Project Specification for Halal Gamble

Tashrique Ahmed, Elyes Laalai

0.1 Introduction

Halal Gamble: The Stock Simulation Game, is a domain-specific programming language (DSL) created to simplify stock market investing. Designed with simplicity and user engagement in mind, it's the perfect starting point for beginners eager to learn and experiment with stock investments without any financial risk. As the complexity of existing tools increases, the need for accessible platforms like Halal Gamble becomes more apparent. This project targets people with different trading experiences and, most importantly, is open to all.

The core of Halal Gamble lies in its interactive simulation. Players embark on a journey starting in 2015, armed with historical data on different stocks from 2010 - 2020. The game challenges players to make informed decisions about buying or selling stocks based on this data, offering new prices each year up to 2020. Players have the freedom to trade or even skip a year's transaction. The real thrill comes at the game's conclusion, where players see the outcome of their decisions in terms of profit or loss. This experience is not just engaging, but also a valuable tool to gauge one's potential in real-world trading, had they invested actual capital in the past.

0.2 Design Principles

Halal Gamble prioritizes simplicity, clarity, and user engagement. The key features include:

- Straightforward and intuitive syntax and semantics for easy comprehension
- Efficient data structures and operations for optimal performance. No operations slower than O(n)
- Ensured safety through robust error handling and parsing failure handling
- Flexibility to choose between stocks and date ranges and customization portfolio management
- Accessible learning through comprehensive game-play instructions
- Type-insensitive, white space-insensitive and natural-language-like syntax
- Features investment-specific functionalities and visualizations, simplifying complex financial concepts
- Tailored for practical application through historical stock market data of 2010 - 2020

0.3 Examples

Sample Program 1

```
dotnet run


start
addcapital(100)
buy(TSLA, 25)
next
next
next
sell(TSLA, 25)
output(bargraph)
exit
```

Sample Program 2

```
dotnet run


start
addcapital(1000)
buy(GOLD, 25)
next
sell(GOLD, 25)
buy(SLVR, 70)
next
sell(SLVR, 20)
buy(SLVR, 50)
next
sell(SLVR,100)
buy(GOLD,50)
buy(GOLD, 50)
output(bargraph)
output(timeseries)
output(portfolio)
exit
```

Sample Program 3

```
dotnet run

start
addcapital(1000)
buy(GOLD, 25)
buy(SLVR, 25)
buy(TSLA, 25)
next
buy(GOLD, 25)
sell(SLVR, 25)
buy(TSLA, 25)
next
buy(GOLD, 25)
buy(SLVR, 25)
sell(TSLA, 25)
next
sell(GOLD, 75)
sell(SLVR, 25)
sell(TSLA, 25)
output(portfolio)
output(timeseries)
exit
```

0.4 Language Concepts

The core concepts in this domain-specific language (DSL) are centered around very basic stock market investment operations like transactions, buy, sell, profit, capital etc. Users need to understand these basic investment terms and actions, as well as how to represent these in the DSL's syntax.

Primitives:

- **Stock Names:** These are primitives in the DSL, representing distinct investment assets like`GOLD` or `TSLA`.
- **Numerical Values:** These include figures like adding capital and investment amounts. All transactions are done in USD.
- **Action Keywords:** Words like 'Buy' and 'Sell' are primitives that represent fundamental operations in the language.

Combining Forms:

- **Sequential Commands:** The combination of various commands (like buying and selling stocks, setting initial capital) in a sequence to form a coherent investment strategy.
- **Investment Operations:** Constructs that combine primitives like stock names and numerical values to define a particular operation, e.g., `buy(GOLD, 50)`.
- **Control Structures:** Commands like `next` or `exit`, control the flow of the simulation, acting like control flow statements in conventional programming languages.
- **Data Visualization Commands:** These combine various data points (such as percentages of investment in different assets) into a cohesive graphical representation, e.g., `output(portfolio)`
- **Portfolio Statements:** Combining different elements of the portfolio (like different stocks and their respective quantities) to generate a comprehensive statement or analysis.

0.5 Formal Syntax

BNF Grammar:

```
<stock> ::= GOLD | SLVR | TSLA
<transactionAmount> ::= <d><transactionAmount> | <d>
<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<command> ::= buy(<stock>, <transactionAmount>, <year>) | sell(<stock>, <transactionAmo

<line> ::= <command> | <output>
<program> ::= <line> | <line><program>

<output> ::= output(<graph>)
<graph> ::= bargraph | timeseries | portfolio
```

Detailed Syntax of Halal Gamble:

- **Add Capital:** Defined using `addcapital(amount)`. This command adds capital budget for the simulation. If you use this command multiple times, it will use the sum of all.
  - Example: `addcapital(100)` allocates a budget of 100 dollars.
- **Stock Transactions:** Executed with `buy(stock, amount)` or `sell(stock, amount)`, allowing users to purchase or sell stocks. If you buy/sell the same stock multiple times in a year, the program will use the total bought/sold amount as the transaction. You cannot make transactions unless you have added capital previously or in the current year. You cannot sell more than you buy. The BNF consists of a key ¡year¿ for all commands, but the user does not have to enter it manually; the program automatically infers the value during runtime.

  - Example: `buy(TSLA,20)` indicates buying 20 dollars of TSLA stock.
  - Example: `sell(GOLD,20)` indicates selling 20 dollars of GOLD stock.

- **Control Commands:** `next`advances the simulation to the next timeframe, while `exit` terminates the simulation. If `next` is used until 2021, the game will exit automatically.

- **Output Visualization:** Defined using `output(type)` to display investment data graphically or in a statement, where type includes `bargraph, timeseries, and portfolio`. If no output is selected, the program will run as usual but will not generate any output.

  - Example: `output(timeseries)` creates a time series plot of the investment result.
  - Example: `output(bargraph)` creates a bargraph of the investment result.
  - Example: `output(portfolio)` generates a portfolio PDF of the investment outcomes.

0.6 Semantics

Primitive Kinds of Values:

- **Stocks:** Represented as strings (e.g., GOLD, SLVR, TSLA), these are fundamental assets in which users can invest.

- **Numerical Values:** Integers and floats are used for transaction amounts, dictating how much of a stock is bought or sold and the simulation's timeline.

- **Years:** Integers representing the specific year of the stock market simulation, crucial for historical data reference.

Actions and Compositional Elements:

- **Transactions (Buy/Sell):** These are the primary actions, combining stock names and numerical values. For instance, `BuyCommand({stock: 'GOLD'; buy: 50; year: 2015})` represents buying $50 worth of GOLD in 2015.

- **Capital Addition:** Involves adding a specified capital amount to the user's portfolio, like `AddCapitalCommand({initial: 'INITIAL'; amount: 1000; year: 2015}).`

- **Output Commands:** These commands generate different forms of output, such as bar graphs or portfolio summaries, based on the current state of the user's investments.

The AST (Abstract Syntax Tree) for this DSL will have several key components represented as algebraic data types:

- **CommandNode:** This node is the primary action node in the AST. It encapsulates the three types of commands that can be performed in the language:

  - **BuyCommand:** Represents a buying transaction. Contains fields for the stock being bought (stock), the amount to buy (buy), and the year of the transaction (year). For example:
    `BuyCommand({stock: "GOLD"; buy: 50.0; year: 2015})`
    This command represents buying $50.0 worth of GOLD stock in the year 2015.
  - **SellCommand:** Represents a selling transaction. Contains fields for the stock being sold (stock), the amount to sell (sell), and the year of the transaction (year). For example:
    `SellCommand({stock: "GOLD"; sell: 30.0; year: 2016})`
    This command represents selling $30.0 worth of GOLD stock in the year 2016.
  - **AddCapitalCommand:** Used for adding capital to the user's portfolio. Contains fields for specifying capital type (initial), the amount of capital being added (amount), and the year of the capital addition (year). For example:
    `AddCapitalCommand({initial: "INITIAL"; amount: 1000; year: 2015})`
    This command adds $1000 to the user's capital in the year 2015.

- **OutputNode:** This node determines the type of output visualization that the program will produce based on the user's investment actions. For example: `OutputNode(Portfolio)`
  An OutputNode can be one of the following types:
  - **Bargraph:** Produces a bar graph visualization.
  - **Timeseries:** Generates a time series plot.
  - **Portfolio:** Creates a detailed view of the user's portfolio.

Note: The control flow is handled internally by the program in order to keep track of the years, thus, no control flow node is used in the AST; the year values get integrated into the CommandNodes.
For a given program, the AST would be structured with a root node representing the entire program and child nodes for each command or output instruction. Each `CommandNode` and `OutputNode` would be processed sequentially, reflecting the order of commands in the user's program.
For instance, an AST for sample program 1 with a series of buy and sell commands followed by a request for a portfolio output might look like this:

```
Program (Root)
|-- CommandNode
|   |-- AddCapitalCommand({initial: "INITIAL"; amount: 100; year: 2015})
|-- CommandNode
|   |-- BuyCommand
|   |   |-- {stock: "TSLA"; buy: 25.0; year: 2015}
|-- CommandNode
|   |-- SellCommand
|   |   |-- {stock: "TSLA"; sell: 25.0; year: 2015}
|-- OutputNode
|   |-- Bargraph
```

AST for sample program 2 would look like -

```
Program (Root)
|-- CommandNode
|   |-- AddCapitalCommand({initial: "INITIAL"; amount: 1000; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "GOLD"; buy: 25.0; year: 2015})
|-- CommandNode
|   |-- SellCommand({stock: "GOLD"; sell: 25.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "SLVR"; buy: 70.0; year: 2015})
|-- CommandNode
|   |-- SellCommand({stock: "SLVR"; sell: 20.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "SLVR"; buy: 50.0; year: 2015})
|-- CommandNode
|   |-- SellCommand({stock: "SLVR"; sell: 100.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "GOLD"; buy: 50.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "GOLD"; buy: 50.0; year: 2015})
|-- OutputNode
|   |-- Bargraph
|-- OutputNode
|   |-- Timeseries
|-- OutputNode
|   |-- Portfolio
```

AST for sample program 3 would look like -

```
Program (Root)
|-- CommandNode
|   |-- AddCapitalCommand({initial: "INITIAL"; amount: 1000; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "GOLD"; buy: 25.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "SLVR"; buy: 25.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "TSLA"; buy: 25.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "GOLD"; buy: 25.0; year: 2015})
|-- CommandNode
|   |-- SellCommand({stock: "SLVR"; sell: 25.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "TSLA"; buy: 25.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "GOLD"; buy: 25.0; year: 2015})
|-- CommandNode
|   |-- BuyCommand({stock: "SLVR"; buy: 25.0; year: 2015})
|-- CommandNode
|   |-- SellCommand({stock: "TSLA"; sell: 25.0; year: 2015})
|-- CommandNode
|   |-- SellCommand({stock: "GOLD"; sell: 75.0; year: 2015})
|-- CommandNode
|   |-- SellCommand({stock: "SLVR"; sell: 25.0; year: 2015})
|-- CommandNode
|   |-- SellCommand({stock: "TSLA"; sell: 25.0; year: 2015})
|-- OutputNode
|   |-- Portfolio
|-- OutputNode
|   |-- Timeseries
```

Program Evaluation:

- **Input Interpretation**
  - **Textual Commands:** The program starts by reading textual commands, each representing an investment action (buy, sell, add capital) or a request for output (e.g., portfolio graph). When the program exits, all the capitalization and whitespace is removed and one large string is passed onto the Parser.
  - **Command Parsing:** Each line in the program is parsed into corresponding AST nodes. For instance, a `buy(GOLD, 50)` command in 2017 is parsed into a BuyCommand node as `BuyCommand({stock: "GOLD"; buy: 50.0; year: 2017})`
- **Sequential Evaluation**
  - **Processing Order:** The evaluation follows the order of commands as they appear in the program. This sequential processing is crucial as each action influences the subsequent state of the investment portfolio.
  - **Capital Management:** The AddCapitalNode is processed first, setting the initial budget for stock transactions. If multiple capital additions are present, they cumulatively affect the total capital.

- **Transaction Handling**
  - **Buy/Sell Execution:** Each BuyCommand or SellCommand node updates the portfolio. BuyCommand increases stock holdings in the portfolio, deducting the corresponding amount from available capital. SellCommand decreases holdings, adding proceeds to the capital based on the stock's value for that year (including profit/loss)
  - **Yearly Context:** Transactions occur within the context of their specified year, affecting the portfolio based on historical stock data for that year.
  - **Cumulative Effect:** Multiple transactions for the same stock in a single year are cumulative. For instance, buying and then selling the same stock within the same year affects the net holding.
  - **Capital Limitation:** Transactions are limited by available capital. Exceeding the capital results in an error.
  - **Selling Limit:** Users can't sell more than they hold. Attempting to do so triggers an error.
- **Output Generation**
  - **Visualization:** Depending on the OutputNode (Bargraph, Timeseries, Portfolio), different visual representations are generated. The visualizations incorporate data from the entire duration of the simulation, reflecting the portfolio's evolution.
  - **Portfolio Summary:** The output includes a summary of gains or losses, highlighting the effectiveness of investment strategies. Users can view their portfolio's performance for each year, illustrating the impact of their decisions.

Detailed Evaluation of Sample Program 1:

1. **Initial State:** The program starts with a default portfolio state of $0
2. **Add Capital:** `AddCapitalNode(100)` – Adds $100 to the investment budget.
3. **First Transaction:** `BuyCommand({stock: 'TSLA', buy: 25, year: 2015})` – Buys $25 worth of TSLA in 2015.
4. **Year Progression:** `next` – Moves the simulation to the next year.
5. **Selling Stocks:** S`SellCommand({stock: 'TSLA', sell: 25, year: 2016})` – Sells $25 worth of TSLA in 2016.
6. **Output:** `OutputNode(BarGraph)` – Generates a bar graph
7. **Program Termination:** `exit` – Ends the program and saves and displays a bargraph

0.7 Remaining Work

For the continued development and enhancement of Halal Gamble, the following areas are identified for future exploration and expansion:

- Expand historical data beyond 2010-2020 to include more diverse financial instruments like bonds, ETFs, and cryptocurrencies.
- Develop an interactive graphical user interface (GUI) for easier navigation and data manipulation.
- Implement real-time market data simulation to provide dynamic and current investment experiences.
- Enhance documentation with more detailed tutorials, examples, and best practices for user assistance.
- Continuously optimize the language's performance in terms of processing speed and memory usage.

These future works are aimed at advancing Halal Gamble into a more comprehensive, user-friendly, and technologically advanced tool for stock market simulation and education.