What is Your Topic?

Topic:
Developing a domain-specific scripting language for defining and executing game mechanics in a turn based pokemon game.

Goal:
Automate complex game behaviors such as triggering effects, updating attributes, and conditional gameplay logic.

Real-World Example: Using a Domain-Specific Scripting Language for Game Mechanics

A domain-specific scripting language (DSL) tailored for game mechanics is pivotal in the game development process.

1. Game Development Context

Modern games, especially character based games like *League of Legends*/ *Pokémon*/ *Overwatch*/ *Valorant*/ *Rainbow 6 Siege,* rely heavily on dynamic game mechanics. These mechanics define how characters interact, how abilities work, and how states like health, damage, or buffs/debuffs are managed during gameplay.

Example Games Using Similar DSL Concepts:

- **League of Legends**: Champions have abilities (like stuns, heals, and shields) that must interact with enemy or ally states dynamically. The scripting of abilities involves defining conditions (e.g., "if enemy health < 30%"), effects (e.g., "deal damage or heal allies"), and triggers (e.g., "activate a shield if health drops below a threshold").
- **Pokémon Games**: Abilities like *Thunderbolt* and *Growl* rely on specific rules to alter the opponent's stats, inflict conditions (e.g., paralysis), or execute turn-based effects.

Dynamic Combat Systems In games like *Elden Ring* or *Final Fantasy*, abilities like *magic spells* or *weapon attacks* often have cascading effects:

- Buffs/Debuffs: A spell might reduce enemy defense for 5 turns while increasing the caster's attack power.
- Conditional Triggers: An ability might deal critical damage if the enemy's defense is below a certain threshold.

A DSL simplifies these mechanics into readable scripts. Instead of writing hundreds of lines of code, developers can define mechanics like:

```
trigger Thunderbolt(Pikachu, Charmander) {
  Charmander.hp = Charmander.hp - (Thunderbolt.power + Pikachu.power)*0.5 ;
  if (random > 50){
    Charmander.state = Thunderbolt.debuff;
  }}
```

Straightforward Techniques for Scripting Game Mechanics

To implement a Domain-Specific Language (DSL) for scripting game mechanics, several techniques and approaches simplify the development and enhance its effectiveness. Below, we outline the key straightforward techniques that can make the scripting language functional, user-friendly, and efficient.

1. Modular Grammar Design

Creating a grammar that separates different aspects of game mechanics into clear, manageable components simplifies both the development and debugging process.

Example:

- Separate grammar rules for defining entities (e.g., Pokémon, moves), actions, conditions, and triggers.

```
pokemon: 'pokemon' IDENTIFIER '{' pokemon_body '}';
move: 'move' IDENTIFIER '{' move_body '}';
trigger_statement: 'trigger' IDENTIFIER '(' IDENTIFIER ',' IDENTIFIER ')' '{' … '}';
```

Each rule focuses on a single responsibility, keeping the DSL maintainable and extensible.

Abstract Syntax Tree (AST) Generation

An AST simplifies the execution of the DSL by converting the parsed text into structured objects. This makes it easier to evaluate conditions, execute actions, and trigger game mechanics.

Steps:

1. Parse the DSL input into tokens using ANTLR or a similar tool.
2. Map the parsed input to an AST structure that represents the logic of the script.
3. Evaluate the AST in a runtime engine (e.g., CodeRunner).

```
trigger ThunderboltEffect(target, user) {
  if (target.hp > 30) {
    target.hp = target.hp - user.power;
  }
}
```

```
TriggerStatement(name="ThunderboltEffect",conditions=[Condition(left=AttributeAccess(entity="target", attribute="hp"),operator=">",right=Int(value=30))],actions=[TriggerAction(target="target",attribute="hp",expression=ArithmeticExpression(left=AttributeAccess(entity="target", attribute="hp"),operator="-",right=AttributeAccess(entity="user", attribute="power")) ) ])
```

Simplified Arithmetic

Game mechanics often involve arithmetic (e.g., target.hp = target.hp - user.power) and conditions (e.g., if (target.hp > 30)). Simplifying how these are evaluated improves the DSL's usability.

- Support operations like +, -, *, /, and nested expressions.

Example:

```
arithmatic: expression ('+' | '-') term | term;

term: term ('/' | '*') factor | factor;

factor: '(' arithmatic ')' | value | IDENTIFIER '.' IDENTIFIER;
```

Solution Overview

The solution involves designing and implementing a **Domain-Specific Language (DSL)** tailored for scripting game mechanics.

1. DSL Design

Objective:

To create a scripting language that allows game developers to:

- Define game entities (e.g., Pokémon, moves).
- Script actions, triggers, and conditions.
- Dynamically update game states.

DSL Features:

- **Entity Definitions**: Define game entities like Pokémon and moves with attributes.
- **Actions**: Allow Pokémon to use moves on other Pokémon, altering their attributes dynamically.
- **Conditions**: Specify logic for abilities or state changes based on game conditions.
- **Triggers**: Define logic for moves, enabling flexible customization of effects.

Example:

pokemon Pikachu { hp: 100; power: 50; } pokemon Charmander { hp: 120; power: 40; } move Growth{hp: 30;} trigger Growth(Charmander, Pikachu) { Charmander.hp = Charmander.hp + Growth.healing; } Charmander uses Growth on Pikachu; status Charmander;

This script:

1. Defines two Pokémon with attributes, Define move Growth
2. Sets up a trigger for the "Growth" ability, healing Charmander.
3. Executes the action, applying the effect.

2. Parsing and Grammar

Objective:

To convert the DSL input into a structured representation using a parser. This is done by defining the grammar and using ANTLR to generate a parser.

Key Grammar Rules:

- **Entity Definitions:**

pokemon: 'pokemon' IDENTIFIER '{' pokemon_body '}';

move: 'move' IDENTIFIER '{' move_body '}';

- **Trigger Definition:**

trigger_statement: 'trigger' IDENTIFIER '(' IDENTIFIER ',' IDENTIFIER ')' '{' ((trigger_action | trigger_condition) ';')* '}';

- **Actions:**

action_statement: IDENTIFIER 'uses' IDENTIFIER 'on' IDENTIFIER;

- **Conditions:**

condition: expression comparison_operator expression (('&&'|'||') condition)*;

Tools:

- **ANTLR: Used to define the grammar and generate the lexer/parser.**
- **Abstract Syntax Tree (AST): Represents the parsed input for further processing.**

3. Abstract Syntax Tree (AST)

Objective:

To transform the parsed input into a tree structure (AST) for evaluation and execution.

Key Components:

- **Entities (Pokémon and Moves):**
    - Represented as nodes with attributes.
    - Example:

Pokemon(name="Pikachu", attributes={"hp": 100, "power": 50})

- **Triggers:**
    - Store conditions and actions for moves.
    - Example:

```
Trigger(name="Growth", user="Charmander", target="Pikachu", actions=[...])
```

- **Conditions:**
    - Handle complex logic like && and ||.
    - Example:

```
Condition(
  left=AttributeAccess(entity="Charmander", attribute="hp"),
  operator=">",
  right=Int(value=50)
)
```

4. Execution Framework

Objective:

To process the AST and execute the defined game mechanics.

Components:

- **Runtime Context:**
    - Stores the game state, including Pokémon and moves.
    - Example:

```
context = {
  "Charmander": {"hp": 120, "power": 40},
  "Pikachu": {"hp": 100, "power": 50}
}
```

- **CodeRunner:**
    - Evaluates triggers, conditions, and actions.
    - Updates the game state dynamically.
    - Example:

```
def visitTriggerAction(self, ctx: TriggerAction):
  target = context[ctx.target]
```

```
    target[ctx.attribute] += ctx.expression.evaluate(context)
```

---

5. Real-Time Game State Updates

Objective:

To apply changes to the game state dynamically based on the DSL script.

Example Flow:

1. **Input Script:**

```
trigger Growth(Charmander, Pikachu) {
    Charmander.hp = Charmander.hp + Growth.healing;
}
Charmander uses Growth on Pikachu;
status Charmander;
```

2. **Parsed and Executed:**
    - Trigger Growth is invoked.
    - Charmander.hp is updated dynamically.
3. **Output:**

```
Status of Pokemon 'Charmander': {hp: 150, power: 40}
```

Pipeline of the Framework

1. **Input:**
    - Game mechanics defined in DSL format.
    - Example:

```
Charmander uses Growth on Pikachu;
```

2. **Parsing:**
    - The script is parsed using ANTLR, generating tokens and a parse tree.
3. **AST Generation:**

- The parse tree is converted into an AST for easier evaluation.

4. **Execution:**
    - The AST is evaluated by the runtime (e.g., CodeRunner), updating the game state.
5. **Output:**
    - Updated game state or error messages if the script is invalid.

The framework pipeline represents the structured flow of how the DSL processes input scripts and converts them into executable game logic. This pipeline ensures that game mechanics, as defined by the DSL, are interpreted correctly, validated, and applied to the game state dynamically.

Here's an in-depth explanation of each phase in the pipeline:

1. Input Phase

Purpose:

To provide the framework with human-readable game mechanic definitions in the form of DSL scripts.

What Happens:

- **Game designers write scripts defining:**
    - Entities: Pokémon, moves, and their attributes.
    - Actions: Pokémon using moves on others.
    - Triggers: Reusable logic for actions, with conditions and effects.
    - Conditions: Logical checks to control behavior (e.g., "if HP < 30").
- **The input script acts as the starting point for the pipeline.**

Example DSL Script:

```
pokemon Pikachu {
  hp: 100;
```

```
    power: 50;
}
pokemon Charmander {
    hp: 120;
    power: 40;
}
Move Growth {
   Healing: 30;
}
trigger Growth(Charmander, Charmander) {
    Charmander.hp = Charmander.hp + Growth.healing;
}
Charmander uses Growth on Pikachu;
status Charmander;
```

2. Parsing Phase

Purpose:

To convert the input script into a structured representation using a grammar-based parser.

What Happens:

- The script is passed through an ANTLR-generated lexer and parser.
- The lexer breaks the input into tokens (e.g., pokemon, hp, 100, etc.).
- The parser uses the grammar to validate the structure of the script and generate a parse tree.

Tools:

- ANTLR: The grammar is defined in a .g4 file, and ANTLR generates the lexer/parser for it.

Output:

- A parse tree that represents the script's structure.

Example Parse Tree: For the script:

Charmander uses Growth on Pikachu;

The parse tree might look like:

(action_statement

IDENTIFIER: "Charmander"

'uses'

IDENTIFIER: "Growth"

'on'

IDENTIFIER: "Pikachu"

)

3. Abstract Syntax Tree (AST) Generation

Purpose:

To convert the parse tree into a more usable Abstract Syntax Tree (AST) for execution.

What Happens:

- The parse tree is traversed, and corresponding AST nodes are created.
- Each node represents a distinct part of the DSL:
    - Entities: Pokémon, moves, triggers.
    - Actions: Operations performed by Pokémon.
    - Conditions: Logical checks controlling behavior.

Tools:

- Custom visitor methods (e.g., visitActionStatement, visitTriggerStatement) convert parse tree nodes into AST nodes.

5. Execution Phase

Purpose:

To evaluate the AST and apply the defined mechanics to the game state.

What Happens:

- The AST is traversed, and each node is executed.
- Entity Definitions:
    - Pokémon and moves are added to the runtime context.
    - Example: Pikachu and Charmander are stored with their attributes.
- **Triggers:**
    - Conditions are checked, and corresponding actions are applied.
    - Example:

Charmander.hp = Charmander.hp + Growth.healing;

This updates Charmander.hp dynamically in the game state.

- **Actions:**
    - Pokémon using moves trigger their respective logic.
    - Example:

Charmander uses Growth on Pikachu;

Executes the Growth trigger, healing Charmander.

Tools:

- CodeRunner: A runtime engine that evaluates AST nodes.
- Example:

```
def visitTriggerAction(self, node, context):
  target = context[node.target]
  target[node.attribute] += node.expression.accept(self, context)
```

6. Output Phase

Purpose:

To display the updated game state or error messages to the user.

What Happens:

- The updated game state is printed after executing the script.
- If errors occur during execution, they are logged.

Example Output:

For the input script:

Charmander uses Growth on Pikachu;

status Charmander;

The output might be:

Status of Pokemon 'Charmander': {hp: 150, power: 40}

How to Define the Grammar?

```
pokemon Pikachu {
   hp: random;
   power: 30;
   type: "electric";
}
status Pikachu
pokemon Charmander {
   hp: 120;
   power: 45;
   type: "fire";
   state: "No";
}
move Thunderbolt {
   power: 40;
   debuff: "paralasis";
   type: "electric";
}
```

```
trigger Thunderbolt (Pikachu, Charmander){
   Charmander.hp = Charmander.hp - (Thunderbolt.power + Pikachu.power)*0.5 ;
   if (random > 50){
      Charmander.state = Thunderbolt.debuff;
   }
}
```

Pikachu uses Thunderbolt on Charmander