

Read, Write, Now Arithmetic

Part 3 – Making Math Happen



From Numbers to Expressions

The Grammar Behind Arithmetic in TIPS

```
value  → term { (PLUS | MINUS) term }

term  → factor { (MULTIPLY | DIVIDE | MOD | CUSTOM_OPER) factor }

factor → [INCREMENT | DECREMENT | MINUS] primary

primary → FLOATLIT | INTLIT | IDENT |
          ( OPENPAREN value CLOSEPAREN )
```

value → the full math expression

term → multiplication, division, and modulo level

factor → handles unary operators (++, --, -)

primary → the raw piece: literal, variable, or parentheses

Implementing these grammars **exactly as written** automatically enforces proper precedence.

Lexer Updates

Teach TIPS to recognize math and custom operators

PLUS	→ +
MINUS	→ -
MULTIPLY	→ *
DIVIDE	→ /
MOD	→ MOD
CUSTOM_OPER	→ ^^
INCREMENT	→ ++
DECREMENT	→ --

Add each of these patterns to **rules.l**.

Define the new tokens in **lexer.h**.

Add readable names to `tokName()` for the
-t flag.

Verify that comments, strings, and identifiers
still scan correctly.

AST Expansion

This is how I did it...NOT how you MUST do it

```
// Part 3 value type
using Value = variant<int,double>;

// Base for anything that computes a value (maps to value/term/factor/primary)
struct ValueNode {
    virtual void print_tree(ostream& os, string prefix) = 0;
    virtual Value interpret(ostream& out) = 0;
};

// Leaves
struct IntLitNode : ValueNode { int v; /* print_tree, interpret */ };
struct RealLitNode : ValueNode { double v; /* print_tree, interpret */ };
struct IdentNode : ValueNode { string name; /* print_tree, interpret (reads symtab) */ };

// Unary and binary ops
struct UnaryOp : ValueNode { Token op; unique_ptr<ValueNode> sub; /* ... */ };
struct BinaryOp : ValueNode { Token op; unique_ptr<ValueNode> left, right; /* ... */ };

// Statement change (Part 2 → Part 3)
struct AssignStmt : Statement {
    string id;
    unique_ptr<ValueNode> rhs; // was a token/string; now a full expression tree
    void print_tree(ostream& os, string prefix) override;
    void interpret(ostream& out) override; // rhs->interpret(out), store into symtab[id]
};
```

UnaryOp

- `++` / `--` only apply to identifiers.
- Update the symbol table and return the new value.
- Unary – just negates the evaluated result.

BinaryOp

- Implements `+` `-` `*` `/` `MOD` `^^`
- Type behavior:
 - `/` → always **REAL**
 - Any **REAL** operand → **REAL** result
 - `MOD` → only valid on two **INTEGER** operands
 - `^^` → uses `pow(double, double)`

AssignStmt

- Evaluates `rhs->interpret(out)`
- Stores result into existing `int` or `double` slot
- Keeps same narrowing / promotion logic from Part 2

What's the Biggest Difference?

```
// Part 3 value type
using Value = variant<int,double>;
```



```
// Base for anything that computes a value (maps to value/term/factor/primary)
struct ValueNode {
    virtual void print_(ostream& os, string prefix) = 0;
    virtual Value interpret(ostream& out) = 0;
};
```



```
// Leaves
struct IntLitNode : ValueNode { int v; /* print_tree, interpret */ };
struct RealLitNode : ValueNode { double v; /* print_tree, interpret */ };
struct IdentNode : ValueNode { string name; /* print_tree, interpret (reads symtab) */ };
```



```
// Unary and binary ops
struct UnaryOp : ValueNode { Token op; unique_ptr<ValueNode> sub; /* ... */ };
struct BinaryOp : ValueNode { Token op; unique_ptr<ValueNode> left, right; /* ... */ };
```

**Interprets for the remaining grammars
actually have values now and are no
longer void type functions!**

AST Helpers I Made

I found these helpful...

```
using Value = variant<int,double>;
inline map<string,Value> symbolTable;

inline double as_double(const Value& v) {
    return holds_alternative<int>(v) ? static_cast<double>(get<int>(v)) : get<double>(v);
}
inline int as_int_strict(const Value& v) {
    if (!holds_alternative<int>(v)) throw runtime_error("MOD requires INTEGER operands");
    return get<int>(v);
}
```

Parser Expansion

AGAIN - This is only how I chose to do it

```
// value → term { (+/-) term }
unique_ptr<ValueNode> parseValue();

// term → factor { (*//|MOD|^) factor }
unique_ptr<ValueNode> parseTerm();

// factor → [ ++ | -- | - ] primary | primary
unique_ptr<ValueNode> parseFactor();

// primary → FLOATLIT | INTLIT | IDENT | ( value )
unique_ptr<ValueNode> parsePrimary();
```

Grammar = Structure

Each `parseX()` mirrors its grammar rule.
No precedence table needed — the grammar handles it.

Return Type

All return `unique_ptr<ValueNode>` to build the AST.

Operator Roles

- `factor()` → unary (-, ++, --)
- `term()` → *, /, MOD, ^^
- `value()` → +, -

Integration

Assignments and future logic reuse `parseValue()` anywhere an expression is allowed.

Run Your Test Cases

Download and Test Your Code...

× + - ÷

All SEVEN Test Cases
Should Interpret As Expected

Any Questions?

We're ALMOST Done!!!

