Capstone 2.0 — Part 2

Variables, Types & Symbol Table

Goal

Your Part 1 language could print a single message. Now it must do more — it must remember, store, and reuse information, and handle more than one data type correctly.

By the end of Part 2, your compiler should be able to interpret programs like:

```
PROGRAM VARDEMO;
VAR
 X : INTEGER;
 Y : REAL;
BEGIN
 READ(X);
 Y := X;
 WRITE(Y)
END
```

Step 1 — Update the Lexer (rules.l)

Add rules for the new tokens in Part 2 (from lexer.h). You already have: PROGRAM, BEGIN, END, WRITE, IDENT, STRINGLIT, OPENPAREN, CLOSEPAREN, SEMICOLON, EOF, UNKNOWN, WHITESPACE.

Now add rules for:

```
VAR
INTEGER
REAL
READ
ASSIGN      // :=
INTLIT      // whole number (digits only)
FLOATLIT    // real number (digits . digits)
COLON       // :
```

Keep keywords case-sensitive.
INTLIT and FLOATLIT must not overlap.
Verify each with -t.

TIPS Grammar Additions (for Part 2)

Use the following grammar to guide your parser design. The structure and flow must match these productions, but your implementation details are up to you.

```
program      → PROGRAM IDENT ';' block EOF

block        → (declarations) compound

declarations  → VAR decl_list | ε
decl_list     → decl_list decl | decl
decl          → IDENT ':' type ';'
type          → INTEGER | REAL

statements    → statement (';' statement)*
statement     → assign | read | write | compound

assign        → IDENT ASSIGN value
read          → READ '(' IDENT ')'
write         → WRITE '(' (STRINGLIT | IDENT) ')'
compound      → BEGIN statements END

value         → INTLIT | FLOATLIT | IDENT
```

Notes:
- Only CompoundStmt uses BEGIN … END.
- The last statement in any statements sequence has no trailing semicolon.
- The value rule is temporary — real expressions come in Part 3.

Block now holds many statements. Create an abstract Statement base class with print_tree() and interpret(), and derive specific statements for assignment, read, write, and compound statements.

A CompoundStmt is a statement consisting of BEGIN … sequence-of-statements … END. It is not a Block and does not represent a "nested block."

Use unique_ptr everywhere for ownership. Add a symbol table that can store both int and double. It may live in Program or in Block — choose and stay consistent. Keep printing readable for -s (name : TYPE = value).

- **Example idea (not for your code):**

```
struct Statement { virtual void interpret(ostream& out)=0; };
struct Block {
   vector<unique_ptr<Statement>> stmts;
   void interpret(ostream& out) {
     for (auto& s : stmts) s->interpret(out); }};
```

Tip: Each derived Statement will follow the same pattern as your Part 1 Write node — parse it, store its data, interpret it.

Step 3 — Extend the Parser (parser.cpp)

Teach the parser to handle an optional declaration section and multiple statements. You'll reuse your Part 1 parsing pattern but extend it for new statement types.

> ***Suggested organization:***
> ***- parseBlock(): expect BEGIN; optionally handle declarations; then parse the statement sequence; expect END.***
> ***- parseDeclarations(): zero or more lines of IDENT ':' (INTEGER\|REAL) ';'.***
> ***- parseStatement(): dispatch to parseAssign/parseRead/parseWrite/parseCompound as needed.***

Statement sequence rule: statements repeat while each is followed by a semicolon. The last statement in a CompoundStmt has no trailing semicolon — that's what ends the loop before END.

- Example control idea (not full code):

```
do {
   stmts.push_back(parseStatement());
} while (peek() == SEMICOLON && nextTok());
```

Tip: Keep your parse errors clear and specific like Part 1 — format and punctuation matter.

Required Parse Error Messages

Not requiring "exact" messages anymore. Just utilize your expect functions to throw appropriately labeled errors as appropriate.

Step 4: Make the Symbol Table "Real"

Goal (what you'll add)

- Maintain a program-wide **symbol table** of variables using map<string, variant<int,double>>.
- Enforce **"declared before use"** for every variable (program name is not a variable).
- Make READ, WRITE(IDENT), and := assignment actually read/write values in the table.
- Support -s to print the symbol table after interpretation.

Design rules (must follow)

- Declarations use **no commas**: one identifier per line. Examples: INTEGER A; or REAL B;
- Types are only INTEGER and REAL.
- Coercions on assignment:
    - assigning REAL → INTEGER truncates
    - assigning INTEGER → REAL widens

What to change (file by file)

1) ast.h

1. **Create**
   - map<string, variant<int, double>> symbolTable;
2. **Correct necessary interprets**
   - Read: check IDENT exists; read as the **declared type** (**visit the variant**).
   - Write(IDENT): find IDENT in table, **visit it**, and print the current value;
   - Write(STRING) stays as in Part 1.
   - Assign: check IDENT exists, assign value appropriately for type.

---

2) parser.cpp

1. When parsing a declaration (INTEGER X; or REAL Y;):
   - **Reject** duplicates: if already declared.
   - **Insert** identifier into symbol table with appropriate initial value and type.
2. When parsing any variable **use** (not declarations):
   - READ id;, id := ...;, WRITE(id)
   - **Require** that the id has been declared ALWAYS

---

3) driver.cpp

   - You'll need a declaration of extern map<string, variant<int,double>> symbolTable to be able to access it.

2. -s flag:
   - After interpretation, if -s is present, print the symbol table

---

- Using an undeclared IDENT anywhere (READ/WRITE/ASSIGN/Expr) **fails at parse time**
- Declaring the same IDENT twice **fails at parse time**.
- READ into an INTEGER consumes an integer; into a REAL consumes a floating-point number.
- := coercions: int→real widens; real→int truncates.
- WRITE(IDENT) prints the current value; WRITE('string') still prints the string.
- -s prints one line per identifier in **sorted** order (map order is fine):
  NAME : INTEGER = 7 or NAME : REAL = 3.5.

💭 Reflection Prompts

1) How did you represent and distinguish INTEGER vs REAL values?
2) What changed inside Block and why was that necessary?
3) How does your parser decide when variable declarations end?
4) Which part was the hardest to debug, and how did you fix it?