CAPSTONE2.0 – The Truth Hurts : Logic, Loops, and SENIORITIS

It’s the end of the road for TIPS — the part where your programs finally begin to reason, compare, decide, and repeat. In this finale, you’ll give TIPS the essential tools of logic and control. At long last, your code can tell truth from lies, make choices with IF, loop with WHILE, and, when it’s all too much, experience the legendary condition known as SENIORITIS.

Previously in TIPS…

Over the past three parts, you’ve slowly taught TIPS how to grow from a collection of characters into a living, breathing (and occasionally buggy) programming language.

- **Part 1** introduced the basics — a working **lexer** and **parser** that could recognize programs, match keywords, and say *hello world* without panicking.
- **Part 2** brought **variables** and a **symbol table**, letting TIPS remember things instead of shouting values into the void.
- **Part 3** gave TIPS a brain for **expressions** and **arithmetic**, allowing it to calculate, combine, and even write results back to the console.

Now, in **Part 4**, TIPS finally gets to make decisions. You’ll add the ability to **compare values**, **evaluate truth**, **branch with IF**, **loop with WHILE**, and **comment** on its own existential crisis using SENIORITIS.

And if you’ve still got creative energy left, you can go the extra mile with **bonus language skins**, giving your TIPS dialects their own flair — from pirate slang to emoji syntax. (Completely optional!)

New and Expanded Grammars

```
statement  →    assign | compound | read | write | if | while | CUSTOM
if         →    IF  expression  THEN  statement [ ELSE  statement ]
while      →    WHILE  expression  statement
expression →    value [ relOp  value ]
relOp      →    LESSTHAN | GREATERTHAN | EQUALTO | NOTEQUALTO
```

These will need to be added to the appropriate grammars considering precedence of logical operators:

AND

OR

NOT

Truth Values

So far, TIPS has worked purely with numbers.

Now those numbers take on meaning — they can represent **truth**.

In TIPS, truth is stored as an **INTEGER**:

- 1 = true
- 0 = false

However, because expressions may also yield **REAL** values, floating-point rounding can create results that are *almost* zero but not quite. For instance, 1.0 – 0.9999999 produces 0.0000001, which should still count as false.

To handle this, we introduce a **tolerance** known as **EPSILON** — a small positive number defining what "close enough to 0" means.

◆ Defining EPSILON

Each implementation of TIPS should define a constant such as:

```
const double EPSILON = 0.00001;
```

This value establishes the boundary for truth testing:

```
bool isTrue(double x) {  return fabs(x) >= EPSILON;  }
```

If |x| < EPSILON, the value is considered **false**.
If |x| ≥ EPSILON, it is considered **true**.

◆ Choosing a Good EPSILON

You should decide your own EPSILON within a sensible range and document it.

- **Too small** (e.g. 1e-10) → floating-point noise may trigger "true" unexpectedly.
- **Too large** (e.g. 0.1) → small but meaningful results may be lost as "false."

A practical guideline for TIPS:

Minimum acceptable: 1e-6

Maximum acceptable: 1e-3

Typical default: 1e-5

Logical Precedence

Your language now supports the logical operators **NOT**, **AND**, and **OR**.
You decide where they belong in your existing grammar structure, but consider the following standard precedence order:

1. **NOT** (highest — unary negation)
2. **AND** (middle — conjunction)
3. **OR** (lowest — disjunction)

When deciding where to place them:

- Think about which part of your grammar loop (value, term, or factor) they most resemble.
- Consider how you handled PLUS, MULTIPLY, and INCREMENT in Part 3.
- Make sure your recursive descent parser respects this hierarchy without breaking associativity.

Comments

Comments are skipped entirely by the lexer and never reach the parser or AST.

Add to your rules to ignore any comment if it starts with ## and does not contain a newline.

SENIORITIS

The CUSTOM statement is a free space for creativity — it can print a message, perform a small action, or just serve as an Easter egg. When the interpreter encounters this statement, it should print **something meaningful or fun** to the console. Feel free to personalize it — humor and originality encouraged.

Don't forget to add it to your rules!

Rules, AST, Parser

From here, it's the same song and dance as before: update your tokens, expand your grammars, build your nodes, and make your interpreter tell the truth.

Demos

For your final demonstration, you will write and run your **own original program** in TIPS using the full feature set introduced in Part 4.

Your program **must include at least**:

- ✅ **One arithmetic operation** (+, -, *, /, MOD, etc.)
- ✅ **One logic operation** (NOT, AND, or OR)
- ✅ **At least one declared variable**
- ✅ **At least one control structure** — either IF **or** WHILE
- ✅ The **SENIORITIS** statement
- ✅ **Comments** that explain what the program is expected to do

During your demo, you will:

1. **Show your source code** — clearly written and commented.
    - Use ## comments to describe the program's purpose and expected output.
    - Example: ## This program counts from 1 to 5, prints each number, then calls SENIORITIS.
2. **Run your parser** and display a correct **parse tree** (no errors).
3. **Run your interpreter** to show the program executing as described.
4. **Display your symbol table** proving that variables were declared, updated, and resolved correctly.

Creativity Encouraged

Your program can be **as simple or as complex as you like**, as long as it meets the requirements above. You may build a short test program that simply loops, makes a comparison, and ends with SENIORITIS, or create something more imaginative — a mini calculator, a number game, or even a tongue-in-cheek "productivity timer."

Report Prompts

(5 prompts × 20 pts = 100 pts total)

Each student will submit a short written report answering the following five prompts.
The **bolded line** in each section is the **prompt you will be graded on**.
The sentences that follow are **food for thought** — you don't need to answer them directly,
but your response should **address the topic** clearly and completely.

---

1. The Nature of Truth (20 pts)

Explain how TIPS now represents truth and falsehood.
How does your implementation handle floating-point results close to 0 using an **EPSILON**?
What value did you choose for EPSILON and why?
Discuss how this tolerance affects the accuracy and stability of logical comparisons in your interpreter.

---

2. Control Flow and Decision Making (20 pts)

Describe how your new **IF** and **WHILE** statements work in your language.
What steps occur between parsing, evaluating the condition, and executing the branch or loop body?
How does your parser ensure that compound blocks (using BEGIN/END) behave correctly?

---

3. Logical Reasoning and Precedence (20 pts)

Your language now supports NOT, AND, and OR.
How did you decide where to place these operators in your grammar to respect proper precedence?
Why does their order of evaluation matter, and how would changing it alter program behavior?

---

4. The Power of Comments and Customization (20 pts)

Why are comments important in a language like TIPS, especially for your demo program?
How did you implement comments so they are ignored by the parser?
Discuss your CUSTOM statement — what message or behavior did you choose, and what inspired it?

---

5. Reflection — The End of the Journey (20 pts)

Looking back across all four parts, what concept or moment finally made your language "click"?
Which part of this project challenged you the most — tokenizing, grammar design, AST construction, or interpretation — and how did you overcome it?
What did you learn about how real programming languages are built from the ground up?

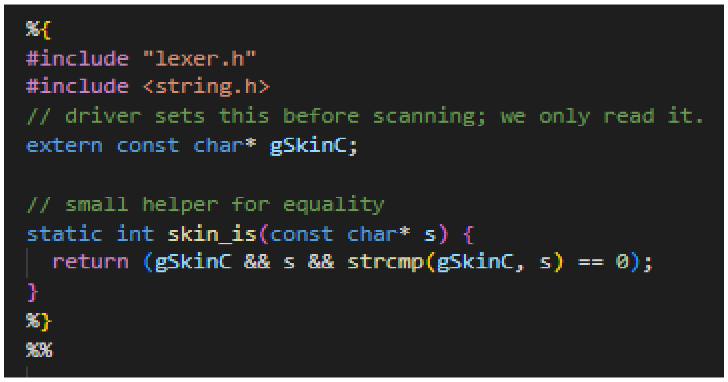
Bonus: Custom Language Skins *(Optional for Extra Credit)*

You've built a working language — now give it *personality*.
For **up to 10 points of extra credit** (added to your lowest test score), you can extend your project to support **custom "skins."** A *skin* lets the same TIPS code be written in a different **dialect or theme** by remapping standard keywords and operators to new words or symbols.

To earn the bonus, write and demo your final program **using your custom skin** — show us your creativity in full color! 🎨

Standard	Pirate Skin	Gen Z Skin	Emoji Skin
PROGRAM	ARRR	YOOO	🏁
WRITE	YELL	POST	🗣️
IF	MAYBE	FR	🤔
WHILE	KEEPDOIN	LOOPIN	🔁
SENIORITIS	NAPTIME	BRB	😴

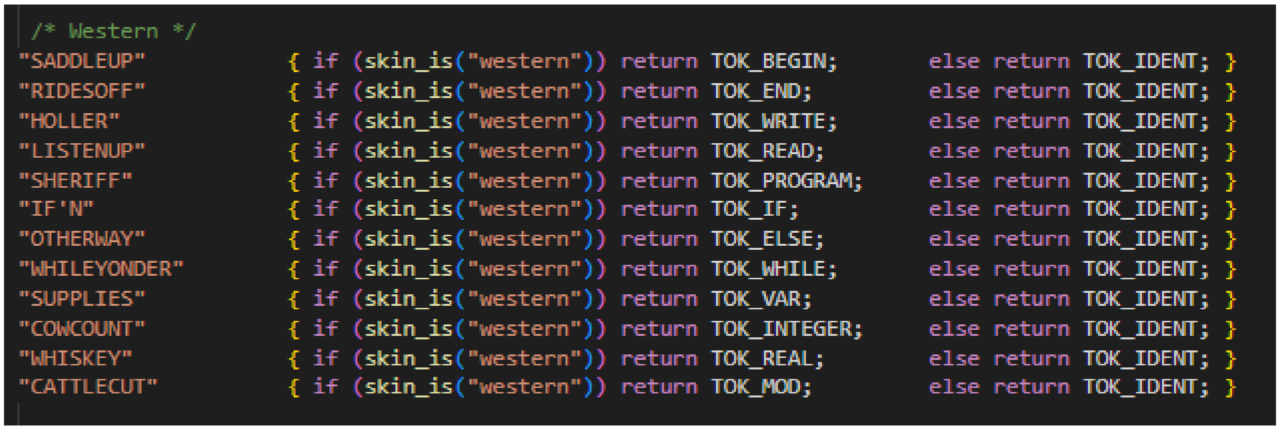
It is as simple as adding the following code in your rules.l file:



```
%{
#include "lexer.h"
#include <string.h>
// driver sets this before scanning; we only read it.
extern const char* gSkinC;

// small helper for equality
static int skin_is(const char* s) {
  return (gSkinC && s && strcmp(gSkinC, s) == 0);
}
%}
%%
```

And as few or as many new rules as you would like structured like these:



```
 /* Western */
"SADDLEUP"          { if (skin_is("western")) return TOK_BEGIN;       else return TOK_IDENT; }
"RIDESOFF"          { if (skin_is("western")) return TOK_END;         else return TOK_IDENT; }
"HOLLER"            { if (skin_is("western")) return TOK_WRITE;       else return TOK_IDENT; }
"LISTENUP"          { if (skin_is("western")) return TOK_READ;        else return TOK_IDENT; }
"SHERIFF"           { if (skin_is("western")) return TOK_PROGRAM;     else return TOK_IDENT; }
"IF'N"              { if (skin_is("western")) return TOK_IF;          else return TOK_IDENT; }
"OTHERWAY"          { if (skin_is("western")) return TOK_ELSE;        else return TOK_IDENT; }
"WHILEYONDER"       { if (skin_is("western")) return TOK_WHILE;       else return TOK_IDENT; }
"SUPPLIES"          { if (skin_is("western")) return TOK_VAR;         else return TOK_IDENT; }
"COWCOUNT"          { if (skin_is("western")) return TOK_INTEGER;     else return TOK_IDENT; }
"WHISKEY"           { if (skin_is("western")) return TOK_REAL;        else return TOK_IDENT; }
"CATTLECUT"         { if (skin_is("western")) return TOK_MOD;         else return TOK_IDENT; }
```

🌈 🧙‍♀️ 🎨 **Creativity Encouraged** 💻 🤠 🏴‍☠️