
script.find('setup(){}').remove()
```

$$$ 通配符

转换步骤

2. 移除 defineComponent

```
// remove defineComponent
script.find('export default $_$1').each((item) => {
  item.remove()
})

// remove import defineComponent
script.replace(
  'import { $$$1,defineComponent } from \'vue\'',
  'import { $$$1 } from \'vue\'',
)
```

replace

Summary

- 什么时候使用AST工具?

Summary

不足点：

1. 纯JS，缺乏一定类型推导
2. 有一定的学习成本。开发过程中可能会有很多调试的工作。不能和DOM一样
    1. 查询：getElement* , querySelector*
    2. 创建：createElement
    3. 修改：prepend，prepend，before，after，remove，replaceWith