Strategic_NLP_DSL

A complete end-to-end system that converts natural language trading strategies into a Domain Specific Language (DSL), parses them into an AST, generates executable Python trading logic, and runs a backtest on historical market data.

This project demonstrates how compiler design principles, rule-based NLP, and LLMs can be combined for financial strategy automation.

Features:

1) Convert English trading rules → DSL

2) Supports rule-based (Regex) and LLM-based NLP conversion

3) Custom DSL grammar for trading strategies

4) AST construction using Lark parser

5) Indicator support: SMA, RSI

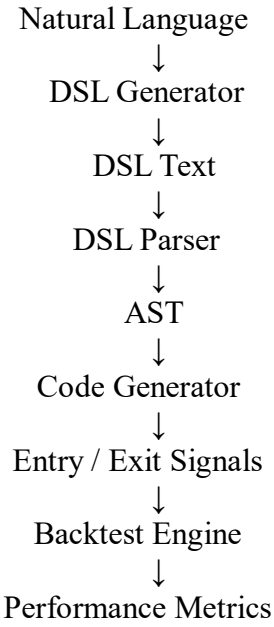
6) Boolean logic: AND / OR

7) Generate vectorized entry & exit signals

8) Run backtests on OHLCV datasets

9) Clear separation of compiler stages

High-Level Pipeline:



Project Structure:

Strategic_NLP_DSL/

```
├── nl_to_dsl.py          # Rule-based NL → DSL (Regex)
├── nl_to_dsl_llm.py      # LLM-based NL → DSL (Gemini)
├── dsl_parser.py         # DSL grammar + AST builder
├── ast_evaluator.py      # AST → Pandas expressions
├── indicators.py         # SMA, RSI implementations
├── code_generator.py     # AST → entry/exit signals
├── backtest.py           # Backtesting engine
├── main.py               # End-to-end demo
├── dataset.csv           # Sample OHLCV data
├── config.py             # API key loader
├── .env                  # API keys (ignored)
├── .gitignore
└── README.md
```

2. DSL Description

The DSL is a declarative rule language designed to describe trading strategies in a precise and unambiguous manner. It supports the definition of entry conditions, exit conditions, boolean logic, comparisons, and technical indicators. The language is intentionally kept minimal, readable, and deterministic to ensure reliable parsing and execution.

Core Structure

Each strategy is composed of two mandatory blocks:

ENTRY:
<condition> [AND <condition>]*

EXIT:
<condition> [AND <condition>]*

Every valid strategy must contain exactly one ENTRY block and exactly one EXIT block.

3. Grammar Specification

Condition Format

A condition follows the structure:

<left_operand> <operator> <right_operand>

Supported Operands

Fields

close
open
high
low
volume

Indicators

sma(close, N)
rsi(close, N)

Supported Operators

<

=
<=
==

Boolean Logic

AND
OR

Nested boolean logic is supported through Abstract Syntax Tree (AST) construction, allowing complex condition combinations while maintaining clear evaluation order.

4. Indicator Syntax

Indicator: SMA
Syntax: sma(close, N)
Description: N-period Simple Moving Average

Indicator: RSI
Syntax: rsi(close, N)
Description: N-period Relative Strength Index

Defaults

If price is mentioned without specifying a field, it defaults to `close`.
If the RSI period is not specified, it defaults to `rsi(close, 14)`.

5. Examples

Example 1: Simple Strategy

Natural Language

Buy when close is above the 20-day moving average and volume is above 1 million.
Exit when RSI falls below 30.

DSL

ENTRY:
close > sma(close,20) AND volume > 1000000

EXIT:
rsi(close,14) < 30

Example 2: Single Condition Strategy

Natural Language

Enter the trade when volume exceeds 500000.
Exit when RSI(2) goes above 70.

DSL

ENTRY:
volume > 500000

EXIT:
rsi(close,2) > 70

Example 3: Boolean Logic Strategy

DSL

ENTRY:
close > sma(close,10) OR volume > 2000000

EXIT:
rsi(close,14) < 40

6. Design Choices Explained

Why DSL?

Natural language is inherently ambiguous and difficult to execute deterministically. A DSL provides precision, repeatability, and strong validation guarantees. It enables deterministic parsing and consistent execution across different strategies.

Why ENTRY and EXIT Blocks?

The ENTRY and EXIT blocks directly mirror the real-world trading lifecycle. This structure simplifies backtest execution logic and enforces clear separation between trade initiation and termination conditions.

Why Minimal Indicators?

Limiting the initial indicator set keeps the grammar simple and easy to reason about. The DSL is designed to be extensible, allowing new indicators to be added without modifying the core language structure.

Why AST-Based Execution?

Using an Abstract Syntax Tree enables a clean separation between parsing and execution. It also allows for future enhancements such as vectorized computation, strategy validation, optimization, and visualization.

7. Error Handling and Validation

Invalid syntax is detected during the DSL parsing stage.
Unsupported indicators or fields raise explicit, descriptive errors.
Only fully valid and well-formed strategies are allowed to proceed to execution.

8. Extensibility

The DSL is designed to be easily extended to support additional features such as:

Time-based lookbacks (e.g., yesterday, last N bars)
Cross events (crosses above, crosses below)

Position sizing rules
Risk management rules such as stop-loss and take-profit

9. Summary

This DSL provides a structured and deterministic way to define trading strategies. It ensures clarity, enforces strong separation of concerns, enables reliable execution, and supports future extensibility while remaining minimal and readable.