简单算术计算器的 C 语言实现

arcasen

https://github.com/arcasen/interpreter-in-cpp

2025 年 10 月 25 日

目录

第一章　编译原理基础

1.1　算术计算器文法

1.1.1　简单计算器的文法

算术计算器的语法可以写成类似 The C Programming Language 附录中的文法:

```
expr ::= expr + term | expr - term
term ::= term * factor | term / factor
factor ::= ( expr ) | integer
integer ::= [0-9]-
```

1.1.2　消除左递归后的文法

左递归不适合自顶向下分析，产生无限递归，不适合手工编写解析器，需要使用 yacc/bison 工具来构造抽象语法树。

1. 左递归消除的通用方法

   基本模式:

   ```
   A → Aα | β
   ```

   消除后:

   ```
   A → βA'
   A' → αA' | ε
   ```

   其中:

   - A 是原非终结符
   - α 是递归部分

- β 是非递归部分
- `A'` 是新引入的非终结符
- ϵ 是空产生式

2. 具体消除步骤

   - 步骤 1：识别左递归模式找出形如 `A → Aα | β` 的产生式
   - 步骤 2：引入新的非终结符为每个左递归规则创建新的非终结符（通常加 `_tail` 或 `'`）
   - 步骤 3：重写文法将左递归转换为右递归

消除左递归的文法如下：

```
expr ::= term expr'
expr' ::= + term expr' | - term expr' | ε
term ::= factor term'
term' ::= * factor term' | / factor term' | ε
factor ::= ( expr ) | integer
integer ::= [0-9]+
```

1.1.3 EBNF 文法

消除左递归的文法略显复杂，以下是将给定的 BNF 文法转化为 EBNF 的结果：

```
expr   ::= term { ( + | - ) term }
term   ::= factor { ( * | / ) factor }
factor ::= ( expr ) | integer
integer ::= [0-9]+
```

这种 EBNF 形式更简洁，并且直接适用于许多解析器生成器。

1.1.4 EBNF 语法规则解释

EBNF（Extended Backus-Naur Form，扩展巴科斯-诺尔范式）是一种用于形式化描述编程语言、协议或数据结构的语法表示法。它是 BNF（Backus-Nor Form）的扩展版本，引入了更多简洁的符号来表示重复、选择和可选元素，使规则更易读和紧凑。

EBNF 的核心思想是将语法规则分解为**非终结符**（non-terminal，通常用粗体或尖括号表示，表示需要进一步定义的符号）和**终结符**（terminal，通常用引号包围的字符串，表示叶子节点）。规则通过 `::=` 符号定义。

1.1.5 优势与注意事项

- **优势**：比 BNF 更紧凑，支持循环和可选结构，易于手写和阅读。
- **注意**：不同工具/语言的 EBNF 实现可能有细微差异（如 ANTLR 用 `?`、`+`、`{}`）。如果用于解析器生成（如 Yacc/Bison），需检查具体语法。
- **扩展**：EBNF 可定义语义动作（如 `{ code }`），但核心是结构描述。

1.2 递归下降分析（Recursive Descent Parsing）

递归下降分析是一种**自顶向下（Top-Down）**的语法分析（Parsing）方法，常用于编译器设计中的词法分析和语法分析阶段。它通过编写一组相互调用的函数（每个函数对应一个非终结符）来实现文法（Grammar）的解析。这种方法简单、直观，且易于实现，尤其适合 **LL(1) 文法**（从左到右扫描、从左到右推导、预测一个符号）。

1.2.1 基本原理

- **核心思想**：为文法中的每个**非终结符**（Non-terminal）编写一个递归函数。该函数尝试从当前输入符号（Token）开始，匹配该非终结符的产生式（Production）。
- **过程**：
  1. 从文法的**起始符号**（Start Symbol）开始调用解析函数。
  2. 函数内部根据当前输入符号的**预测分析表**（Predictive Parsing Table）或简单条件，选择合适的产生式。
  3. 如果产生式是终结符（Terminal），则直接匹配输入；如果是递归非终结符，则递归调用相应函数。
  4. 匹配成功后，继续处理下一个符号；失败则报错（可能回溯，但纯递归下降通常不需回溯）。
- **优点**：
  - 实现简单：直接用函数调用模拟文法规则，无需构建复杂的解析表。
  - 易于调试：每个函数独立，错误定位清晰。
  - 适合手写编译器或解释器。
- **缺点**：
  - 仅适用于**无左递归**（Left-Recursion）和**非歧义**的文法；否则需预处理文法。
  - 效率较低：递归调用可能导致栈溢出（深度过大）。
  - 不支持任意上下文无关文法（Context-Free Grammar）。

第二章　抽象语法树（Abstract Syntax Tree, AST）

抽象语法树（AST）是计算机科学中一种重要的数据结构，用于表示源代码的**抽象语法结构**。它是一种树状表示形式，将编程语言的源代码解析成一个层次化的节点集合，每个节点代表代码中的一个语法元素（如表达式、语句、函数定义等）。与具体语法树（Concrete Syntax Tree）不同，AST 会忽略源代码中的无关细节（如括号、分号、空白等），只保留语义相关的核心结构，便于后续处理。

2.1　AST 的核心特点

- **树状结构**：根节点通常是程序的入口（如整个模块），子节点表示嵌套的语法单元。
- **抽象性**：不包含词法细节（如标识符的拼写），专注于语法规则。
- **用途**：
  - **编译器/解释器**：在编译过程中，词法分析（Lexing）和语法分析（Parsing）后生成 AST，然后进行语义分析、优化和代码生成。
  - **代码工具**：如 ESLint（代码检查）、Babel（JS 转译）、Prettier（格式化）等，都依赖 AST 来遍历和修改代码。
  - **静态分析**：用于代码重构、错误检测或生成文档。

2.2　二叉树与 AST 转换

抽象语法树（AST）通常是多叉树结构，每个节点（如函数定义）可能有多个子节点（如参数列表、函数体）。为了在二叉树中表示和处理 AST，我们可以使用**左手孩子-右兄弟表示法**（Left-Child Right-Sibling），这是一种将多叉树转换为二叉树的经典方法。

这种转换便于在内存中统一处理树结构，尤其在编译器或代码工具中。

第三章　Panic-Mode 错误恢复概述

在编译原理（Compiler Design）中，**Panic-Mode**（恐慌模式）是一种简单的语法错误恢复策略，主要用于**自底向上语法分析**（Bottom-Up Parsing，如 LR 分析器）或**预测分析**（Predictive Parsing）阶段。它旨在处理源代码中的语法错误，而不让整个编译过程崩溃，而是通过 “恐慌” 式跳过无效部分，继续解析剩余代码，从而报告多个错误并保持编译器的鲁棒性。

3.1　核心原理

- **触发条件**：当分析器（Parser）在构建抽象语法树（AST）时，遇到不符合当前产生式（Production Rule）的输入符号（Token）时，即触发错误。
- **恢复机制**：
    1. **丢弃无效输入**：从当前错误位置开始，逐个跳过（Skip）输入 Token，直到遇到一个 “同步点”（Synchronization Token）。同步点通常是：
        - 语句结束符（如分号）。
        - 表达式分隔符（如运算符 `+`、`)`）。
        - 块结束符（如 `}`）。
    2. **重置栈**：在移进-归约分析中，可能弹出栈顶无效符号，直到栈顶匹配同步点。
    3. **继续解析**：到达同步点后，从该点重新开始正常分析，尝试恢复正常流程。
- **优点**：
    - 简单易实现，不需要复杂的 FIRST/FOLLOW 集计算。
    - 能快速报告多个错误，而非在第一个错误处停止。
- **缺点**：
    - 可能跳过过多有效代码，导致 “级联错误”（Cascading Errors），报告过多假阳性错误。
    - 不适合精确错误定位。

示例

假设语法规则为简单表达式：`expr ::= term (('+' | '-')term)*`，输入为无效字符串 `"a + 3 * ("`（`a` 是无效 Token）。

- 正常流程：Lexer 生成 Token 链表 `[ID(a)PLUS NUMBER(3)MULT LPAREN]`。
- Parser 在解析 `term` 时遇到 `ID(a)`（非 `NUMBER` 或 `LPAREN`），触发 Panic-Mode。
- 恢复：跳过 `ID(a)`，直到同步点（如 `PLUS` 或 `MULT`），然后从 `NUMBER(3)` 继续解析。
- 输出：报告 “Syntax Error: Expected number or (at ID”，但继续构建部分 AST（如 `3 * (` 的子树）。

在实际实现中，如之前的 C 代码示例中，Panic-Mode 通过 `while` 循环跳过非同步 Token（如非运算符/括号），直到 `EOL` 或分隔符。

Panic-Mode 是编译器错误处理的基础策略，帮助开发者在开发阶段快速迭代，而非完美语法检查工具。如果您是在实现自定义 Parser（如之前的动态链表/树结构），它特别适合作为入门级垃圾处理机制。

在编译器（特别是像《Crafting Interpreters》项目中描述的解析器）错误处理机制中，“panic-mode 恢复” 是一种编译期错误恢复策略。当解析器检测到语法错误时，它会进入 “panic 模式”（panicMode），报告错误并隔离其影响，避免错误级联导致整个编译过程崩溃。具体来说，解析器会跳过后续的无效标记（tokens），直到找到一个 “同步点”（synchronization point），然后从中恢复正常解析，从而继续尝试编译剩余代码。

3.2 同步点（Sync Point）

同步点是代码中的 “安全恢复位置”，允许解析器在 panic 模式下重新 “同步” 并恢复正常工作。它通常是语句边界或结构化的代码点，例如：- 分号）之后（表示语句结束）。- 语句开始的关键字（如 `if`、`while`、`for`、`return` 等）。- 块结束（如 `{` 和 `}` 的边界）。- 表达式结束符（如 `)` 或 `]`）。

假设代码中有语法错误（如缺少分号）：

```
int main() {
  print("Hello")  // 缺少分号
  print("World");
}
```

- 解析器在第一行检测到错误，进入 panic 模式。
- 它会跳过无效标记，直到遇到下一个同步点（如第二行的 `print` 关键字，表示新语句开始）。

- 然后恢复正常解析第二行，避免报告多余的 “级联错误”（如 “意外的 print”）。

这种设计提高了编译器的鲁棒性和用户体验：错误不会 “传染” 整个文件，而是让开发者看到更多有用的诊断信息。

3.3 算术计算器文法的同步点分析

在给定的算术表达式文法(LL(1) 风格的递归下降解析)中,**同步点(synchronization points)** 是 panic-mode 错误恢复机制的关键，用于在检测到语法错误后跳过无效 token，直到遇到一个 “安全” 位置重新同步解析。该文法没有显式的语句边界（如分），因此同步点主要基于表达式的结构：操作符（优先级边界）、括号匹配和终结符（假设在更大上下文中，如语句结束）。

3.3.1 文法回顾

- `expr ::= term { ( "+" | "-" )term }` // 低优先级：加减
- `term ::= factor { ( "*" | "/" )factor }` // 中优先级：乘除
- `factor ::= "(" expr ")" | integer` // 高优先级：括号或整数
- `integer ::= [0-9]+` // 叶子节点

这是一个右递归的算术表达式文法，优先级从 expr（最低）到 factor（最高）递增。

3.3.2 同步点的定义与选择原则

- **原则**：同步点是当前非终结符的 **FOLLOW 集**（跟随符号）或 **FIRST 集**（预期开始符号）的元素。这些点允许解析器 “重置” 到下一个有效结构，而不产生级联错误。
  - FOLLOW(expr)：通常包括 `)`（括号结束）（语句结束，假设上下文）或 EOL（文件结束）。
  - FOLLOW(term)：包括 `+`、`-`、`)`、`;` 等。
  - FOLLOW(factor)：包括 `*`、`/`、`+`、`-`、`)`、`;` 等。
- 在 panic-mode 中，当错误发生时：
  - 报告错误。
  - 进入 panic 模式，跳过 token 直到匹配同步点。
  - 恢复正常解析。

- **为什么这些点安全？**它们标记表达式的边界或子结构结束，便于从“干净”位置继续（如新操作数或结束）。

3.3.3 具体同步点

基于文法结构，以下是每个非终结符的典型同步点（按优先级层级列出）。这些是解析器在递归下降实现中常用于错误恢复的位置：

表 3.1: 简单算术计算器的同步点分析

非终结符	同步点（token）	解释
expr	`+ - ) EOL`	expr 是顶层，同步到加减操作符（新 term 开始）或括号/语句结束。避免在无效 expr 后继续无限递归。示例：错误后跳到下一个加法项。
term	`* / + - ) EOL`	term 跟随乘除，但可同步到更高层的加减（优先级提升）或结束。示例：除法错误后跳到加法操作。
factor	`( INTEGER * / + - ) EOL`	factor 是最内层，同步到任何操作数开始（括号或数字）或操作符/结束。示例：无效因子后跳到下一个因子或 term 结束。
integer	无需特殊同步（叶子）	作为终结符，如果无效，直接报告并向上层同步。

- **通用同步点**（跨层级）：任何**操作符** (`+ - * /`) 或**结构边界** (`)`) 都是强同步点，因为它们表示优先级变化或子表达式结束。
- **括号特殊处理**：在 factor 中，如果遇到未匹配的 `(`，可同步到对应的 `)` 以“丢弃”无效子 expr。

3.3.4 示例：panic-mode 恢复流程

假设输入：`2 + ( 3 * x )- 5`，其中 `x` 是无效 token（非 integer）。

1. 解析 expr：消费`2`，预期`+`。
2. 消费`+`，解析 term：消费`( 3 *`，然后在`x`处错误（factor 预期 integer 或`(`）。
3. 进入 panic：从 factor 同步，跳过`x`，直到下一个同步点）（factor 的 FOLLOW）。

4. 恢复：消费`)`，继续 expr 的`- 5`。
5. 输出：报告 “意外 token ‘x’ ”，但成功解析剩余`- 5`。

如果这是基于《Crafting Interpreters》的实现，同步点设计参考第 7 章 “Parsing” 中的`pratt`或`recursive descent`错误处理。该文法简单，可直接扩展到语句级别（如添作为 expr 的 FOLLOW）。