主要实现的功能有：

用空格和换行符替代花括号、冒号和分号；

支持选择器的嵌套组合；

支持以“$”符号开头的变量定义和使用。

如果你对这种风格不是很熟悉也没关系，通过下面这个例子你就能很快明白。

目标 CSS 代码，为 5 条样式规则。第 1 条和第 5 条样式规则是最简单的，使用 1 个选择器，定义了 1 条样式属性；第 2 条规则多用了一个标签选择器，样式属性值为多个字符串组成；第 3 条规则使用了类选择器；第 4 条规则增加了属性选择器，并且样式属性增加为 2 条。

div {color:darkkhaki;}

div p {border:1px solid lightgreen;}

div .a-b {background-color:lightyellow;}

div .a-b [data] {padding:15px;font-size:12px;}

.d-ib {display:inline-block;}

再来看看“源代码”，首先声明了两个变量，然后通过换行缩进定义了上述样式规则中的选择器和样式：

$ib inline-block

$borderColor lightgreen

div

  p

    border 1px solid $borderColor

  color darkkhaki

  .a-b

    background-color lightyellow

    [data]

      padding 15px

      font-size 12px

.d-ib

  display $ib

像上面这种强制缩进换行的风格应用非常广泛，比如编程语言 Python、HTML 模板 pug、预处理器 Sass（以“.sass”为后缀的文件）。

这种风格可能有些工程师并不适应，因为缩进空格数不一致就会导致程序解析失败或执行出错。但它也有一些优点，比如格式整齐，省去了花括号等冗余字符，减少了代码量。推荐大家在项目中使用。

编译器

对预处理器这种能将一种语言（法）转换成另一种语言（法）的程序一般称之为“编译器”。我们平常所知的高级语言都离不开编译器，比如 C++、Java、JavaScript。

不同语言的编译器的工作流程有些差异，但大体上可以分成三个步骤：解析（Parsing）、转换（Transformation）及代码生成（Code Generation）。

解析

解析步骤一般分为两个阶段：词法分析和语法分析。

词法分析就是将接收到的源代码转换成令牌（Token），完成这个过程的函数或工具被称之为词法分析器（Tokenizer 或 Lexer）。

令牌由一些代码语句的碎片生成，它们可以是数字、标签、标点符号、运算符，或者其他任何东西。

将代码令牌化之后会进入语法分析，这个过程会将之前生成的令牌转换成一种带有令牌关系描述的抽象表示，这种抽象的表示称之为抽象语法树（Abstract Syntax Tree，AST）。完成这个过程的函数或工具被称为语法分析器（Parser）。

抽象语法树通常是一个深度嵌套的对象，这种数据结构不仅更贴合代码逻辑，在后面的操作效率方面相对于令牌数组也更有优势。

可以回想一下，我们在第 06 讲中提到的解析 HTML 流程也包括了这两个步骤。

转换

解析完成之后的下一步就是转换，即把 AST 拿过来然后做一些修改，完成这个过程的函数或工具被称之为转换器（Transformer）。

在这个过程中，AST 中的节点可以被修改和删除，也可以新增节点。根本目的就是为了代码生成的时候更加方便。

代码生成

编译器的最后一步就是根据转换后的 AST 来生成目标代码，这个阶段做的事情有时候会和转换重叠，但是代码生成最主要的部分还是根据转换后的 AST 来输出代码。完成这个过程的函数或工具被称之为生成器（Generator）。

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令牌，有些会创建独立代码

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。

代码生成器必须知道如何“打印”转换后的 AST 中所有类型的节点，然后递归地调用自身，直到所有代码都被打印到一个很长的字符串中。

代码实现

学习了编译器相关知识之后，我们再来按照上述步骤编写代码。

词法分析

在进行词法分析之前，首先要考虑字符串可以被拆分成多少种类型的令牌，然后再确定令牌的判断条件及解析方式。

通过分析源代码，可以将字符串分为变量、变量值、选择器、属性、属性值 5 种类型。但其中属性值和变量可以合并成一类进行处理，为了方便后面语法分析，变量可以拆分成变量定义和变量引用。

由于缩进会对语法分析产生影响（样式规则缩进空格数决定了属于哪个选择器），所以也要加入令牌对象。

因此一个令牌对象结构如下，type 属性表示令牌类型，value 属性存储令牌字符内容，indent 属性记录缩进空格数：

{

  type: "variableDef" | "variableRef" | "selector" | "property" | "value", //枚举值，分别对应变量定义、变量引用、选择器、属性、值

  value: string, // token字符值，即被分解的字符串

  indent: number // 缩进空格数，需要根据它判断从属关系

}

然后确定各种类型令牌的判断条件：

variableDef，以“$”符号开头，该行前面无其他非空字符串；

variableRef，以“$”符号开头，该行前面有非空字符串；

selector，独占一行，该行无其他非空字符串；

property，以字母开头，该行前面无其他非空字符串；

value，非该行第一个字符串，且该行第一个字符串为 property 或 variableDef 类型。

最后再来确定令牌解析方式。

一般进行词法解析的时候，可以逐个字符进行解析判断，但考虑到源代码语法的特殊性——换行符和空格缩进会影响语法解析，所以可以考虑逐行逐个单词进行解析。

词法分析代码如下所示：

function tokenize(text) {

  return text.trim().split(/\n|\r\n/).reduce((tokens, line, idx) => {

    const spaces = line.match(/^\s+/) || ['']

    const indent = spaces[0].length

    const input = line.trim()

    const words = input.split(/\s/)

    let value = words.shift()

    if (words.length === 0) {

      tokens.push({

        type: 'selector',

        value,

        indent

      })

    } else {

      let type = ''

      if (/^\$/.test(value)) {

        type = 'variableDef'

      } else if (/^[a-zA-Z-]+$/.test(value)) {

        type = 'property'

      } else {

        throw new Error(`Tokenize error:Line ${idx} "${value}" is not a vairable or property!`)

      }

      tokens.push({

        type,

        value,

        indent

      })

      while (value = words.shift()) {

        tokens.push({

          type: /^\$/.test(value) ? 'variableRef' : 'value',

          value,

          indent: 0

        })

      }

    }

    return tokens;

  }, [])

}

语法分析

现在我们来分析如何将上一步生成的令牌数组转化成抽象语法树，树结构相对于数组而言，最大的特点是具有层级关系，哪些令牌具有层级关系呢？

从缩进中不难看出，选择器与选择器、选择器与属性都存在层级关系，那么我们可以分别通过 children 属性和 rules 属性来描述这两类层级关系。

要判断层级关系需要借助缩进空格数，所以节点需要增加一个属性 indent。

考虑到构建树时可能会产生回溯，那么可以设置一个数组来记录当前构建路径。当遇到非父子关系的节点时，沿着当前路径往上找到其父节点。

最后为了简化树结构，这一步也可以将变量值进行替换，从而减少变量节点。

所以抽象语法树可以写成如下结构。首先定义一个根节点，在其 children 属性中添加选择器节点，选择器节点相对令牌而言增加了 2 个属性：

rules，存储当前选择器的样式属性和值组成的对象，其中值以字符串数组的形式存储；

children，子选择器节点。

复制{

  type: 'root',

  children: [{

  type: 'selector',

value: string

    rules: [{

      property: string,

      value: string[],

    }],

    indent: number,

    children: []

  }]

}

由于考虑到一个属性的值可能会由多个令牌组成，比如 border 属性的值由“1px” “solid” “$borderColor” 3 个令牌组成，所以将 value 属性设置为字符串数组。

语法分析代码如下所示。首先定义一个根节点，然后按照先进先出的方式遍历令牌数组，遇到变量定义时，将变量名和对应的值存入到缓存对象中；当遇到属性时，插入到当前选择器节点的 rules 属性中，遇到值和变量引用时都将插入到当前选择器节点 rules 属性数组最后一个对象的 value 数组中，但是变量引用在插入之前需要借助缓存对象的变量值进行替换。当遇到选择器节点时，则需要往对应的父选择器节点 children 属性中插入，并将指针指向被插入的节点，同时记得将被插入的节点添加到用于存储遍历路径的数组中：

function parse(tokens) {

  var ast = {

    type: 'root',

    children: [],

    indent: -1

  };

  let path = [ast]

  let preNode = ast

  let node

  let vDict = {}

  while (node = tokens.shift()) {

    if (node.type === 'variableDef') {

      if (tokens[0] && tokens[0].type === 'value') {

        const vNode = tokens.shift()

        vDict[node.value] = vNode.value

      } else {

        preNode.rules[preNode.rules.length - 1].value = vDict[node.value]

      }

      continue;

    }

    if (node.type === 'property') {

      if (node.indent > preNode.indent) {

        preNode.rules.push({

          property: node.value,

          value: []

        })

      } else {

        let parent = path.pop()

        while (node.indent <= parent.indent) {

          parent = path.pop()

        }

        parent.rules.push({

          property: node.value,

          value: []

        })

        preNode = parent

        path.push(parent)

      }

      continue;

    }

    if (node.type === 'value') {

      try {

        preNode.rules[preNode.rules.length - 1].value.push(node.value);

      } catch (e) {

        console.error(preNode)

      }

      continue;

    }

    if (node.type === 'variableRef') {

      preNode.rules[preNode.rules.length - 1].value.push(vDict[node.value]);

      continue;

    }

    if (node.type === 'selector') {

      const item = {

        type: 'selector',

        value: node.value,

        indent: node.indent,

        rules: [],

        children: []

      }

      if (node.indent > preNode.indent) {

        path[path.length - 1].indent === node.indent && path.pop()

        path.push(item)

        preNode.children.push(item);

        preNode = item;

      } else {

        let parent = path.pop()

        while (node.indent <= parent.indent) {

          parent = path.pop()

        }

        parent.children.push(item)

        path.push(item)

      }

    }

  }

  return ast;

}

转换

在转换之前我们先来看看要生成的目标代码结构，其更像是一个由一条条样式规则组成的数组，所以我们考虑将抽象语法树转换成“抽象语法数组”。

在遍历树节点时，需要记录当前遍历路径，以方便选择器的拼接；同时可以考虑将“值”类型的节点拼接在一起。最后形成下面的数组结构，数组中每个元素对象包括两个属性，selector 属性值为当前规则的选择器，rules 属性为数组，数组中每个元素对象包含 property 和 value 属性：

{

 selector: string,

 rules: {

   property: string,

   value: string

 }[]

}[]

具体代码实现如下，递归遍历抽象语法树，遍历的时候完成选择器拼接以及属性值的拼接，最终返回一个与 CSS 样式规则相对应的数组：

function transform(ast) {

  let newAst = [];

  function traverse(node, result, prefix) {

    let selector = ''

    if (node.type === 'selector') {

      selector = [...prefix, node.value];

      result.push({

        selector: selector.join(' '),

        rules: node.rules.reduce((acc, rule) => {

          acc.push({

            property: rule.property,

            value: rule.value.join(' ')

          })

          return acc;

        }, [])

      })

    }

    for (let i = 0; i < node.children.length; i++) {

      traverse(node.children[i], result, selector)

    }

  }

  traverse(ast, newAst, [])

  return newAst;

}

实现方式比较简单，通过函数递归遍历树，然后重新拼接选择器和属性的值，最终返回数组结构。

有了新的“抽象语法数组”，生成目标代码就只需要通过 map 操作对数组进行遍历，然后将选择器、属性、值拼接成字符串返回即可。

具体代码如下：

function generate(nodes) {

  return nodes.map(n => {

    let rules = n.rules.reduce((acc, item) => acc += `${item.property}:${item.value};`, '')

    return `${n.selector} {${rules}}`

  }).join('\n')

}