算术计算器的实现

arcasen

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

2025 年 10 月 26 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

上述文法存在左递归，不适合自顶向下分析时产生无限递归，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。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，以下是简化后 EBNF 的文法:

```
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。

1.2 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）。

- **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（Panic Mode Error Recovery ）

在编译原理（Compiler Design）中，Panic Mode（恐慌模式）是一种常见的语法错误恢复策略，主要用于语法分析（Parsing）阶段。Panic Mode 是处理编译时语法错误的最简单且最流行的方法之一，当解析器遇到无法处理的无效输入时，通过 “恐慌” 方式快速恢复，继续分析剩余代码，从而避免整个编译过程崩溃。基本思想是一旦检测到错误，语法分析器就 “恐慌” 起来，认为当前所处的状态（比如正在分析一个结构）已经不可靠了。于是丢弃后续的输入符号（Token），直到找到一个同步点（synchronizing token），然后从该点重新开始分析。

现在要分析典型的四则运算表达式文法（递归下降风格）的同步点（synchronizing token）。通常，同步点会选择在一个非终结符的 FIRST 集或一个规则结束后的 FOLLOW 集中的 Token。

3.1　非终结符的 FIRST 集合和 FOLLOW 集合

四则运算表达式文法的非终结符和终结符：

- 非终结符：`expr`, `term`, `factor`。
- 终结符：`+`, `-`, `*`, `/`, `(`, `)`, `digit`，以及输入结束 `$`。（`integer` 可视为终结符 `digit` 或直接处理为数字令牌）

FIRST 集合：

- FIRST(factor) = { `(`, `digit` }
- FIRST(term) = { `(`, `digit` }
- FIRST(expr) = { `(`, `digit` }

FOLLOW 集合：

- FOLLOW(expr) = {), $ }
- FOLLOW(term) = { +, -,), $ }
- FOLLOW(factor) = { *, /, +, -,), $ }

3.2 同步点的选择

在递归下降分析中:- 进入某个非终结符的函数时,如果当前 Token 不在它的 FIRST 集里，可以报错，并跳过 Token 直到遇到 FIRST 或 FOLLOW 中的 Token。
- FOLLOW 集里的 Token 可作为从该非终结符退出的同步点。

在 Panic Mode 中，对于每个非终结符，使用其 FOLLOW 集中的终结符作为同步点（丢弃 Token 直到遇到这些)。这能确保恢复到 “预期结束” 位置，避免过度丢弃有效代码。

表 3.1: 同步点的选择

非终结符	FOLLOW 集（同步点）	解释
expr	{), $}	表达式结束于右括号或文件末尾。
term	{+, -,), $}	项结束于加减运算符、右括号或末尾。
factor	{*, /, +, -,), $}	因子结束于乘除/加减运算符、右括号或末尾。

同步点不一定只能选 FOLLOW 集。FOLLOW 集是推荐和最优的选择，但实际实现中可以灵活调整，甚至使用固定或自定义的同步点集。

第四章　实现一元运算和乘方运算

4.1　浮点数分析

C 语言（C11/C17/C23 标准，§6.4.4.2）中的十进制浮点常量的正则表达式如下：

```
^[+-]?(?:(?:\d+\.\d*|\.\d+)(?:[eE][+-]?\d+)?|\d+[eE][+-]?\d+)(?:[
   fFlL])?$
```

考虑到编写词法分析器比较费时，直接使用库函数 `strtod` 来读取浮点数。

4.2　增加一元运算和乘方运算的文法

参考 Python 语法[1]将文法改写为：

```
expr   ::= term { ( + | - ) term }
term   ::= unary { ( * | / ) unary }
unary  ::= ( + | - ) unary | power
power  ::= factor { '^' unary }  # 一元在这里处理
factor ::= ( expr ) | integer | float
```

要特别注意运算的优先级和结合顺序（乘方运算是右结合的），如果文法不正确将导致表达式解析错误或计算顺序错误，如 `-3^2`，`2^3^4`。

[1] https://docs.python.org/3/reference/grammar.html