Pratt解析算法：从理论到实现

概述

Pratt解析算法（又称Top-Down Operator Precedence解析）是一种高效的表达式解析技术，由Vaughan Pratt于1973年提出。它通过**优先级驱动**的方式，优雅地处理运算符优先级和结合性问题。

为什么需要Pratt解析?

传统递归下降的痛点

```
// 传统递归下降需要多层函数
parse_expression() → parse_term() → parse_factor() → parse_primary()
```

问题:

- 每个优先级需要独立函数
- 函数调用栈深度大
- 扩展新运算符需要修改多层结构

Pratt解析的优势

- **单函数处理**：所有优先级通过单个函数处理
- **动态控制**：通过参数控制解析深度
- **扩展性强**：添加新运算符只需修改优先级表
- **性能优秀**：减少函数调用开销

核心概念

优先级表

```
typedef struct {
    int left;   // 左结合优先级
    int right;  // 右结合优先级
} Priority;

static const Priority priority[] = {
    {10, 10},  // +, -（左结合）
    {11, 11},  // *, /, %（左结合）
    {12, 11},  // ^（右结合）
    // ...
};
```

结合性处理

- 左结合：`left > right`
- 右结合：`left < right`
- 无结合：`left == right`

算法实现

1. 基础框架

```
typedef enum {
    OP_ADD, OP_SUB, OP_MUL, OP_DIV, OP_POW, OP_EOF
} OpType;

typedef struct {
    OpType op;
    int precedence;
} Token;
```

2. 核心解析函数

```
Expr* parse_expr(Parser* p, int min_prec) {
    // 1. 解析左操作数
    Expr* left = parse_primary(p);

    // 2. 处理运算符
    while (p->token.op != OP_EOF &&
           priority[p->token.op].left >= min_prec) {
        Token op = p->token;
        advance(p);

        // 3. 递归解析右操作数
        Expr* right = parse_expr(p, priority[op.op].right);

        // 4. 构建AST节点
        left = make_binary(op.op, left, right);
    }

    return left;
}
```

3. 完整实现示例

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

typedef enum {
    TK_INT, TK_ADD, TK_SUB, TK_MUL, TK_DIV, TK_POW, TK_EOF, TK_ERR
} TokenType;

typedef struct Token {
    TokenType type;
    int value;
} Token;

typedef struct Expr {
    TokenType op;
    struct Expr *left, *right;
    int value;
} Expr;

// 优先级表
static const int prec_left[] = {0, 10, 10, 20, 20, 30};
static const int prec_right[] = {0, 10, 10, 20, 20, 29}; // ^ 右结合

// 词法分析
Token next_token(const char **s) {
    while (isspace(**s)) (*s)++;

    if (isdigit(**s)) {
        int val = 0;
        while (isdigit(**s)) val = val * 10 + *(*s)++ - '0';
        return (Token){TK_INT, val};
    }

    switch (**s) {
        case '+': (*s)++; return (Token){TK_ADD, 0};
        case '-': (*s)++; return (Token){TK_SUB, 0};
        case '*': (*s)++; return (Token){TK_MUL, 0};
        case '/': (*s)++; return (Token){TK_DIV, 0};
        case '^': (*s)++; return (Token){TK_POW, 0};
```

```
        case '\0': return (Token){TK_EOF, 0};
        default: return (Token){TK_ERR, 0};
    }
}

// Pratt解析器
Expr* parse_pratt(const char **s, int min_prec) {
    Token tok = next_token(s);
    if (tok.type != TK_INT) return NULL;

    Expr* expr = malloc(sizeof(Expr));
    expr->op = TK_INT;
    expr->value = tok.value;
    expr->left = expr->right = NULL;

    while (1) {
        Token op = next_token(s);
        if (op.type == TK_EOF || prec_left[op.type] < min_prec) {
            // 回退token
            *s -= (op.type == TK_ADD || op.type == TK_SUB ||
                   op.type == TK_MUL || op.type == TK_DIV ||
                   op.type == TK_POW) ? 1 : 0;
            break;
        }

        Expr* right = parse_pratt(s, prec_right[op.type]);
        if (!right) break;

        Expr* new_expr = malloc(sizeof(Expr));
        new_expr->op = op.type;
        new_expr->left = expr;
        new_expr->right = right;
        expr = new_expr;
    }

    return expr;
}
```

实际应用案例

1. Go编译器实现

```go
func (p *parser) binary(prec int, x syntax.Expr) syntax.Expr {
    for prec < p.op.prec() {
        op := p.op
        p.next()
        y := p.binary(prec+1, nil)  // Pratt核心
        x = p.binaryExpr(x, op, y)
    }
    return x
}
```

2. Lua实现变种

```c
static BinOpr subexpr (LexState *ls, expdesc *v, int limit) {
    /* 初始化代码 */
    while (op != OPR_NOBINOPR && priority[op].left > limit) {
        /* 处理运算符 */
        nextop = subexpr(ls, &v2, priority[op].right);
    }
}
```

性能对比

解析方式	函数调用数	代码行数	扩展性
传统递归下降	O(n×m)	高	差
Pratt解析	O(n)	中	优
表驱动	O(n)	低	优

- **n**: 表达式长度
- **m**: 优先级层级数

总结

Pratt解析算法通过**优先级参数化**的优雅设计，解决了传统解析器的复杂性问题。它特别适合：

- 需要频繁扩展运算符的语言
- 对性能要求较高的场景
- 教学和学习编译原理

掌握Pratt解析算法，将极大提升你设计和实现语言解析器的能力。