

# 表达式计算器设计

arcasen

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

2025 年 10 月 28 日



# 目录

<b>第一章</b>	<b>编译原理基础</b>	<b>1</b>
1.1	算术计算器文法 . . . . .	1
1.1.1	简单计算器的文法 . . . . .	1
1.1.2	消除左递归后的文法 . . . . .	1
1.1.3	EBNF 文法 . . . . .	1
1.2	递归下降分析 (Recursive Descent Parsing) . . . . .	2
<b>第二章</b>	<b>抽象语法树 (Abstract Syntax Tree, AST)</b>	<b>3</b>
2.1	AST 的核心特点 . . . . .	3
2.2	二叉树与 AST 转换 . . . . .	3
<b>第三章</b>	<b>Panic Mode 错误恢复 (Panic Mode Error Recovery)</b>	<b>5</b>
3.1	非终结符的 FIRST 集合和 FOLLOW 集合 . . . . .	5
3.2	同步点的选择 . . . . .	6
3.3	同步点为什么不能 (或不宜) 用 FIRST 集 . . . . .	6
3.3.1	为什么用 FOLLOW 集作为同步点? . . . . .	7
3.3.2	为什么不能 (或不宜) 用 FIRST 集? . . . . .	7
3.4	使用 FIRST 和 FOLLOW 集检查错误 . . . . .	7
<b>第四章</b>	<b>实现一元运算和乘方运算</b>	<b>13</b>
4.1	浮点数分析 . . . . .	13
4.2	增加一元运算和乘方运算的文法 . . . . .	13
<b>第五章</b>	<b>增加数学函数与常量</b>	<b>15</b>
5.1	修改后的文法 . . . . .	15
5.2	支持的数学函数 . . . . .	15
5.3	文法分析 . . . . .	17
5.3.1	FIRST 集计算步骤 . . . . .	17
5.3.2	FOLLOW 集计算步骤 . . . . .	18

5.3.3	结果汇总 . . . . .	20
<b>第六章</b>	<b>REPL 的实现</b>	<b>21</b>
6.1	简单 REPL 实现 . . . . .	21
6.2	改进 REPL: 支持退格和历史记录 . . . . .	22
6.3	在 Windows 上编译 . . . . .	25

# 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:

- $\text{FIRST}(\text{factor}) = \{ (, \text{digit} \}$
- $\text{FIRST}(\text{term}) = \{ (, \text{digit} \}$
- $\text{FIRST}(\text{expr}) = \{ (, \text{digit} \}$

FOLLOW 集合:

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

## 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
<b>expr</b>	$\{ ), \$ \}$	表达式结束于右括号或文件末尾。
<b>term</b>	$\{ +, -, ), \$ \}$	项结束于加减运算符、右括号或末尾。
<b>factor</b>	$\{ *, /, +, -, ), \$ \}$	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（或不宣）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  $\text{FOLLOW}(A)$  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  $\text{FIRST}(\text{expr}) = \{ +, -, (, \text{id}, \dots \}$  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  $\text{lookahead} \in \text{FIRST}(A)$  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  $\in$  FOLLOW(A)，检测多余符号。
- 错误恢复时，跳过输入直到 lookahead  $\in$  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：

```
1 class Parser:
2     def __init__(self, tokens):
3         self.tokens = tokens + ['$'] # Add end marker
4         self.pos = 0
5         self.lookahead = self.tokens[self.pos]
6         self.pos += 1
7         self.current_non_terminal = None
8
9         # Precomputed FIRST and FOLLOW sets for the grammar:
10        # E -> T {+ T}
11        # T -> F {* F}
12        # F -> id | ( E )
13        # Assuming D is replaced by id | ( E ) for a standard
14        # expression grammar
15        self.FIRST = {
16            'E': {'id', '('},
17            'T': {'id', '('},
18            'F': {'id', '('}
19        }
20        self.FOLLOW = {
21            'E': {')', '$'},
22            'T': {'+', ')', '$'},
23            'F': {'*', '+', ')', '$'}
24        }
25
26    def advance(self):
27        if self.pos < len(self.tokens):
28            self.lookahead = self.tokens[self.pos]
29            self.pos += 1
```

```

29
30 def match(self, expected, context=""):
31     if self.lookahead == expected:
32         self.advance()
33         return True
34     else:
35         self.report_error(f"Expected {expected} but got {self.
36             lookahead} ({context})")
37         return False
38
39 def report_error(self, msg):
40     print(f"Syntax error: {msg} at position {self.pos - 1},
41         token: {self.lookahead}")
42     return self.sync()
43
44 def sync(self):
45     while self.lookahead not in self.FOLLOW[self.
46         current_non_terminal]:
47         if self.lookahead == '$':
48             print("Unrecoverable error: end of input")
49             return False
50         self.advance()
51     print(f"Recovery: skipped to FOLLOW symbol {self.lookahead}"
52 )
53     return True
54
55 def parse_E(self):
56     self.current_non_terminal = 'E'
57     if self.lookahead not in self.FIRST['E']:
58         if not self.report_error(f"Unexpected token: {self.
59             lookahead}, expected FIRST(E): {self.FIRST['E']}"):
60             return False
61         return False # E cannot be empty
62
63     self.parse_T()
64     # Handle {+ T} loop
65     while self.lookahead == '+':
66         self.match('+', "additive operator")
67         self.parse_T()
68
69     if self.lookahead not in self.FOLLOW['E'] and self.lookahead
70         != '$':
71         self.report_error(f"Extra token after E: {self.lookahead
72             }, expected FOLLOW(E): {self.FOLLOW['E']}")
73
74     return True
75
76 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},
73                             expected FIRST(T): {self.FIRST['T']}")
74         return False
75
76     self.parse_F()
77     # Handle {*} F} loop
78     while self.lookahead == '*':
79         self.match('*', "multiplicative operator")
80         self.parse_F()
81
82     # 新增: FOLLOW 检查 (结束验证)
83     if self.lookahead not in self.FOLLOW['T'] and self.lookahead
84         != '$':
85         self.report_error(f"Extra token after T: {self.lookahead
86                             }, expected FOLLOW(T): {self.FOLLOW['T']}")
87
88     return True
89
90 def parse_F(self):
91     self.current_non_terminal = 'F'
92     if self.lookahead not in self.FIRST['F']:
93         self.report_error(f"Unexpected token: {self.lookahead},
94                             expected FIRST(F): {self.FIRST['F']}")
95         return False
96
97     if self.lookahead == 'id':
98         self.match('id', "factor identifier")
99     elif self.lookahead == '(':
100         self.match('(', "left parenthesis")
101         self.parse_E()
102         self.match(')', "right parenthesis")
103     else:
104         # Should not reach here due to FIRST check, but handle
105         return False
106
107     # 新增: FOLLOW 检查 (结束验证)
108     if self.lookahead not in self.FOLLOW['F'] and self.lookahead
109         != '$':
110         self.report_error(f"Extra token after F: {self.lookahead
111                             }, expected FOLLOW(F): {self.FOLLOW['F']}")
112
113     return True
114
115 def parse(self):
116     success = self.parse_E()
117     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}")
117         print("Parse failed")
118         return False
```





## 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<sup>1</sup>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  $-3^2$ ， $2^{3^4}$ 。

---

<sup>1</sup><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`。

$$\pi = 3.1415926535897932384626434$$

$$e = 2.7182818284590452353602875$$

$$\phi = 1.6180339887498948482045868$$



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
<code>sin</code>	<code>double sin(double x);</code>	正弦函数 (弧度)
<code>cos</code>	<code>double cos(double x);</code>	余弦函数 (弧度)
<code>tan</code>	<code>double tan(double x);</code>	正切函数 (弧度)
<code>asin</code>	<code>double asin(double x);</code>	反正弦函数 (返回弧度)
<code>acos</code>	<code>double acos(double x);</code>	反余弦函数 (返回弧度)
<code>atan</code>	<code>double atan(double x);</code>	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
<code>sinh</code>	<code>double sinh(double x);</code>	双曲正弦
<code>cosh</code>	<code>double cosh(double x);</code>	双曲余弦
<code>tanh</code>	<code>double tanh(double x);</code>	双曲正切
<code>asinh</code>	<code>double asinh(double x);</code>	反双曲正弦 (C99+)
<code>acosh</code>	<code>double acosh(double x);</code>	反双曲余弦 (C99+)
<code>atanh</code>	<code>double atanh(double x);</code>	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
<code>exp</code>	<code>double exp(double x);</code>	指数函数 $e^x$
<code>log</code>	<code>double log(double x);</code>	自然对数 ( $\ln x$ )
<code>log10</code>	<code>double log10(double x);</code>	常用对数 ( $\log_{10} x$ )
<code>log2</code>	<code>double log2(double x);</code>	以 2 为底对数 (C99+)

表 5.4: 开方 (Root Functions)

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

表 5.5: 取整与舍入函数 (Rounding Functions)

函数名	原型	描述
<code>ceil</code>	<code>double ceil(double x);</code>	向上取整 (到最近整数)
<code>floor</code>	<code>double floor(double x);</code>	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
<code>fabs</code>	<code>double fabs(double x);</code>	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  $\epsilon$ -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  $A \rightarrow \alpha$  ( $\alpha$  为右部符号序列), 将  $\text{FIRST}(\alpha$  的第一个符号) (减去  $\epsilon$ ) 加入  $\text{FIRST}(A)$ 。
2. 如果  $\alpha$  的前缀可空 (nullable), 则继续向后符号添加  $\text{FIRST}$ 。
3. 迭代直到固定点 (无变化)。
4. 重复和可选部分  $\{ \beta \}$  ( $\beta$  为非空) 对  $\text{FIRST}$  无影响 (因为可选且在首位后), 只需关注首位符号的  $\text{FIRST}$ 。

- **$\text{FIRST}(\text{factor})$ :** 产生式为  $(\text{expr})$  (首  $($ )、 $\text{id}(\text{expr})$  (首  $\text{id}$ )、 $\text{id}$  (首  $\text{id}$ )、 $\text{integer}$  (首  $\text{integer}$ )、 $\text{float}$  (首  $\text{float}$ )。

$\text{FIRST}(\text{factor}) = \{ (, \text{id}, \text{integer}, \text{float} \}$ 。

- **$\text{FIRST}(\text{power})$ :**  $\text{power} \rightarrow \text{factor} \{ ^ \text{unary} \}$  (首位  $\text{factor}$ , 可选部分不影响)。

$\text{FIRST}(\text{power}) = \text{FIRST}(\text{factor}) = \{ (, \text{id}, \text{integer}, \text{float} \}$ 。

- **$\text{FIRST}(\text{unary})$ :** 产生式为  $+\text{unary}$  (首  $+$ )、 $-\text{unary}$  (首  $-$ )、 $\text{power}$  (首位  $\text{power}$ )。

$\text{FIRST}(\text{unary}) = \{ +, - \} \cup \text{FIRST}(\text{power}) = \{ +, -, (, \text{id}, \text{integer}, \text{float} \}$ 。

- **$\text{FIRST}(\text{term})$ :**  $\text{term} \rightarrow \text{unary} \{ ( * | / ) \text{unary} \}$  (首位  $\text{unary}$ )。

$\text{FIRST}(\text{term}) = \text{FIRST}(\text{unary}) = \{ +, -, (, \text{id}, \text{integer}, \text{float} \}$ 。

- **$\text{FIRST}(\text{expr})$ :**  $\text{expr} \rightarrow \text{term} \{ ( + | - ) \text{term} \}$  (首位  $\text{term}$ )。

$\text{FIRST}(\text{expr}) = \text{FIRST}(\text{term}) = \{ +, -, (, \text{id}, \text{integer}, \text{float} \}$ 。

### 5.3.2 FOLLOW 集计算步骤

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

1.  $\text{FOLLOW}(\text{起始符号}) \supseteq \{ \$ \}$ 。
2. 对于每个产生式  $B \rightarrow \alpha A \beta$ , 将  $\text{FIRST}(\beta)$  (减去  $\epsilon$ ) 加入  $\text{FOLLOW}(A)$ ; 若  $\beta$  可空, 则将  $\text{FOLLOW}(B)$  加入  $\text{FOLLOW}(A)$ 。
3. 对于可选/重复部分  $\{ \gamma \}$  ( $\gamma$  非空), 在重复位置后添加  $\text{FIRST}(\gamma)$  和  $\text{FOLLOW}(\text{所在非终结符})$ 。
4. 迭代直到固定点。

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

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

现在迭代传播:

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

### 5.3.3 结果汇总

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

```
#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

```
#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
```

