表达式计算器设计

arcasen

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

2025 年 10 月 28 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（synchronization points），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（synchronization points)。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。

在编译器设计中的 panic mode 错误恢复策略下, 如果当前 token 位于错误位置（即不匹配当前非终结符的预测集, 导致语法错误), 且它同时是一个同步点（synchronization points，例如 ;、} 等属于同步集的符号），则不需要跳过它。

简要解释

- Panic Mode 的错误恢复流程：检测到错误后，解析器进入 “恐慌” 模式，反复跳过（丢弃）当前 token，直到当前 token 属于同步集（sync set）。同步集是预定义的符号集合，用于 “重置” 解析上下文（如语句结束符、块分隔符），以最小化错误传播并继续解析后续有效代码。
- 当前 token 是同步点的情况：即使这个 token 导致了错误（即它是 “错误位置”），但因为它已经是同步点，跳过循环会立即停止。解析器会保留并消费这个同步 token，将其作为锚点来恢复正常解析。例如：
  - 如果预期一个表达式，但遇到了 ;（同步点），即使 ; 在此位置是错误的，也不会跳过它，而是用它结束当前语句并转向下一个语句的解析。
- 为什么不跳过?

- 跳过同步点会破坏重新同步的机会，导致解析器继续“恐慌”，可能丢弃更多有效代码。
- 目标是“宽容”错误：用同步点快速“跳出”当前错误上下文，而不是过度丢弃输入。
- 这是一种权衡：可能遗漏部分代码，但能让编译器报告错误并继续处理文件其余部分。

伪代码示例（基于 LL(1) 或类似预测解析）：

```
if (当前 token 不匹配预期 FIRST/FOLLOW 集) {  // 检测到错误
    报告错误("Unexpected token: " + 当前 token);

    while (当前 token 不在同步集) {  // 跳过循环
        消费当前 token;  // 丢弃
        读取下一个 token;
    }
    // 循环结束：当前 token 是同步点（即使它是错误位置）
    // 直接消费它作为同步锚点，继续解析下一个产生式
    匹配当前 token;  // 保留并使用它
}
```

- 如果进入 while 时，当前 token 就是同步点，循环体不执行，因此不跳过。

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 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。

3.3 同步点为什么不能（或不宜）用 FIRST 集

在自顶向下解析（如 LL(1) 解析器）的错误恢复机制中，同步点（synchronization points）的主要目的是在遇到错误时 “同步” 输入流，继续解析而非崩溃。通常采用 panic-mode 错误恢复：当栈顶非终结符 A 无法匹配当前输入时，跳过输入符号直到遇到可能跟随 A 后的终结符（即 FOLLOW(A) 中的符号），然后弹出 A 并继续。

3.3.1 为什么用 FOLLOW 集作为同步点?

- 语义匹配：FOLLOW(A) 精确表示 “在 A 后可能出现的终结符”，这符合同步的逻辑——它告诉解析器 “在 A 结束后，输入应该是什么样的符号”，从而安全地

恢复上下文。

- 避免无限循环：使用 FOLLOW 可以确保跳过后，继续从正确的位置解析更高层的结构（如在表达式中，遇到 + 后可以继续 term）。
- 标准实践：在经典编译器设计（如 Dragon Book）中，同步点明确定义为 FOLLOW 集。

3.3.2 为什么不能（或不宜）用 FIRST 集？

- 语义不匹配：FIRST(A) 表示从 A 开始可能出现的首终结符，用于选择产生式（预测分析表中的行选择），而非 “跟随后” 的同步。它描述 A 的开头，不是结束后的预期符号。如果用 FIRST(A) 作为同步点，会导致：
- 错误跳过：例如，栈顶是 expr，但当前输入是 +（不匹配 FIRST(expr)），如果用 FIRST(expr) = { +, −, (, id, … } 同步，会跳过 +（因为它在 FIRST 中），但 + 其实可能是 FOLLOW(expr) 中的符号，用于连接多个 term，导致解析中断。
- 过度匹配：FIRST 集往往较大（包含多种可能起点），容易跳过多余输入，丢失上下文或造成新错误。不安全：在嵌套结构中（如括号内 expr），用 FIRST 可能误判内部/外部边界，导致栈污染或无限递归。

3.4 使用 FIRST 和 FOLLOW 集检查错误

在递归下降分析中，FIRST 集用于预测和选择产生式（决定是否进入某个非终结符的解析），而 FOLLOW 集用于错误恢复和结束验证（同步输入，确保非终结符后跟的符号正确）。同时使用它们能实现更鲁棒的错误检测：FIRST 驱动解析过程，FOLLOW 提供 “安全点” 来恢复，避免级联错误。

同时使用的核心原理：

- **FIRST 的作用**：在函数入口，检查 lookahead ∈ FIRST(A) 以选择产生式或进入空产生式。如果不匹配，立即报错并用 FOLLOW 恢复。
- **FOLLOW 的作用**：
  - 对于可空非终结符，函数结束时验证 lookahead ∈ FOLLOW(A)，检测多余符号。
  - 错误恢复时，跳过输入直到 lookahead ∈ FOLLOW(A)，然后重试解析。

- **结合优势**：FIRST 确保唯一路径（LL(1)），FOLLOW 提供上下文同步（如在表达式中，FOLLOW(E) 包含) 或 $，允许在括号后结束）。

如果文法有多个产生式，使用预测分析表（基于 FIRST/FOLLOW）加速选择。

用 Python 实现了递归下降解析器。该实现针对简单算术表达式文法，并同时使用 FIRST 和 FOLLOW 集进行错误检查和恢复。文法如下：

```
E ::= T {+ T}
T ::= F {* F}
F :: id | ( E )
```

Python 代码：

```python
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens + ['$']  # Add end marker
        self.pos = 0
        self.lookahead = self.tokens[self.pos]
        self.pos += 1
        self.current_non_terminal = None

        # Precomputed FIRST and FOLLOW sets for the grammar:
        # E -> T {+ T}
        # T -> F {* F}
        # F -> id | ( E )
        # Assuming D is replaced by id | ( E ) for a standard
           expression grammar
        self.FIRST = {
            'E': {'id', '('},
            'T': {'id', '('},
            'F': {'id', '('}
        }
        self.FOLLOW = {
            'E': {')', '$'},
            'T': {'+', ')', '$'},
            'F': {'*', '+', ')', '$'}
        }

    def advance(self):
        if self.pos < len(self.tokens):
            self.lookahead = self.tokens[self.pos]
            self.pos += 1

    def match(self, expected, context=""):
        if self.lookahead == expected:
            self.advance()
            return True
```

```
34          else:
35              self.report_error(f"Expected {expected} but got {self.
                    lookahead} ({context})")
36              return False
37
38      def report_error(self, msg):
39          print(f"Syntax error: {msg} at position {self.pos - 1},
                token: {self.lookahead}")
40          return self.sync()
41
42      def sync(self):
43          while self.lookahead not in self.FOLLOW[self.
                current_non_terminal]:
44              if self.lookahead == '$':
45                  print("Unrecoverable error: end of input")
46                  return False
47              self.advance()
48          print(f"Recovery: skipped to FOLLOW symbol {self.lookahead}"
                )
49          return True
50
51      def parse_E(self):
52          self.current_non_terminal = 'E'
53          if self.lookahead not in self.FIRST['E']:
54              if not self.report_error(f"Unexpected token: {self.
                    lookahead}, expected FIRST(E): {self.FIRST['E']}"):
55                  return False
56              return False  # E cannot be empty
57
58          self.parse_T()
59          # Handle {+ T} loop
60          while self.lookahead == '+':
61              self.match('+', "additive operator")
62              self.parse_T()
63
64          if self.lookahead not in self.FOLLOW['E'] and self.lookahead
                != '$':
65              self.report_error(f"Extra token after E: {self.lookahead
                    }, expected FOLLOW(E): {self.FOLLOW['E']}")
66
67          return True
68
69      def parse_T(self):
70          self.current_non_terminal = 'T'
71          if self.lookahead not in self.FIRST['T']:
72              self.report_error(f"Unexpected token: {self.lookahead},
                    expected FIRST(T): {self.FIRST['T']}")
73              return False
```

```
74
75          self.parse_F()
76          # Handle {* F} loop
77          while self.lookahead == '*':
78              self.match('*', "multiplicative operator")
79              self.parse_F()
80
81          # 新增：FOLLOW 检查（结束验证）
82          if self.lookahead not in self.FOLLOW['T'] and self.lookahead
                != '$':
83              self.report_error(f"Extra token after T: {self.lookahead
                   }, expected FOLLOW(T): {self.FOLLOW['T']}")
84
85          return True
86
87      def parse_F(self):
88          self.current_non_terminal = 'F'
89          if self.lookahead not in self.FIRST['F']:
90              self.report_error(f"Unexpected token: {self.lookahead},
                   expected FIRST(F): {self.FIRST['F']}")
91              return False
92
93          if self.lookahead == 'id':
94              self.match('id', "factor identifier")
95          elif self.lookahead == '(':
96              self.match('(', "left parenthesis")
97              self.parse_E()
98              self.match(')', "right parenthesis")
99          else:
100             # Should not reach here due to FIRST check, but handle
101             return False
102
103         # 新增：FOLLOW 检查（结束验证）
104         if self.lookahead not in self.FOLLOW['F'] and self.lookahead
                != '$':
105             self.report_error(f"Extra token after F: {self.lookahead
                   }, expected FOLLOW(F): {self.FOLLOW['F']}")
106
107         return True
108
109     def parse(self):
110         success = self.parse_E()
111         if success and self.lookahead == '$':
112             print("Parse successful")
113             return True
114         else:
115             if self.lookahead != '$':
116                 self.report_error(f"Extra input after end: {self.
```

```
                    lookahead}")
            print("Parse failed")
            return False
```

3.5 避免同一位置多次报错的策略

递归下降解析器（Recursive Descent Parser）是一种自顶向下的 LL(1) 解析方法，每个非终结符对应一个函数。这种设计简单且易于实现，但错误处理是一个常见挑战：当输入 token 序列出现语法错误时，多个递归调用的函数可能会在**同一位置（即同一个 token）**检测到错误，并反复报告相同的错误信息，导致输出冗余和混乱。例如，在解析表达式时，如果遇到意外的 token（如缺少操作数），expr、term、unary 等函数都可能在同一处报错：“Expected operand after ‘+’ ”。

核心问题原因

- 级联错误：错误传播到上层函数，导致多层函数都尝试报告。
- 无状态跟踪：默认实现中，每个函数独立检查当前 token，而不共享错误状态。

解决方案

避免多次报错的关键是错误恢复（Error Recovery）和状态管理。

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

第五章　增加数学函数与常量

5.1　修改后的文法

```
expr   ::= term { ( + | - ) term }
term   ::= unary { ( * | / ) unary }
unary  ::= ( + | - ) unary | power
power  ::= factor { ^ unary }
factor ::= ( expr ) | id ( expr ) | id | integer | float
id     ::= [a-zA-Z]+[0-9]*
```

支持的常量有 `pi`、`e` 和 `phi`，以及计算器上一次计算结果 `ans`。

$$\pi = 3.14159265358979323846264338$$

$$e = 2.71828182845904523536028752$$

$$\phi = 1.61803398874989484820458683$$

计算器不区分大小写，如：`Sin(pi/2)` 与 `sin(Pi/2)` 等价。
`M_PI`、`M_E` 非标准（POSIX 扩展），依赖它容易出问题，可以自己定义。

5.2　支持的数学函数

计算器支持下面的函数用于进行各种数学运算，如三角函数、指数函数、对数函数、幂运算等。按上述文法设计的计算器仅支持含一个参数的函数，多元函数的调用涉及处理参数列表的问题，需要修改文法。

表 5.1: 三角函数（Trigonometric Functions）

函数名	原型	描述
sin	double sin(double x);	正弦函数（弧度）
cos	double cos(double x);	余弦函数（弧度）
tan	double tan(double x);	正切函数（弧度）
asin	double asin(double x);	反正弦函数（返回弧度）
acos	double acos(double x);	反余弦函数（返回弧度）
atan	double atan(double x);	反正切函数（返回弧度）

表 5.2: 双曲函数（Hyperbolic Functions）

函数名	原型	描述
sinh	double sinh(double x);	双曲正弦
cosh	double cosh(double x);	双曲余弦
tanh	double tanh(double x);	双曲正切
asinh	double asinh(double x);	反双曲正弦（C99+）
acosh	double acosh(double x);	反双曲余弦（C99+）
atanh	double atanh(double x);	反双曲正切（C99+）

表 5.3: 指数与对数函数（Exponential and Logarithmic Functions）

函数名	原型	描述
exp	double exp(double x);	指数函数 e^x
log	double log(double x);	自然对数（ln x）
log10	double log10(double x);	常用对数（log10 x）
log2	double log2(double x);	以 2 为底对数（C99+）

表 5.4: 开方（Root Functions）

函数名	原型	描述
sqrt	double sqrt(double x);	平方根
cbrt	double cbrt(double x);	立方根（C99+）

表 5.5: 取整与舍入函数（Rounding Functions）

函数名	原型	描述
ceil	double ceil(double x);	向上取整（到最近整数）
floor	double floor(double x);	向下取整（到最近整数）

表 5.6: 绝对值函数（Absolute Value Function）

函数名	原型	描述
fabs	double fabs(double x);	绝对值

C 语言的标准数学库函数主要定义在 <math.h> 头文件中。大多数函数返回 double 类型的值，并接受 double 类型参数。对于整数或浮点输入，通常通过类型提升转换为 double。
使用时需包含 #include <math.h>，并在链接时添加 -lm（Linux/Unix）。

5.3 文法分析

给定的文法为（假设起始符号为 expr，终结符包括 +、-、*、/、^、(、)、id、integer、float，并引入结束符 $）：

- expr ::= term { (+ | -)term }
- term ::= unary { (* | /)unary }
- unary ::= (+ | -)unary | power
- power ::= factor { ^ unary }
- factor ::= (expr)| id (expr)| id | integer | float

该文法无 ε-产生式(空产生式),无左递归。以下逐步说明计算 FIRST 集和 FOLLOW 集的过程。计算基于标准算法：

5.3.1 FIRST 集计算步骤

FIRST(X) 表示从非终结符 X 导出的串的可能首终结符集合。算法：

1. 对于每个产生式 `A → α`（α 为右部符号序列），将 FIRST(α 的第一个符号)（减去 ε）加入 FIRST(A)。
2. 如果 α 的前缀可空（nullable），则继续向后符号添加 FIRST。
3. 迭代直到固定点（无变化）。
4. 重复和可选部分 `{ β }`（β 为非空）对 FIRST 无影响（因为可选且在首位后），只需关注首位符号的 FIRST。

- **FIRST(factor)**：产生式为 `( expr )`（首 (）、`id ( expr )`（首 id）、`id`（首 id）、`integer`（首 integer）、`float`（首 float）。

  FIRST(factor) = `{ (, id, integer, float }`。

- **FIRST(power)**：`power → factor { ^ unary }`（首位 factor，可选部分不影响）。

  FIRST(power) = FIRST(factor) = `{ (, id, integer, float }`。

- **FIRST(unary)**：产生式为 `+ unary`（首 +）、`-unary`（首 -）、power（首位 power）。

  FIRST(unary) = `{ +, - }` ∪ FIRST(power) = `{ +, -, (, id, integer , float }`。

- **FIRST(term)**：`term → unary { ( * | / )unary }`（首位 unary）。

  FIRST(term) = FIRST(unary) = `{ +, -, (, id, integer, float }`。

- **FIRST(expr)**：`expr → term { ( + | - )term }`（首位 term）。

  FIRST(expr) = FIRST(term) = `{ +, -, (, id, integer, float }`。

5.3.2 FOLLOW 集计算步骤

FOLLOW(A) 表示可能紧跟在 A 后的终结符集合（包括 `$`）。算法：

1. FOLLOW(起始符号) ⊇ `{ $ }`。
2. 对于每个产生式 `B → α A β`，将 FIRST(β)（减去 ε）加入 FOLLOW(A)；若 β 可空，则将 FOLLOW(B) 加入 FOLLOW(A)。
3. 对于可选/重复部分 `{ γ }`（γ 非空），在重复位置后添加 FIRST(γ) 和 FOLLOW(所在非终结符)。
4. 迭代直到固定点。

假设起始符号 `expr`，初始 FOLLOW(expr) = `{ $ }`。

- **FOLLOW(factor)**：factor 仅出现在 `power → factor { ^ unary }`。`{ ^ unary }` 可空，故 FOLLOW(factor) ⊇ FOLLOW(power)；同时 ⊇ FIRST(^ unary) = { ^ }。（后续计算 FOLLOW(power)，此处暂记）。
- **FOLLOW(power)**：power 仅出现在 `unary → power`，故 FOLLOW(power) ⊇ FOLLOW(unary)。在 power 的重复 `{ ^ unary }` 中，unary 后为 ^ 或 power 结束，故 FOLLOW(unary) ⊇ { ^ } ∪ FOLLOW(power)。（循环依赖，后续统一）。
- **FOLLOW(unary)**：unary 出现在：
  - `unary → + unary / - unary`：内层 unary 后为外层 unary，故 FOLLOW(unary) 传播自身。
  - `term → unary { ( * | / )unary }`：首位 unary 后可选部分可空，故 FOLLOW(unary) ⊇ FOLLOW(term)；重复中 unary 后为 * / 或 term 结束，故 ⊇ { *, / } ∪ FOLLOW(term)。
  - power 的重复中：如上，⊇ { ^ } ∪ FOLLOW(power)。（后续计算 FOLLOW(term)）。
- **FOLLOW(term)**：term 出现在：
  - `expr → term { ( + | - )term }`：首位 term 后可选部分可空，故 FOLLOW(term) ⊇ FOLLOW(expr)；重复中 term 后为 + − 或 expr 结束，故 ⊇ { +, - } ∪ FOLLOW(expr)。
    (FOLLOW(expr) 已知)。
- **FOLLOW(expr)**：expr 出现在 factor → (expr) / id (expr)，expr 后均为)，故 FOLLOW(expr) ⊇ {)}。结合起始，FOLLOW(expr) = { $,)}。

现在迭代传播：

- FOLLOW(term) = { +, - } ∪ FOLLOW(expr) = { +, -, $,)}。
- FOLLOW(unary) = { *, / } ∪ { ^ } ∪ FOLLOW(term) = { *, /, ^, +, -, $,)}。
- FOLLOW(power) = FOLLOW(unary) = { +, -, *, /, ^, $,)}。
- FOLLOW(factor) = { ^ } ∪ FOLLOW(power) = { +, -, *, /, ^, $,)}（^ 已包含）。

5.3.3 结果汇总

表 5.7: 文法非终结符的 FIRST 和 FOLLOW 集

非终结符	FIRST 集	FOLLOW 集
expr	{ +, -, (, id, integer, float }	{ $,)}
term	{ +, -, (, id, integer, float }	{ +, -, $,)}
unary	{ +, -, (, id, integer, float }	{ +, -, *, /, ^, $,) }
power	{ (, id, integer, float }	{ +, -, *, /, ^, $,) }
factor	{ (, id, integer, float }	{ +, -, *, /, ^, $,) }

该文法满足 LL(1) 条件（FIRST 集两两不相交，FOLLOW 中无冲突），可用于自顶向下解析。

第六章　REPL 的实现

6.1　简单 REPL 实现

以下是一个用 C 语言实现的简单 REPL（Read-Eval-Print Loop）。它的功能是：

- 持续读取用户输入的一行文本。
- 立即回显（echo）该行。
- 支持通过输入 “exit” 来退出循环（可选退出条件，以使 REPL 更实用）。
- 使用 `fgets` 读取输入行，支持处理空格等字符。

完整代码

```c
#include <stdio.h>
#include <string.h>

#define MAX_LINE 1024  // 最大输入行长度

int main() {
    char line[MAX_LINE];

    printf("欢迎进入简单 REPL！输入 'exit' 退出。\n");

    while (fgets(line, sizeof(line), stdin) != NULL) {
        // 回显用户输入的行（包括换行符）
        printf("Echo: %s", line);

        // 检查是否为退出命令（去除换行符后比较）
        line[strcspn(line, "\n")] = '\0';  // 移除换行符
        if (strcmp(line, "exit") == 0) {
            printf("退出 REPL。\n");
            break;
        }
    }

    return 0;
```

```
}
```

编译和运行

1. 保存代码为 `repl.c`。
2. 编译：`gcc repl.c -o repl`（需要安装 GCC 编译器）。
3. 运行：`./repl`。
4. 示例交互：

```
欢迎进入简单 REPL！输入 'exit' 退出。
Hello World!
Echo: Hello World!

这是一行测试
Echo: 这是一行测试

exit
Echo: exit
退出 REPL。
```

说明

- **读取输入**：`fgets` 函数安全地读取一行（最多 `MAX_LINE` 字符），直到遇到换行符或 EOF。
- **回显**：直接打印 `line`，保留原样（包括尾部的换行符）。
- **退出机制**：使用 `strcmp` 检查是否为 “exit”（忽略大小写可进一步优化，但这里保持简单）。如果不需要退出，可以移除 `if` 块，使其无限循环直到 Ctrl+D (EOF)。
- **潜在改进**：如果需要处理更长的输入，增大 `MAX_LINE`；或使用 `readline` 库支持历史记录（需链接 `-lreadline`）。

6.2 改进 REPL：支持退格和历史记录

基于之前的简单 REPL，添加了以下功能：

- **退格支持**：允许在输入过程中使用退格键（Backspace）删除字符，支持光标移动、插入等基本行编辑。
- **历史记录**：保存之前的输入命令，支持向上/向下箭头键（↑/↓）导航历史记录。历史文件默认为 `~/.repl_history`，最多保存 100 条记录。

这些功能依赖 **GNU Readline** 库（一个标准的 C 行编辑库），它提供了完整的终端交互支持。如果你的系统没有安装，可以通过包管理器安装：

- Ubuntu/Debian: `sudo apt install libreadline-dev`
- macOS: `brew install readline`
- CentOS/RHEL: `sudo yum install readline-devel`

完整代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>

#define HISTFILE ".repl_history"  // 历史文件路径
#define MAX_HIST 100              // 最大历史记录数

int main() {
    char *line;
    int histfile = 0;  // 历史文件扩展名

    // 读取历史文件
    read_history(HISTFILE);
    stifle_history(MAX_HIST);  // 限制历史大小

    printf("欢迎进入增强 REPL! 支持退格、箭头键历史导航。输入 'exit'
        退出。\n");

    while ((line = readline("> ")) != NULL) {
        // 添加空行跳过
        if (*line == '\0') {
            free(line);
            continue;
        }

        // 回显用户输入的行
        printf("Echo: %s\n", line);

        // 添加到历史记录
        if (strlen(line) > 0) {
            add_history(line);
        }

        // 检查退出命令
        if (strcmp(line, "exit") == 0) {
            printf("退出 REPL。\n");
```

```
            break;
        }

        free(line);
    }

    // 保存历史到文件
    write_history(HISTFILE);

    return 0;
}
```

编译和运行

1. 保存代码为 `repl_enhanced.c`。
2. 编译：`gcc repl_enhanced.c -o repl_enhanced -lreadline`（注意链接 `-lreadline`）。
3. 运行：`./repl_enhanced`。
4. 示例交互（在终端中）：

```
欢迎进入增强 REPL! 支持退格、箭头键历史导航。输入 'exit' 退出。
> Hello World!
Echo: Hello World!
> 这是一行测试 [按退格删除“试”字，然后回车]
Echo: 这是一行测
> [按 ↑ 键]  # 会显示上一个命令 "这是一行测"，可编辑后回车
Echo: 这是一行测 (编辑版)
> exit
Echo: exit
退出 REPL。
```

- 历史记录会持久化：下次运行时，↑/↓ 键可导航之前的命令。
- 支持 Tab 补全（如果 Readline 配置了）、Ctrl+C 中断输入等。

说明

- **Readline 集成**：
  - `readline("> ")`：读取一行，支持退格、箭头键、历史导航。
  - `add_history(line)`：将非空输入添加到历史。
  - `read_history` / `write_history`：加载/保存历史文件到用户主目录。
- **限制**：历史最多 100 条（可调整 `MAX_HIST`）。如果输入为空，直接跳过。

- **无 Readline 依赖的替代**：如果不想用外部库，可以手动实现行编辑（使用 `termios` 处理键码），但代码会复杂得多（约 200+ 行）。

这个版本更接近真实 REPL（如 Python 的 Shell）。

6.3 在 Windows 上编译

Readline 是 GNU 项目的一个 C 库，主要针对 Unix/Linux 系统设计，提供命令行编辑（如退格、历史导航）。Windows 的命令行（CMD/PowerShell）有自己的输入机制，但不支持 Readline 的高级功能（如箭头键历史、Tab 补全）。Microsoft 的控制台 API（如 ReadConsole）可以实现基本输入，但不等同于 Readline。集成 Readline 到 C 项目中（基于之前的 REPL 代码）。推荐使用 MinGW 或 Dev-C++ 等工具链，这些支持 GCC 和 GNU 库。

安装 MinGW-w64 或通过 MSYS2。在 MSYS2 终端运行：

```
pacman -S mingw-w64-x86_64-readline mingw-w64-x86_64-ncurses # (
    ncurses 是依赖) 。
```

编译命令：

```
gcc repl_enhanced.c -o repl_enhanced -lreadline -lhistory
```