写一个编译器

程序员cxuan 2023-01-12 16:30 Posted on 河北

The following article is from HelloGitHub Author 点击关注→

HelloGitHub

分享 GitHub 上有趣、入门级的开源项目。

不知道你是不是和我一样，看到“编译器”三个字的时候，就感觉非常高大上，同时心底会升起一丝丝“害怕”！

我始终认为编译器是很复杂...很复杂的东西，不是我这种小白能懂的。而且一想到要学习编译器的知识，脑海里就浮现出那种 500 页起的厚书。

一直到我发现 the-super-tiny-compiler 这个宝藏级的开源项目，它是一个仅 1000 行左右的迷你编译器，其中注释占了代码量的 80%，实际代码只有 200 行！麻雀虽小但五脏俱全，完整地实现了编译器所需基本功能，通过 代码+注释+讲解 让你通过一个开源项目入门编译器。

> 地址：https://github.com/jamiebuilds/the-super-tiny-compiler
>
> 中文：https://github.com/521xueweihan/OneFile/blob/main/src/javascript/the-super-tiny-compiler.js

下面我将从介绍 什么是编译器 开始，使用上述项目作为示例代码，更加细致地讲解编译的过程，把编译器入门的门槛再往下砍一砍。如果你之前没有接触过编译器相关的知识，那这篇文章可以让你对**编译器所做的事情，以及原理有一个初步的认识！**

准备好变强了吗？那我们开始吧！

一、什么是编译器

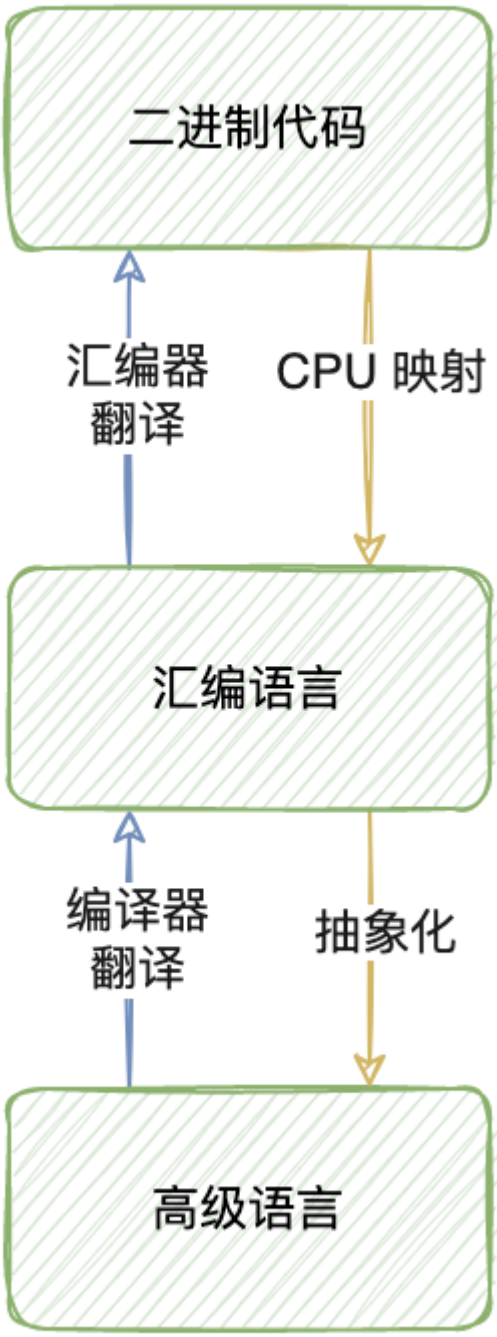
从概念上简单讲：

编译器就是将“一种语言（通常为高级语言）”翻译为“另一种语言（通常为低级语言）”的程序。

对于现代程序员来说我们最熟悉的 JavaScript、Java 这些都属于高级语言，也就是便于我们编程者编写、阅读、理解、维护的语言，而低级语言就是计算机能直接解读、运行的语言。

编译器也可以理解成是这两种语言之间的“桥梁”。编译器存在的原因是因为计算机 CPU 执行数百万个微小的操作，因为这些操作实在是太“微小”，你肯定不愿意手动去编写它们，于是就有了二进制的出现，二进制代码也被理解成为机器代码。很显然，二进制看上去并不好理解，而且编写二进制代码很麻烦，因此 CPU 架构支持把二进制操作映射作为一种更容易阅读的语言——汇编语言。

虽然汇编语言非常低级，但是它可以转换为二进制代码，这种转换主要靠的是“汇编器”。因为汇编语言仍然非常低级，对于追求高效的程序员来说是无法忍受的，所以又出现了更高级的语言，这也是大部分程序员使用且熟悉的编程语言，这些抽象的编程语言虽然不能直接转化成机器操作，但是它比汇编语言更好理解且更能够被高效的使用。所以我们需要的其实就是能理解这些复杂语言并正确地转换成低级代码的工具——编译器。

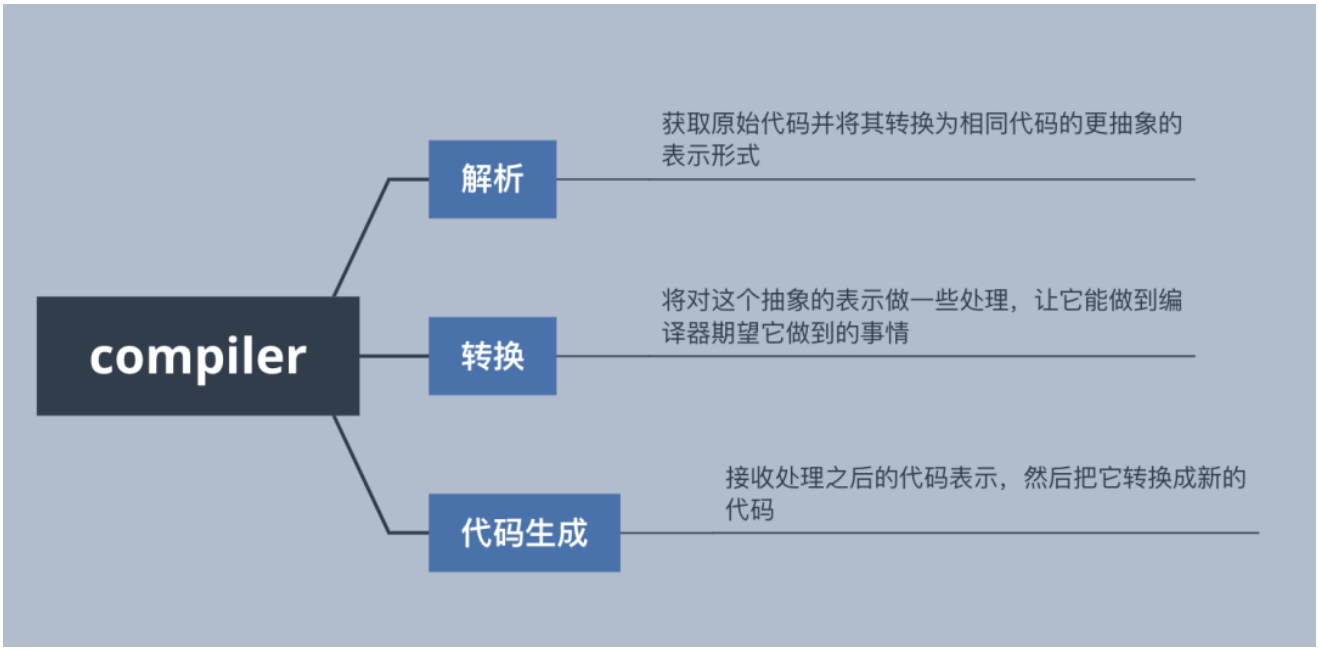


我觉得对于初学者来说到这里有个大致的了解就可以了。因为接下去要分析的这个例子非常简单但是能覆盖大多数场景，你会从最真实最直接的角度来直面这个“大敌”——编译器。

二、“迷你”编译器

下面我们就用 the-super-tiny-compiler 为示例代码，带大家来简单了解一下编译器。

不同编译器之间的不同阶段可能存在差别，但基本都离不开这三个主要组成部分：解析、转换和代码生成。



其实这个“迷你”编译器开源项目的目的就是这些：

- 证明现实世界的编译器主要做的是什么
- 做一些足够复杂的事情来证明构建编译器的合理性
- 用最简单的代码来解释编译器的主要功能，使新手不会望而却步

以上就解释了这个开源项目存在的意义了，所以如果你对编译器有很浓厚的兴趣希望一学到底的，那肯定还是离不开大量的阅读和钻研啦，但是如果你希望对编译器的功能有所了解，那这篇文章就别错过啦！

现在我们就要对这个项目本身进行进一步的学习了，有些背景需要提前了解一下。这个项目主要是把 LISP 语言编译成我们熟悉的 JavaScript 语言。

那为什么要用 LISP 语言呢?

> LISP 是具有悠久历史的计算机编程语言家族，有独特和完全用括号的前缀符号表示法。起源于 1958 年，是现今第二悠久仍广泛使用的高端编程语言。

首先 LISP 语言和我们熟悉的 C 语言和 JavaScript 语言很不一样，虽然其他的语言也有强大的编译器，但是相对于 LISP 语言要复杂得多。LISP 语言是一种超级简单的解析语法，并且很容易被翻译成其他语法，像这样：

如果我们有两个函数 `add` 和 `subtract` 他们会写成这样：

LISP	C
2 + 2 (add 2 2)	add(2, 2)
4 – 2 (subtract 4 2)	subtract(4, 2)
2 + (4 – 2) (add 2 (subtract 4 2))	add(2, subtract(4, 2))

所以到这里你应该知道我们要干什么了吧？那让我们再深入地了解一下具体要怎么进行“翻译”（编译）吧！

三、编译过程

前面我们已经提过，大部分的编译器都主要是在做三件事：

1. 解析
2. 转换
3. 代码生成

下面我们将分解 the-super-tiny-compiler 的代码，然后进行逐行解读。

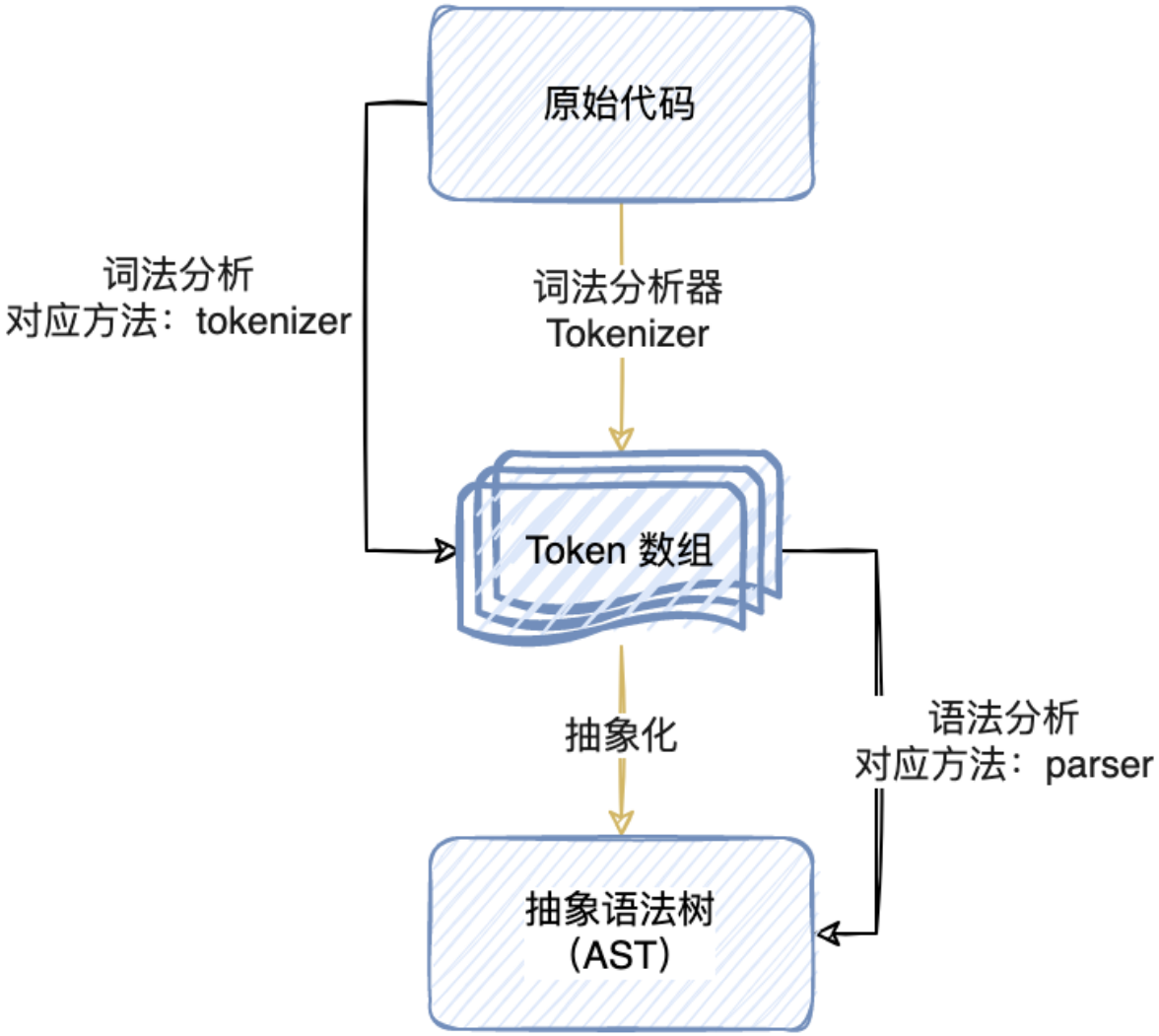
让我们一起跟着代码，弄清楚上述三个阶段具体做了哪些事情～

3.1 解析

解析通常分为两个阶段：**词法分析**和**句法分析**

1. 词法分析：获取原始代码并通过一种称为标记器（或词法分析器 Tokenizer）的东西将其拆分为一种称为标记（Token）的东西。标记是一个数组，它描述了一个独立的语法片段。这些片段可以是数字、标签、标点符号、运算符等等。

2. 语法分析：获取之前生成的标记（Token），并把它们转换成一种抽象的表示，这种抽象的表示描述了代码语句中的每一个片段以及它们之间的关系。这被称为中间表示（intermediate representation）或抽象语法树（Abstract Syntax Tree，缩写为AST）。AST 是一个深度嵌套的对象，用一种更容易处理的方式代表了代码本身，也能给我们更多信息。



比如下面这个语法:

```
(add 2 (subtract 4 2))
```

拆成 Token 数组就像这样:

```
[
  { type: 'paren',  value: '('        },
  { type: 'name',   value: 'add'      },
  { type: 'number', value: '2'        },
  { type: 'paren',  value: '('        },
  { type: 'name',   value: 'subtract' },
  { type: 'number', value: '4'        },
```

```
  { type: 'number', value: '2'       },
  { type: 'paren',  value: ')'       },
  { type: 'paren',  value: ')'       },
]
```

代码思路：

因为我们需要去解析字符串，就需要有一个像指针/光标来帮我们辨认目前解析的位置是哪里，所以会有一个 current 的变量，从 0 开始。而我们最终的目的是获取一个 token 数组，所以也先初始化一个空数组 tokens 。

```
// `current` 变量就像一个指光标一样让我们可以在代码中追踪我们的位置
let current = 0;

// `tokens` 数组用来存我们的标记
let tokens = [];
```

既然要解析字符串，自然少不了遍历啦！这里就用一个 while 循环来解析我们当前的字符串。

```
// 在循环里面我们可以将`current`变量增加为我们想要的值
while (current < input.length) {
  // 我们还将在 `input` 中存储 `current` 字符
  let char = input[current];
}
```

如何获取字符串里面的单个字符呢？答案是用像数组那样的中括号来获取：

```
var char = str[0]
```

这里新增一个知识点来咯！在 JavaScript 中 String 类的实例，是一个类数组，从下面这个例子可以看出来：

```
> new String('abc')
< String {'abc'}
    0: "a"
    1: "b"
    2: "c"
    length: 3
  ▶ [[Prototype]]: String
    [[PrimitiveValue]]: "abc"
```

可能之前你会用 charAt 来获取字符串的单个字符，因为它是在 String 类型上的一个方法：

```
< String {'', constructor: ƒ, anchor: ƒ, big: ƒ, blink: ƒ, …}
  ▶ anchor: ƒ anchor()
  ▶ at: ƒ at()
  ▶ big: ƒ big()
  ▶ blink: ƒ blink()
  ▶ bold: ƒ bold()
  ▶ charAt: ƒ charAt()
  ▶ charCodeAt: ƒ charCodeAt()
```

这两个方法都可以实现你想要的效果，但是也存在差别。下标不存在时 str[index] 会返回 undefined，而 str.charAt(index) 则会返回 "" (空字符串)：

```
> var a = 'abc'
< undefined
> a[0]
< 'a'
> a[1]
< 'b'
> a.charAt(3)
< ''
> a[3]
< undefined
```

随着光标的移动和字符串中字符的获取，我们就可以来逐步解析当前字符串了。

那解析也可以从这几个方面来考虑，以 (add 2 (subtract 4 2)) 这个为例，我们会遇到这些：(左括号、字符串、空格、数字、) 右括号。对于不同的类型，就要用不同的 if 条件判断分别处理：

- 左右括号匹配代表一个整体，找到对应的括号只要做上标记就好
- 空格代表有字符分割，不需要放到我们的 token 数组里，只需要跳到下一个非空格的字符继续循环就好

```
// 检查是否有一个左括号:
if (char === '(') {

  // 如果有，我们会存一个类型为 `paren` 的新标记到数组，并将值设置为一个左括号。
  tokens.push({
    type: 'paren',
    value: '(',
  });

  // `current`自增
  current++;

  // 然后继续进入下一次循环。
  continue;
}

// 接下去检查右括号， 像上面一样
if (char === ')') {
  tokens.push({
    type: 'paren',
    value: ')',
  });
  current++;
  continue;
}

// 接下去我们检查空格，这对于我们来说就是为了知道字符的分割，但是并不需要存储为标记。

// 所以我们来检查是否有空格的存在，如果存在，就继续下一次循环，做除了存储到标记数组之外的其他
let WHITESPACE = /\s/;
if (WHITESPACE.test(char)) {
  current++;
  continue;
}
```

字符串和数字因为具有各自不同的含义，所以处理上面相对复杂一些。先从数字来入手，因为数字的长度不固定，所以要确保获取到全部的数字字符串呢，就要经过遍历，从遇到第一个数字开始直到遇到一个不是数字的字符结束，并且要把这个数字存起来。

```
//   (add 123 456)
//        ^^^ ^^^
//        只有两个单独的标记
//
// 因此，当我们遇到序列中的第一个数字时，我们就开始了
let NUMBERS = /[0-9]/;
if (NUMBERS.test(char)) {

  // 我们将创建一个`value`字符串，并把字符推送给他
  let value = '';

  // 然后我们将遍历序列中的每个字符，直到遇到一个不是数字的字符
  // 将每个作为数字的字符推到我们的 `value` 并随着我们去增加 `current`
  // 这样我们就能拿到一个完整的数字字符串，例如上面的 123 和 456，而不是单独的 1 2 3 4 5 6
  while (NUMBERS.test(char)) {
    value += char;
    char = input[++current];
  }

  // 接着我们把数字放到标记数组中，用数字类型来描述区分它
  tokens.push({ type: 'number', value });

  // 继续外层的下一次循环
  continue;
}
```

为了更适用于现实场景，这里支持字符串的运算，例如 `(concat "foo" "bar")` 这种形式的运算，那就要对 " 内部的字符串再做一下解析，过程和数字类似，也需要遍历，然后获取全部的字符串内容之后再存起来：

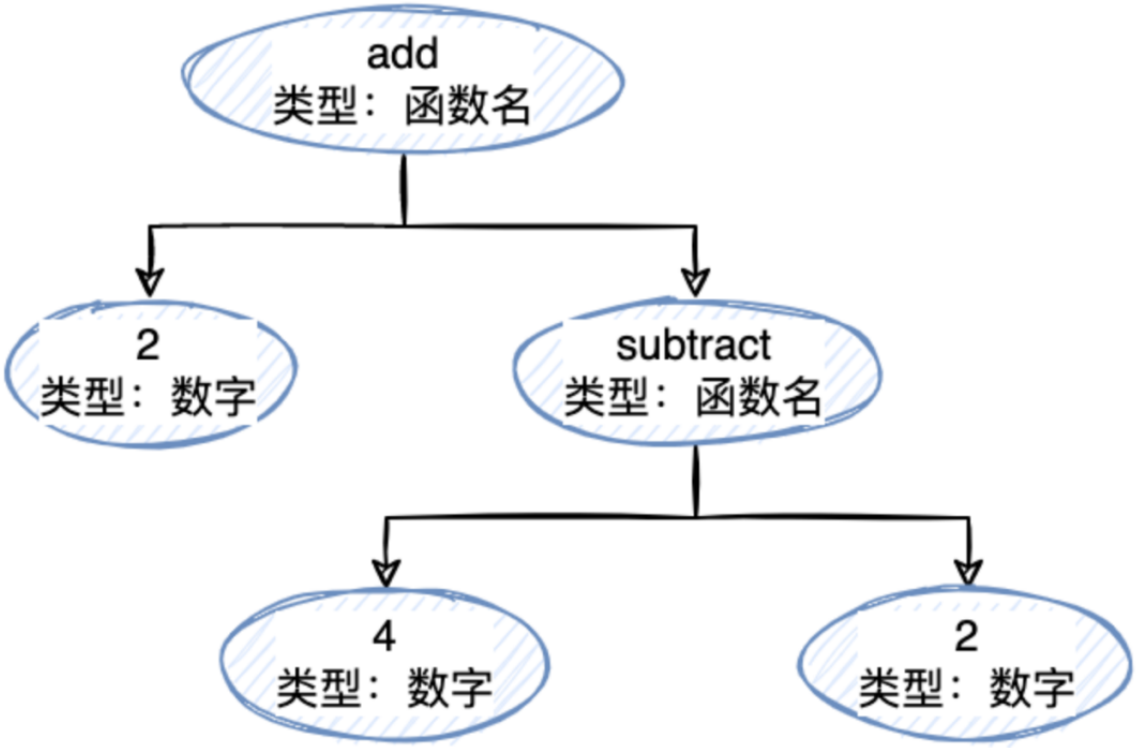
```
// 从检查开头的双引号开始:
if (char === '"') {
  // 保留一个 `value` 变量来构建我们的字符串标记。
  let value = '';

  // 我们将跳过编辑中开头的双引号
  char = input[++current];

  // 然后我们将遍历每个字符，直到我们到达另一个双引号
  while (char !== '"') {
```

```
    value += char;
    char = input[++current];
  }

  // 跳过相对应闭合的双引号.
  char = input[++current];

  // 把我们的字符串标记添加到标记数组中
  tokens.push({ type: 'string', value });

  continue;
}
```

最后一种标记的类型是名称。这是一个字母序列而不是数字，这是我们 lisp 语法中的函数名称：

```
// (add 2 4)
//  ^^^
//  名称标记
//
let LETTERS = /[a-z]/i;
if (LETTERS.test(char)) {
  let value = '';

  // 同样，我们遍历所有，并将它们完整的存到`value`变量中
  while (LETTERS.test(char)) {
    value += char;
    char = input[++current];
  }

  // 并把这种名称类型的标记存到标记数组中，继续循环
  tokens.push({ type: 'name', value });

  continue;
}
```

以上我们就能获得一个 tokens 数组了，下一步就是构建一个抽象语法树（AST）可能看起来像这样：




```
{
  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',
      }]
    }]
  }]
}
```

代码思路：

同样我们也需要有一个光标/指针来帮我们辨认当前操作的对象是谁，然后预先创建我们的 AST 树，他有一个根节点叫做 `Program` ：

```
let current = 0;

let ast = {
  type: 'Program',
  body: [],
};
```

再来看一眼我们之前获得的 `tokens` 数组：

```
[
  { type: 'paren',  value: '('        },
  { type: 'name',   value: 'add'      },
  { type: 'number', value: '2'        },
  { type: 'paren',  value: '('        },
  { type: 'name',   value: 'subtract' },
  { type: 'number', value: '4'        },
  { type: 'number', value: '2'        },
  { type: 'paren',  value: ')'        },
  { type: 'paren',  value: ')'        },
]
```

你会发现对于 `(add 2 (subtract 4 2))` 这种具有嵌套关系的字符串，这个数组非常“扁平”也无法明显的表达嵌套关系，而我们的 AST 结构就能够很清晰的表达嵌套的关系。对于上面的数组来说，我们需要遍历每一个标记，找出其中是 `CallExpression` 的 `params` ，直到遇到右括号结束，所以递归是最好的方法，所以我们创建一个叫 `walk` 的递归方法，这个方法返回一个 `node` 节点，并存入我们的 `ast.body` 的数组中：

```
function walk() {
 // 在 walk 函数里面，我们首先获取`current`标记
  let token = tokens[current];
}

while (current < tokens.length) {
  ast.body.push(walk());
}
```

下面就来实现我们的 walk 方法。我们希望这个方法可以正确解析 tokens 数组里的信息，首先就是要针对不同的类型 type 作区分：

首先先操作“值”，因为它是不会作为父节点的所以也是最简单的。在上面我们已经了解了值可能是数字 (subtract 4 2) 也可能是字符串 (concat "foo" "bar")，只要把值和类型匹配上就好：

```
// 首先先检查一下是否有`number`标签.
if (token.type === 'number') {

  // 如果找到一个,就增加`current`.
  current++;

  // 然后我们就能返回一个新的叫做`NumberLiteral`的 AST 节点，并赋值
  return {
    type: 'NumberLiteral',
    value: token.value,
  };
}

// 对于字符串来说，也是和上面数字一样的操作。新增一个`StringLiteral`节点
if (token.type === 'string') {
  current++;

  return {
    type: 'StringLiteral',
    value: token.value,
  };
}
```

接下去我们要寻找调用的表达式（CallExpressions）。每匹配一个左括号，就能在下一个得到表达式的名字，在没有遇到右括号之前都经过递归把树状结构丰富起来，直到遇到右括号停止递归，直到循环结束。从代码上看更加直观：

```
if (
  token.type === 'paren' &&
  token.value === '('
) {

  // 我们将增加`current`来跳过这个插入语，因为在 AST 树中我们并不关心这个括号
  token = tokens[++current];
```

```
    // 我们创建一个类型为“CallExpression”的基本节点，并把当前标记的值设置到 name 字段上
    // 因为左括号的下一个标记就是这个函数的名字
    let node = {
      type: 'CallExpression',
      name: token.value,
      params: [],
    };

    // 继续增加`current`来跳过这个名称标记
    token = tokens[++current];

    // 现在我们要遍历每一个标记，找出其中是`CallExpression`的`params`，直到遇到右括号
    // 我们将依赖嵌套的`walk`函数来增加我们的`current`变量来超过任何嵌套的`CallExpression`
    // 所以我们创建一个`while`循环持续到遇到一个`type`是'paren'并且`value`是右括号的标记
    while (
      (token.type !== 'paren') ||
      (token.type === 'paren' && token.value !== ')')
    ) {
      // 我们把这个节点存到我们的`node.params`中去
      node.params.push(walk());
      token = tokens[current];
    }

    // 我们最后一次增加`current`变量来跳过右括号
    current++;

    // 返回node节点
    return node;
  }
```

3.2 转换

编译器的下一个阶段是转换。要做的就是获取 AST 之后再对其进行更改。它可以用相同的语言操作 AST，也可以将其翻译成一种全新的语言。

那如何转换 AST 呢?

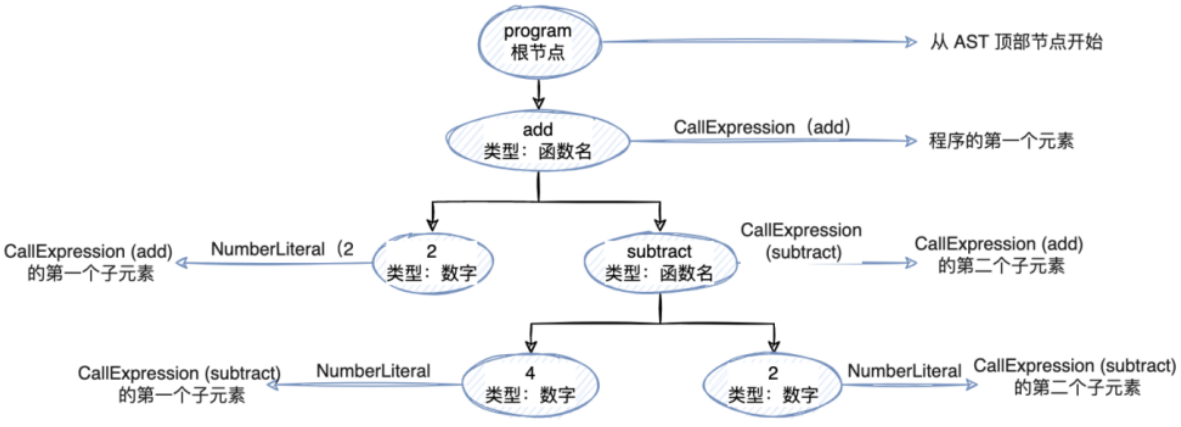
你可能会注意到我们的 AST 中的元素看起来非常相似。这些元素都有 type 属性，它们被称为 AST 结点。这些节点含有若干属性，可以用于描述 AST 的部分信息。

```
// 我们可以有一个“NumberLiteral”的节点：
{
  type: 'NumberLiteral',
  value: '2',
}

// 或者可能是“CallExpression”的一个节点：
{
  type: 'CallExpression',
  name: 'subtract',
  params: [...嵌套节点放在这里...],
}
```

对于转换 AST 无非就是通过新增、删除、替换属性来操作节点，或者也可以新增节点、删除节点，甚至我们可以在原有的 AST 结构保持不变的状态下创建一个基于它的全新的 AST。

由于我们的目标是一种新的语言，所以我们将要专注于创造一个完全新的 AST 来配合这个特定的语言。

为了能够访问所有这些节点，我们需要遍历它们，使用的是深度遍历的方法。对于我们在上面获取的那个 AST 遍历流程应该是这样的：



如果我们直接操作这个 AST 而不是创造一个单独的 AST，我们很有可能会在这里引入各种抽象。但是仅仅访问树中的每个节点对于我们来说想做和能做的事情已经很多了。

（使用访问（visiting）这个词是因为这是一种模式，代表在对象结构内对元素进行操作。）

所以我们现在创建一个访问者对象（visitor），这个对象具有接受不同节点类型的方法：

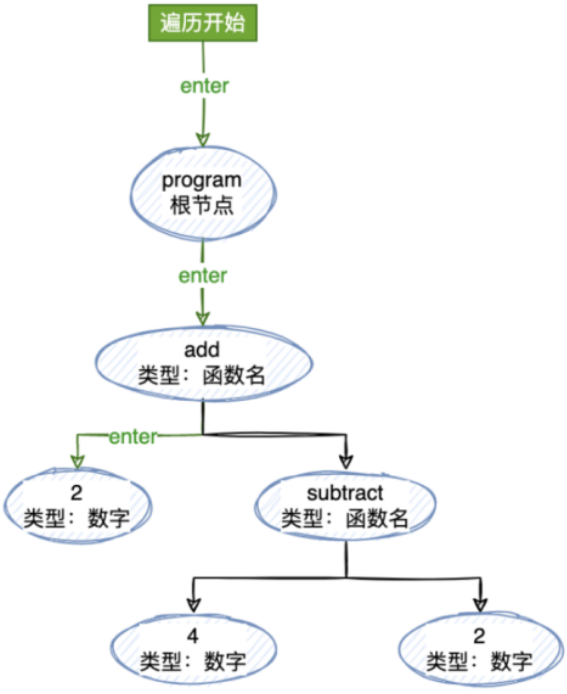
```
var visitor = {
    NumberLiteral() {},
    CallExpression() {},
};
```

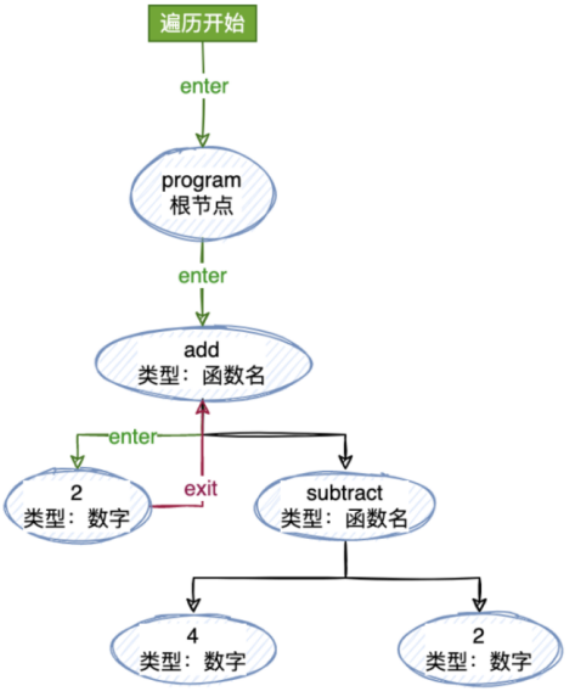
当我们遍历 AST 的时候，如果遇到了匹配 type 的结点，我们可以调用 visitor 中的方法。

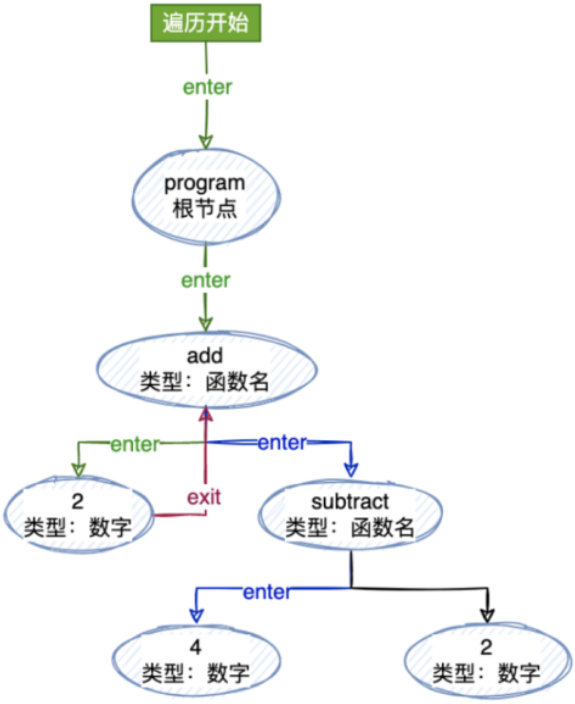
一般情况下为了让这些方法可用性更好，我们会把父结点也作为参数传入。

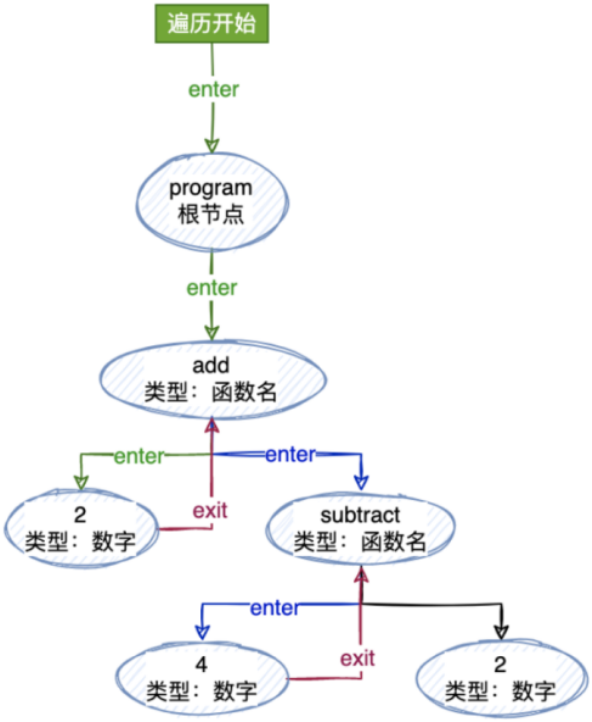
```
var visitor = {
    NumberLiteral(node, parent) {},
    CallExpression(node, parent) {},
};
```

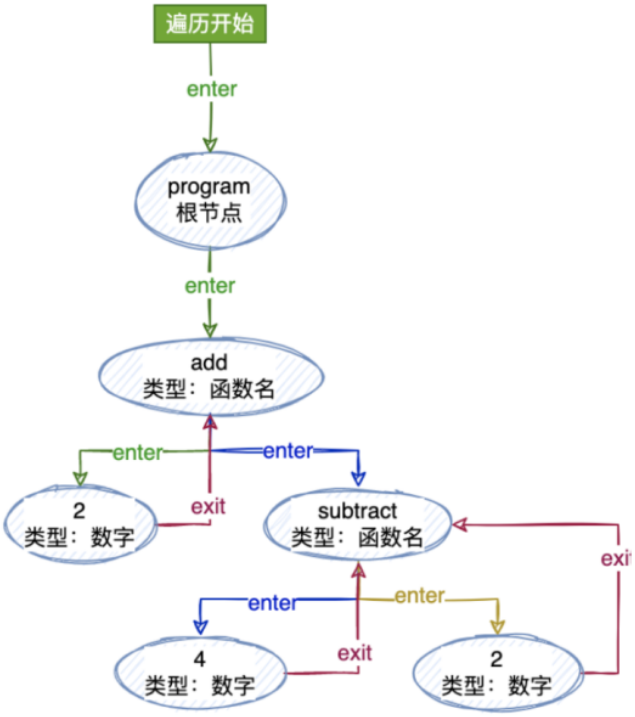
当然啦，对于深度遍历的话我们都知道，往下遍历到最深的自节点的时候还需要“往回走”，也就是我们所说的“退出”当前节点。你也可以这样理解：向下走“进入”节点，向上走“退出”节点。

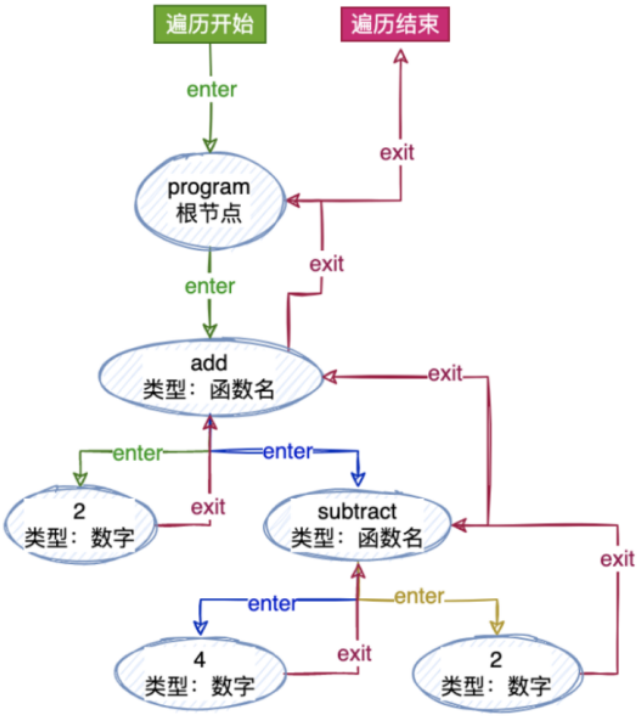












为了支持这点，我们的“访问者”的最终形式应该像是这样：

```
var visitor = {
  Program: {
    enter(node, parent) {},
```

```
    exit(node, parent) {},
  },
  NumberLiteral: {
    enter(node, parent) {},
    exit(node, parent) {},
  },
  CallExpression: {
     enter(node, parent) {},
     exit(node, parent) {},
  },
  ...
};
```

遍历

首先我们定义了一个接收一个 AST 和一个访问者的遍历器函数（traverser）。需要根据每个节点的类型来调用不同的访问者的方法，所以我们定义一个 `traverseNode` 的方法，传入当前的节点和它的父节点，从根节点开始，根节点没有父节点，所以传入 `null` 即可。

```
function traverser(ast, visitor) {
  // traverseNode 函数将接受两个参数：node 节点和他的 parent 节点
  // 这样他就可以将两者都传递给我们的访问者方法（visitor）
  function traverseNode(node, parent) {

    // 我们首先从匹配`type`开始，来检测访问者方法是否存在。访问者方法就是（enter 和 exit）
    let methods = visitor[node.type];

    // 如果这个节点类型有`enter`方法，我们将调用这个函数，并且传入当前节点和他的父节点
    if (methods && methods.enter) {
      methods.enter(node, parent);
    }

    // 接下去我们将按当前节点类型来进行拆分，以便于对子节点数组进行遍历，处理到每一个子节点
    switch (node.type) {
      // 首先从最高的`Program`层开始。因为 Program 节点有一个名叫 body 的属性，里面包含了节点数组
      // 我们调用`traverseArray`来向下遍历它们
      //
      // 请记住，`traverseArray`会依次调用`traverseNode`，所以这棵树将会被递归遍历
      case 'Program':
        traverseArray(node.body, node);
```

```
      break;

    // 接下去我们对`CallExpression`做相同的事情，然后遍历`params`属性
    case 'CallExpression':
      traverseArray(node.params, node);
      break;

    // 对于`NumberLiteral`和`StringLiteral`的情况，由于没有子节点，所以直接 break 即可
    case 'NumberLiteral':
    case 'StringLiteral':
      break;

    // 接着，如果我们没有匹配到上面的节点类型，就抛出一个异常错误
    default:
      throw new TypeError(node.type);
  }

  // 如果这个节点类型里面有一个`exit`方法，我们就调用它，并且传入当前节点和他的父节点
  if (methods && methods.exit) {
    methods.exit(node, parent);
  }
}

// 调用`traverseNode`来启动遍历，传入之前的 AST 树，由于 AST 树最开始
// 的点没有父节点，所以我们直接传入 null 就好
traverseNode(ast, null);
}
```

因为 Program 和 CallExpression 两种类型可能会含有子节点，所以对这些可能存在的子节点数组需要做进一步的处理，所以创建了一个叫 traverseArray 的方法来进行迭代。

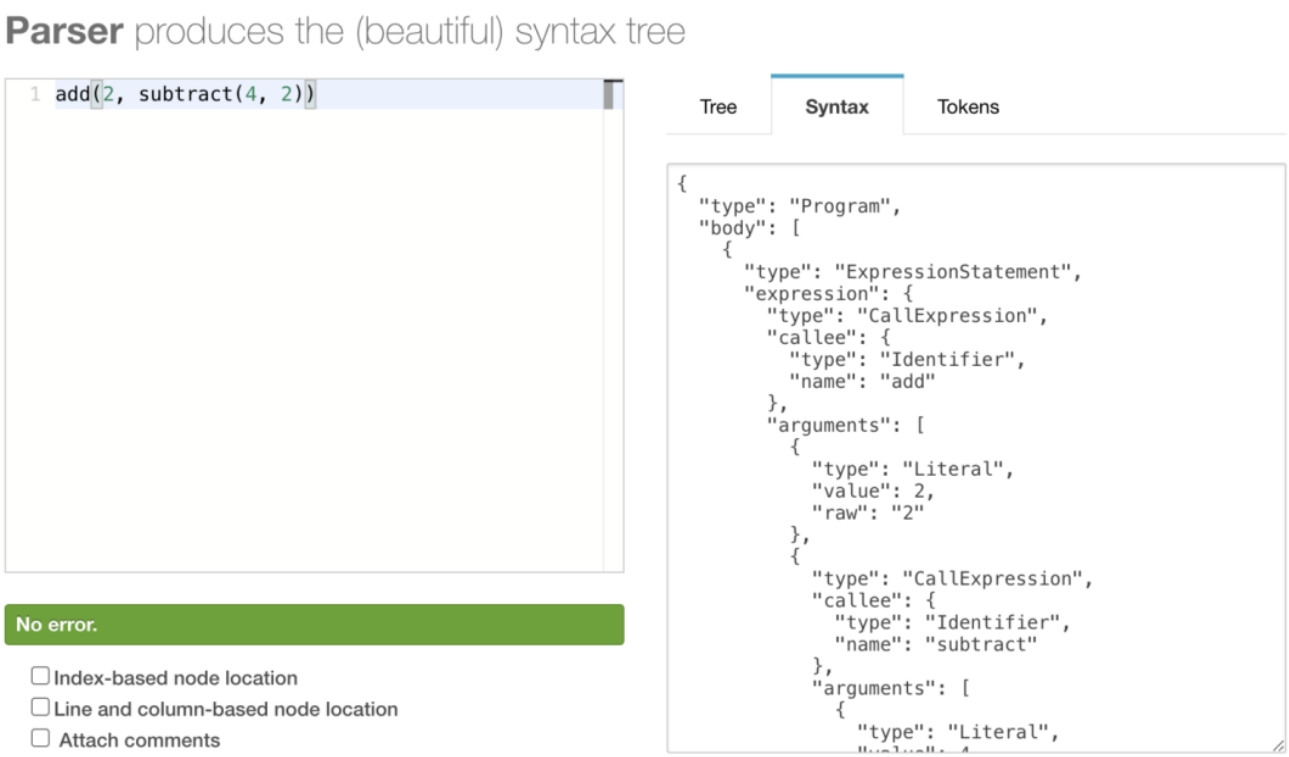
```
// traverseArray 函数来迭代数组，并且调用 traverseNode 函数
function traverseArray(array, parent) {
  array.forEach(child => {
    traverseNode(child, parent);
  });
}
```

转换

下一步就是把之前的 AST 树进一步进行转换变成我们所期望的那样变成 JavaScript 的 AST 树：

```
* ----------------------------------------------------------------------------
*   Original AST                     |   Transformed AST
* ----------------------------------------------------------------------------
*   {                                |   {
*     type: 'Program',               |     type: 'Program',
*     body: [{                       |     body: [{
*       type: 'CallExpression',      |       type: 'ExpressionStatement',
*       name: 'add',                 |       expression: {
*       params: [{                   |         type: 'CallExpression',
*         type: 'NumberLiteral',     |         callee: {
*         value: '2'                 |           type: 'Identifier',
*       }, {                         |           name: 'add'
*         type: 'CallExpression',    |         },
*         name: 'subtract',          |         arguments: [{
*         params: [{                 |           type: 'NumberLiteral',
*           type: 'NumberLiteral',   |           value: '2'
*           value: '4'               |         }, {
*         }, {                       |           type: 'CallExpression',
*           type: 'NumberLiteral',   |           callee: {
*           value: '2'               |             type: 'Identifier',
*         }]                         |             name: 'subtract'
*       }]                           |           },
*     }]                             |           arguments: [{
*   }                                |             type: 'NumberLiteral',
*                                    |             value: '4'
* ---------------------------------- |           }, {
*                                    |             type: 'NumberLiteral',
*                                    |             value: '2'
*                                    |           }]
*  (sorry the other one is longer.)  |         }
*                                    |       }
*                                    |     }]
*                                    |   }
* ----------------------------------------------------------------------------
```

如果你对 JS 的 AST 的语法解析不是很熟悉的话，可以借助在线工具网站来帮助你知道大致要转换成什么样子的 AST 树，就可以在其他更复杂的场景进行应用啦 ~



我们创建一个像之前的 AST 树一样的新的 AST 树，也有一个 Program 节点：

```
function transformer(ast) {

  let newAst = {
    type: 'Program',
    body: [],
  };

  // 这里有个 hack 技巧：这个上下文（context）属性只是用来对比新旧 ast 的
  // 通常你会有比这个更好的抽象方法，但是为了我们的目标能实现，这个方法相对简单些
  ast._context = newAst.body;

  // 在这里调用遍历器函数并传入我们的旧的 AST 树和访问者方法(visitor)
  traverser(ast, {...});

  // 在转换器方法的最后，我们就能返回我们刚创建的新的 AST 树了
  return newAst;
}
```

那我们再来完善我们的 visitor 对象，对于不同类型的节点，可以定义它的 enter 和 exit 方法（这里因为只要访问到节点并进行处理就可以了，所以用不到退出节点的方法： exit ）：

```
{
  // 第一个访问者方法是 NumberLiteral
  NumberLiteral: {
    enter(node, parent) {
      // 我们将创建一个也叫做`NumberLiteral`的新节点，并放到父节点的上下文中去
      parent._context.push({
        type: 'NumberLiteral',
        value: node.value,
      });
    },
  },

  // 接下去是 `StringLiteral`
  StringLiteral: {
    enter(node, parent) {
      parent._context.push({
        type: 'StringLiteral',
        value: node.value,
      });
    },
  },

  CallExpression: {...}
}
```

对于 CallExpression 会相对比较复杂一点，因为它可能含有嵌套的内容。

```
CallExpression: {
  enter(node, parent) {

    // 首先我们创建一个叫`CallExpression`的节点，它带有表示嵌套的标识符"Identifier"
    let expression = {
      type: 'CallExpression',
      callee: {
        type: 'Identifier',
        name: node.name,
      },
      arguments: [],
    };
```

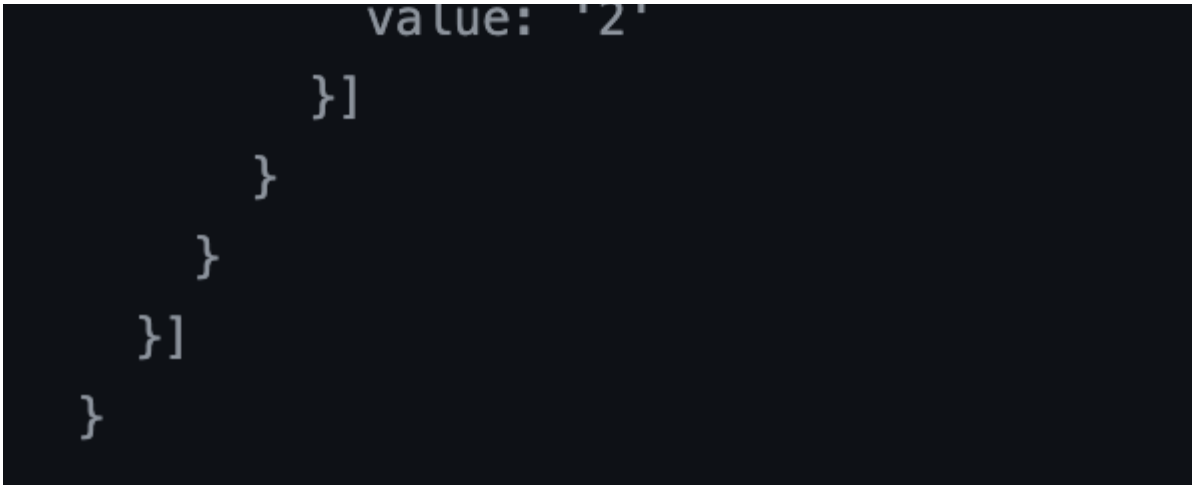
```
      // 下面我们在原来的 `CallExpression` 节点上定义一个新的 context，
      // 它是 expression 中 arguments 这个数组的引用，我们可以向其中放入参数。
      node._context = expression.arguments;

      // 之后我们将检查父节点是否是`CallExpression`类型
      // 如果不是的话
      if (parent.type !== 'CallExpression') {

        // 我们将用`ExpressionStatement`来包裹`CallExpression`节点
        // 这么做是因为单独存在的 `CallExpressions` 在 JavaScript 中也可以被当做是声明语句。
        //
        // 比如 `var a = foo()` 与 `foo()`，后者既可以当作表达式给某个变量赋值，
        // 也可以作为一个独立的语句存在
        expression = {
          type: 'ExpressionStatement',
          expression: expression,
        };
      }

      // 最后我们把我们的`CallExpression`(可能有包裹)放到父节点的上下文中去
      parent._context.push(expression);
    },
  }
```

3.3 代码生成

编译器的最后一个阶段是代码生成，这个阶段做的事情有时候会和转换（transformation）重叠，但是代码生成最主要的部分还是根据 AST 来输出代码。代码生成有几种不同的工作方式，有些编译器将会重用之前生成的 token，有些会创建独立的代码表示，以便于线性地输出代码。但是接下来我们还是着重于使用之前生成好的 AST。

根据前面的这几步骤，我们已经得到了我们新的 AST 树：

```
{
  type: 'Program',
  body: [{
    type: 'ExpressionStatement',
    expression: {
      type: 'CallExpression',
      callee: {
        type: 'Identifier',
        name: 'add'
      },
      arguments: [{
        type: 'NumberLiteral',
        value: '2'
      }, {
        type: 'CallExpression',
        callee: {
          type: 'Identifier',
          name: 'subtract'
        },
        arguments: [{
          type: 'NumberLiteral',
          value: '4'
        }, {
          type: 'NumberLiteral',
```



接下来将调用代码生成器将递归的调用自己来打印树的每一个节点，最后输出一个字符串。

```
function codeGenerator(node) {

  // 还是按照节点的类型来进行分解操作
  switch (node.type) {

    // 如果我们有`Program`节点，我们将映射`body`中的每个节点
    // 并且通过代码生成器来运行他们，用换行符将他们连接起来
    case 'Program':
      return node.body.map(codeGenerator)
        .join('\n');

    // 对于`ExpressionStatement`，我们将在嵌套表达式上调用代码生成器，并添加一个分号
    case 'ExpressionStatement':
      return (
        codeGenerator(node.expression) +
        ';' // << 这是因为保持代码的统一性（用正确的方式编写代码）
      );

    // 对于`CallExpression`我们将打印`callee`，新增一个左括号
    // 然后映射每一个`arguments`数组的节点，并用代码生成器执行，每一个节点运行完之后加上逗号
    // 最后增加一个右括号
    case 'CallExpression':
      return (
        codeGenerator(node.callee) +
        '(' +
        node.arguments.map(codeGenerator)
          .join(', ') +
        ')'
      );
```

```
    // 对于`Identifier`直接返回`node`的名字就好.
    case 'Identifier':
      return node.name;

    // 对于`NumberLiteral`直接返回`node`的值就好.
    case 'NumberLiteral':
      return node.value;

    // 对于`StringLiteral`，在`node`的值的周围添加双引号.
    case 'StringLiteral':
      return '"' + node.value + '"';

    // 如果没有匹配到节点的类型，就抛出异常
    default:
      throw new TypeError(node.type);
  }
}
```

经过代码生成之后我们就得到了这样的 JS 字符串：`add(2, subtract(4, 2));` 也就代表我们的编译过程是成功的！

四、结语

以上就是编写一个 LISP 到 JS 编译器的全过程，逐行中文注释的完整代码地址：

> https://github.com/521xueweihan/OneFile/blob/main/src/javascript/the-super-tiny-compiler.js

那么，今天学到的东西哪里会用到呢？众所周知的 Vue 和 React 虽然写法上有所区别，但是“殊途同归”都是通过 AST 转化的前端框架。这中间最重要的就是转换 AST，它是非常“强大”且应用广泛，比较常见的使用场景：

- IDE 的错误提示、代码高亮，还可以帮助实现代码自动补全等功能
- 常见的 Webpack 和 rollup 打包（压缩）

AST 被广泛应用在编译器、IDE 和代码优化压缩上，以及前端框架 Vue、React 等等。虽然我们并不会常常与 AST 直接打交道，但它总是无时无刻不陪伴着我们。

当然啦！看完文章不一定算真正了解了，所有学习过程都离不开动手实践，或许实践过程中你也会有不一样的理解。实践方法十分简单：只需打开浏览器的“开发者模式” ——> 进入控制台（console）——> 复制/粘贴代码，就可以直接运行看到结果了！

以上就是本文的所有内容 本文只能算粗略带大家了解一下编译器迷你的样子，如果有不同的见解，欢迎评论区留言互动，共同进步呀！