



第一次Lab:

Tree-Sitter Typescript 语法解析



什么是tree sitter

- 一个通用性的代码解析工具
 - 输入: *JavaScript*写成的语法规则`grammar.js`
 - 输出: 代码语法解析树
- 优势
 - 代码规则配置简单, 配置语言为*JavaScript*
 - 通用性强: 支持几十种语言解析
 - 性能好, 底层代码是*rust*, 速度很快



一、环境配置

1.1 操作系统: Linux

tree-sitter CLI: 0.24.6

1.2 安装Ubuntu

(1) 安装VMware-workstation-Pro-17.5.2 (MacOS为VMware-Fusion-Pro-13.5.2)

<https://softwareupdate.vmware.com/cds/vmw-desktop/ws/17.5.2/23775571/windows/core/VMware-workstation-17.5.2-23775571.exe.tar>

(2) 下载Ubuntu镜像文件

<https://mirrors.aliyun.com/ubuntu-releases/22.04/ubuntu-22.04.5-desktop-amd64.iso>

(3) 安装虚拟机

<https://blog.csdn.net/shenliu128/article/details/127262743>

二、运行Tree-Sitter



2.1 安装tree-sitter环境

- (1) 根据平台下载tree-sitter0.24.6

<https://github.com/tree-sitter/tree-sitter/releases>

- (2) 将下载的.gz文件解压

```
gzip -d tree-sitter-linux-x64.gz
```

- (3) 添加可执行权限

```
chmod +x tree-sitter-linux-x64
```

- (4) 【可选】添加到系统路径

```
sudo mv tree-sitter-linux-x64 /usr/local/bin/tree-sitter
```

- (5) 验证安装

```
tree-sitter --version
```

二、运行Tree-Sitter

2.2 项目设置



(1) 新建并命名文件夹

```
$ mkdir tree-sitter-typescript
```

```
$ cd tree-sitter-typescript
```

(2) 初始化tree-sitter

```
$ tree-sitter init
```

(3) 编译grammar.js, 此后每次更改grammar.js后都要重新运行此命令

```
$ tree-sitter generate
```

(4) 测试

```
$ tree-sitter parse <待测试代码文件  
路径>
```

测试
范例

```
$ tree-sitter parse <待测试代码文件路径>

(program [0, 0] - [5, 1]
  (expression_statement [0, 0] - [0, 6]
    (assignment_expression [0, 0] - [0, 5]
      left: (identifier [0, 0] - [0, 1])
      right: (identifier [0, 4] - [0, 5])))
  (if_statement [1, 0] - [5, 1]
    condition: (parenthesized_expression [1, 3] - [1, 10]
      (binary_expression [1, 4] - [1, 9]
        left: (identifier [1, 4] - [1, 5])
        right: (identifier [1, 8] - [1, 9])))
    consequence: (block [1, 11] - [3, 1]
      (expression_statement [2, 4] - [2, 10]
        (assignment_expression [2, 4] - [2, 9]
          left: (identifier [2, 4] - [2, 5])
          right: (identifier [2, 8] - [2, 9]))))
      alternative: (block [3, 7] - [5, 1]
        (expression_statement [4, 4] - [4, 10]
          (assignment_expression [4, 4] - [4, 9]
            left: (identifier [4, 4] - [4, 5])
            right: (identifier [4, 8] - [4, 9])))))
```

二、运行Tree-Sitter



在线测试

➤ 官网playground: <https://tree-sitter.github.io/tree-sitter/7-playground.html>

Code

TypeScript ▾

☐ Log

☐ Show anonymous nodes

☐ Query

☐ Accessibility

```
1  class foo
2  {
3
4  }
5
6
```

Tree

0.1 ms

```
program [1, 4] - [6, 0]
  class_declaration [1, 4] - [4, 5]
    name: type_identifier [1, 10] - [1, 13]
    body: class_body [2, 4] - [4, 5]
```

选择对应的语言

输入要测试的代码

生成语法树

二、运行Tree-Sitter



源代码:

```
if (a > b){  
  c = a  
} else if (c > d){  
  c = b  
}
```

生成
AST

抽象语法树:

```
program [0, 0] - [5, 0]  
  if_statement [0, 0] - [5, 0]  
    condition: parenthesized_expression [0, 3] - [0, 10]  
      binary_expression [0, 4] - [0, 9]  
        left: identifier [0, 4] - [0, 5]  
        right: identifier [0, 8] - [0, 9]  
      consequence: statement_block [0, 10] - [2, 1]  
        expression_statement [1, 0] - [1, 5]  
          assignment_expression [1, 0] - [1, 5]  
            left: identifier [1, 0] - [1, 1]  
            right: identifier [1, 4] - [1, 5]  
        alternative: else_clause [2, 2] - [5, 0]  
          if_statement [2, 7] - [5, 0]  
            condition: parenthesized_expression [2, 9] - [2, 16]  
              binary_expression [2, 10] - [2, 15]  
                left: identifier [2, 10] - [2, 11]  
                right: identifier [2, 14] - [2, 15]  
            consequence: statement_block [2, 16] - [5, 0]  
              expression_statement [3, 0] - [3, 5]  
                assignment_expression [3, 0] - [3, 5]  
                  left: identifier [3, 0] - [3, 1]  
                  right: identifier [3, 4] - [3, 5]
```

整个程序的根节点

if语句节点

if条件语句节点

“then”代码块节点

“else”代码块节点

赋值语句节点

- 抽象语法树 (Abstract Syntax Tree, 简称AST) 是源代码的树状表示形式。每个节点代表源代码中的一个语法结构 (如表达式、语句、函数等)。
- AST的节点有层次关系, 父节点表示更大的语法结构, 子节点表示更小的语法结构。

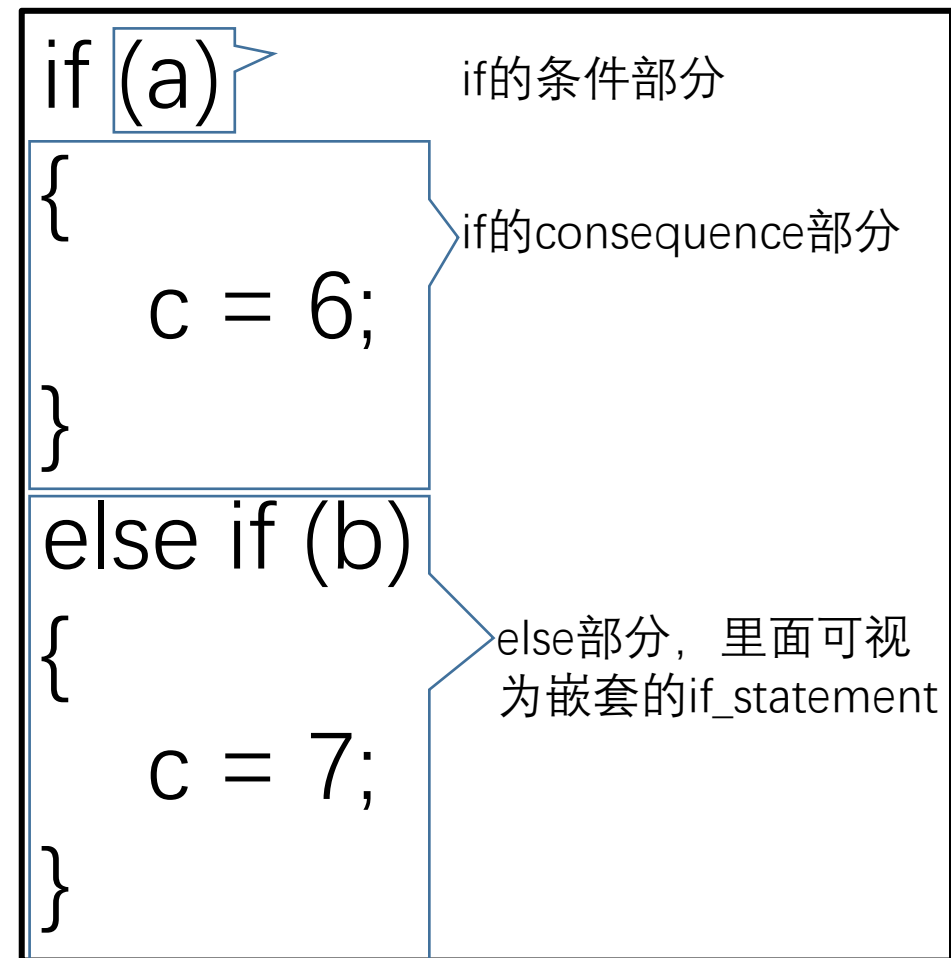


三、任务

- 本学期代码仓库: <https://gitee.com/fdu-ssr/compiler2025spring>
- 第三周任务要求, 能够解析week3case即可
 - 补充grammar.js中variable_declaration语句, 必须支持以下形式
 - 带初始化: `let a = 0`、`const b = 2`
 - 带类型注解: `let c: number`

三、任务

- 第三周任务要求，能够解析week3case即可
 - 补充grammar.js中if_statement语句，必须支持以下形式
 - 单层 if: `if (a) { ... }`
 - 单层 if-else: `if (a) { ... } else { ... }`
 - 多层 else if: `if (a) { ... } else if (b) { ... }`
(解析为嵌套的 if_statement)。



附、Typescript教程



- (1) 教程文档: <https://www.runoob.com/w3cnote/getting-started-with-typescript.html>
- (2) typescript是javascript的超集, 与javascript语法类似, 简单了解即可。

```
function area(shape: string, width: number, height: number) {  
    var area = width * height;  
    return "I'm a " + shape + " with an area of " + area + " cm squared."  
}  
  
document.body.innerHTML = area("rectangle", 30, 15);
```

附、grammar详解



1 基础设置

- **PREC (优先级)**：定义了不同操作符和语句的优先级，用于解析表达式时决定运算顺序。例如，乘法 (`*`) 的优先级高于加法 (`+`)。
- **extras**：指定了在语法解析过程中可以忽略的内容，如空白符和注释。
- **supertypes**：定义了一些通用的类型，例如 `statement` (语句)、`declaration` (声明)、`type` (类型) 和 `comment` (注释)。
- **inline**：内联规则，减少语法树的复杂度。
- **conflicts**：解决语法解析中的冲突，确保语法规则能够正确解析代码。

```
const PREC = {  
    ASSIGN: 1,          // Assignment operators  
    OR: 2,              // ||  
    AND: 3,            // &&  
    BIT_OR: 4,         // |  
    BIT_XOR: 5,        // ^  
    BIT_AND: 6,        // &  
    EQUALITY: 7,       // == !=  
    REL: 8,            // < <= > >=  
    SHIFT: 9,         // << >> >>>  
    ADD: 10,           // + -  
    MULT: 11,          // * / %  
    UNARY: 12,         // Unary operators like -a, !a  
    CALL: 13,          // Function calls  
};
```

附、grammar详解



2 程序结构

- **program**: 程序的顶层结构, 由多个顶层语句组成。
- **_toplevel_statement**: 顶层语句, 可以是声明或其他语句。

```
rules: {  
    program: $ => repeat($_toplevel_statement),  
  
    _toplevel_statement: $ => choice(  
        $.statement,  
        $.function_declaration,  
    ),  
}
```

附、grammar详解



3 类型系统

- **primitive_type**: 基本类型, 如 ``i32``、``u64``、``bool`` 等。
- **array_type**: 数组类型, 支持指定元素类型和长度。

```
primitive_type: _ => choice(  
    'any', 'number', 'boolean', 'string', 'symbol', 'void', 'unknown', 'string', 'never', 'object',  
)
```

附、grammar详解



4 声明

declaration: 声明, 可以是变量声明、常量声明或函数声明。

- **variable_declaration:** 变量声明, 支持可变和不可变变量。
- **const_declaration:** 常量声明, 定义不可变常量。
- **function_declaration:** 函数声明, 定义函数的名称、参数、返回类型和函数体等等。

附、grammar详解

5 语句



- **assignment_statement**: 赋值语句，将表达式的结果赋值给左值。
- **return_statement**: 返回语句，退出函数并返回值。
- **block** 和 **basic_block**: 块语句，包含一组语句，可能带有标签和终结符。
- **scope**: 作用域语句，定义一个新的作用域。
- **debug_statemen**和 **assert_statement**: 调试和断言语句，用于调试和条件检查。

附、grammar详解

6 表达式

- **binary_expression**: 二元表达式, 如加减乘除、逻辑运算等。
- **unary_expression**: 一元表达式, 如取反、取负、引用和解引用。
- **_lvalue**和 **_rvalue**: 左值和右值, 表示可以赋值的变量和表达式。
- **function_call_expression**: 函数调用表达式, 调用函数并传递参数等等。



```
// Expressions
expression: $ => prec.left(PREC.CALL, choice(
    $.function_call_expression,
    $.binary_expression,
    $.unary_expression,
    $_lvalue,
    $_rvalue,
    $.const_expression,
    $.copy_expression,
    $.move_expression,
    $.tuple_access_expression,
    $.tuple_expression,
    $.array_expression,
    $.as_expression,
    $.struct_initialization_expression,
    $.complex_value,
    $.parenthesized_expression,
));
```


附、grammar详解

7 常量

- int、uint、float、bool、bytes、

static_string: 基本的常量类型。

```
// Data Types
region: $ => token(seq("'", /[_a-zA-Z][_a-zA-Z0-9]*/),),

lifetime: $ => token(seq("'", /[_a-zA-Z][_a-zA-Z0-9]*/),),

identifier: $ => /[_a-zA-Z][_a-zA-Z0-9]*/,

int: $ => /\d+/,

uint: $ => /\d+u\d*/,

float: $ => /\d+\.\d+/,

bool: $ => choice('true', 'false'),

bytes: $ => /b".*"/,
```



附、grammar详解



8 注释

- **line_comment**: 单行注释, 以 `//` 开头。
- **block_comment**: 块注释, 以 `/*` 开头, 以 `*/` 结尾。

```
line_comment: _ => token(prec(PREC.COMMENT, seq('//', /.*/))),  
  
block_comment: _ => token(prec(PREC.COMMENT,  
    seq('/*', /[^*]*\*+([^/*][^*]*\*+)*/, '/')  
)),
```

附、grammar详解



9 辅助函数

sep1、commaSep1、commaSep: 用于匹配由特定分隔符分隔的一个或多个规则，简化语法规则的编写。例如，`commaSep1` 用于匹配逗号分隔的列表。

```
/**
 * Creates a rule to match one or more of the rules separated by separator
 *
 * @param {RuleOrLiteral} rule
 *
 * @param {RuleOrLiteral} separator
 *
 * @return {SeqRule}
 */
函数注释 | 行间注释 | 生成单测 | 代码解释 | 调优建议
function sep1(rule, separator) {
  return seq(rule, repeat(seq(separator, rule)));
}
```