Elysium DSL

Introdução

O que é uma DSL?

DSL (*domain-specific-language*) nada mais é do que uma linguagem criada para resolver um problema de domínio muito específico, em vez de se propor a resolver problemas gerais. No caso, essa DSL se propõe a permitir que seja possível definir efeitos de itens de formas mais eficientes

Por que uma DSL para o Elysium?

Hoje em dia, os itens são controlados por planilhas do Google Sheets, e os seus efeitos se encontram em ou em colunas específicas de atributos e status específicos (como colunas de Força, Destreza, Constituição, etc.) ou em colunas de efeitos, com o texto por extenso do efeito (o que o bot não consegue interpretar). Em ambos os casos, temos problemas para gerar novos efeitos, sempre precisaremos adicionar colunas novas nas tabelas para novos satus alterados, o que implica em uma alta manutenção no código de interpretação da planilha no bot.

Com uma DSL, conseguimos padronizar esses efeitos de forma que com apenas uma coluna, conseguimos dizer que um item adiciona 5 HP ao valor atual do player e 2 FP com apenas uma coluna e de uma forma estruturada que o bot consegue entender. Com essa mesma linguagem, conseguimos dizer que uma armadura aumenta a CA do player em 2 pontos, e que um acessório dá um bônus de 1 ponto de Força ao ser equipada. Isso tudo com apenas uma única coluna, sem precisar ficar alterando a planilha a toda vez q precisarmos de um novo efeito.

Sintaxe

A sintaxe da **Elysium DSL** será bem simples, ainda mais se você tiver alguma familiaridade com alguma linguagem de programação:

```
Alvo.Propriedade.Operação(Expresão)[.time.ContadoresDeTempo(tempo)];
```

Entraremos no detalhe de cada parte nos tópicos a seguir

Alvo

O **Alvo** irá indicar quem sofrerá o efeito descrito:

- `self` - o próprio player que está usando/equipando o item sofre o efeito descrito
- `target` - o player consegue escolher o(s) alvo(s) que irá sofrer o feito descrito

Propriedade

A **Propriedade** se refere a algum atributo ou status associado ao **Alvo**, podendo ser:

- **Atributos (vide *Cheatsheet*)**
- **Status (vide *Cheatsheet*)**
- **Perícias (vide *Cheatsheet*)**
    - Para perícias, o único operador válido é o `set`, uma vez que não faz sentido aplicar operações matemáticas nas perícias; Os únicos valores possíveis para o `set` de perícias são: `PROF` (proficiente), `NO_PROF` (sem proficiência) e `SPEC` (especialista)

Operação

A **Operação** indica o tipo de alteração que deve ser realizada na **Propriedade**.

Tendo em mente que as propriedades são essencialmente numéricas, as operações possíveis são, em sua maioria, operações matemáticas:

- `add` - soma
- `sub` - subtração
- `mult` - multiplicação
- `div` - divisão
- `set` - sobrescrita (marreta um valor específico)

Vale ter em mente que, em caso de operações que resultem em "valor quebrado", será considerada apenas a parte inteira do resultado, como no exemplo:

```
1. O hp do player está em 10
2. É aplicado um efeito de item com a seguinte expressão:
    self.hp.add(3.5);
3. O resultado final da expressão é 13.5
4. Como o resultado não é um número inteiro, é considerada apenas a parte inteira
do resultado, que no caso é 13
5. O hp final do player é 13
```

As operações podem ser *concatenadas* em sequência, como no exemplo:

```
self.hp.add(10).add(5);

Obs.: é apenas um exemplo, nesse caso, muito provavelmente, faz mais sentido a
expressão ser self.hp.add(15);
mas veremos casos em que essa concatenação de operações pode ser útil
```

Expressão

A **Expressão** é o quantificador da **Operação** desejada.

Ela pode ter as seguintes naturezas:

- Expressão numérica - quando a expressão é literalmente apenas um número ou uma expressão matemática
    - Exemplos: `10`, `5 + 5`, `10 - 2`

- Expressão de dados - quando a expressão é uma ou mais rolagem de dados
  - Exemplos: `1d4`, `2d20`, `1d6+1d8`
- Expressão mista - quando a expressão envolve tanto uma parte numérica como uma parte de rolagem de dados
  - Exemplos: `1d6+2`, `20-1d12`, `2d10+2d12-10`

No entanto, é recomendável que uma expressão complexa seja quebrada em várias expressões mais simples, para serem operadas de forma independente (e ter menos risco de ter resultados inesperados).

Por exemplo, imagine que temos a seguinte linha de DSL:

```
self.hp.add(1d6+1d4-2);
```

Podemos quebrar essa operação em múltiplas operações mais simples, atingindo o mesmo resultado:

```
self.hp.add(1d6).add(1d4).sub(-2);
```

Obs.: Não há suporte para uso de parênteses para definir prioridade de cálculo dentro dos argumentos das operações

Caracteres delimitadores

Na **Elysium DSL** teremos apenas dois caracteres delimitadores:

- `.` - indica delimitação entre alvos, propriedades e operações; com o `.`, acessamos coisas internas de um objeto
  - Exemplo: usamos o `.` para acessar o HP máximo de um alvo: `target.maxhp`
- `;` - indica o final de uma linha de Elysium DSL; é utilizado para fazer um mesmo item aplicar diversos efeitos diferentes
  - Exemplo: `self.hp.add(5); self.fp.add(2);`

Espaços em branco

Espaços em branco podem ser usado livremente. Segue abaixo alguns exemplos de uso:

- Separar linhas de **Elysium DSL**
  - ```
    self.for.add(1);
    self.des.add(1);
    self.con.sub(1);
    ```
- Separar partes de uma expressão mista
  - ```
    self.hp.add(1 + 2 - 1d4);
    self.hp.add(
      20 + 1d10 + 2d20 -
      5d6 - 3d4 + 10
    );
    ```

Por ora, todos os espaços entre tokens são ignorados, de forma que a expressão

```
se lf.      h       p .
a
d
d(2  d6);
```

é válida, no entanto, não é recomendado fazer uso dessa forma, uma vez que torna a leitura muito mais difícil e possivelmente no futuro pode vir a ter uma validação sobre os espaços em expressões não numéricas ou de dados.

Contadores de tempo

Os contadores de tempo são opcionais, constituem uma família de operações especiais para definir um tempo de vida para um determinado efeito.

Consideremos uma linha de efeito permanente como a seguinte:

```
self.maxhp.add(10);
```

Nesse caso, o hp máximo do player é aumentado em 10 permanentemente. Em alguns casos, não queremos que o efeito seja permanente, mas sim temporário (ainda mais para efeitos que envolvem propriedades máximas, como o max hp); para esses casos, podemos adicionar contadores de tempo para definir por quanto tempo esse determinado efeito irá durar.

Para separar o contador de tempo do efeito em si, é utilizada a partícula `time`, entre delimitadores:

```
self.maxhp.add(10).time;
```

Mas adicionar a partícula `time` sozinha não traz significado algum, ela só faz sentido quando acompanhada de algum contador, sendo eles:

- `days` - contador de dias
- `hours` - contador de horas
- `minutes` - contador de minutos

Adicionando um contador de horas ao efeito acima, para deixarmos o efeito atuando por **12 horas**, temos:

```
self.maxhp.add(10).time.hours(12);
```

Note que podemos adicionar múltiplos contadores, embora adicionar o mesmo contador várias vezes talvez não faça tanto sentido:

```
i.  self.maxhp.add(10).time.hours(4).hours(4).hours(4);
        é a mesma coisa que
ii. self.maxhp.add(10).time.hours(12);
```

No entando, o uso de múltiplos contadores faz sentido quando lidamos com contadores diferentes:

```
i.  self.maxhp.add(10).time.days(1).hours(12).minutes(30); -> 1 dia, 12 horas e
30 minutos
        é a mesma coisa que
ii. self.maxhp.add(10).time.minutes(2190); -> 2190 minutos = 1 dia, 12 horas e 30
minutos
```

No exemplo acima, ambos os exemplos geram o mesmo resultado, mas o primeiro é muito mais fácil de ler do que o segundo.

Cheatsheet

Alvo	Partícula	Exemplo
Jogador que usou o item	`self`	`self.hp.add(10);`
Jogador(es) selecionado(s)	`target`	`target.hp.add(10);`

Atributo	Partícula	Exemplo
Força	`for`	`self.for.add(1);`
Destreza	`des`	`self.des.add(1);`
Constituição	`const`	`self.const.add(1);`
Conhecimento	`con`	`self.con.add(1);`
Carisma	`car`	`self.car.add(1);`

Status	Partícula	Exemplo
HP	`hp`	`self.hp.add(10);`
FP	`fp`	`self.fp.add(10);`
SP	`sp`	`self.sp.add(10);`
Max HP	`maxhp`	`self.maxhp.add(10).time.hours(12);`
Max FP	`maxfp`	`self.maxfp.add(10).time.hours(12);`
Max SP	`maxsp`	`self.maxsp.add(10).time.hours(12);`
Temp HP	`temphp`	`self.temphp.add(10);`
Temp FP	`tempfp`	`self.tempfp.add(10);`
Temp SP	`tempsp`	`self.tempsp.add(10);`
EXP	`exp`	`self.exp.add(100);`
GOLD	`gold`	`self.gold.add(100);`

Status	Partícula	Exemplo
Slots	`slots`	`self.slots.add(2);`
Ganho de EXP*	`expgain`	`self.expgain.mult(1.2).time.days(1);`
Ganho de GOLD*	`goldgain`	`self.goldgain.mult(2).time.days(1);`

Perícia	Partícula	Exemplo
Acrobacia	`acro`	`self.acro.set(NO_PROF);`
Adestrar animais	`ades`	`self.ades.set(PROF);`
Atletismo	`atle`	`self.atle.set(ESPEC);`
Enganação	`enga`	`self.enga.set(NO_PROF);`
Furtividade	`furt`	`self.furt.set(PROF);`
Intimidação	`inti`	`self.inti.set(ESPEC);`
Intuição	`intu`	`self.intu.set(NO_PROF);`
Investigação	`inve`	`self.inve.set(PROF);`
Natureza	`natu`	`self.natu.set(ESPEC);`
Percepção	`perc`	`self.perc.set(NO_PROF);`
Performance	`perf`	`self.perf.set(PROF);`
Persuasão	`pers`	`self.pers.set(ESPEC);`
Prestidigitação	`pres`	`self.pres.set(NO_PROF);`
Sobrevivência	`sobr`	`self.sobr.set(PROF);`
Força	`pfor`	`self.pfor.set(SPEC);`
Destreza	`pdes`	`self.pdes.set(NO_PROF);`
Constituição	`pconst`	`self.pconst.set(PROF);`
Conhecimento	`pcon`	`self.pcon.set(SPEC);`
Carisma	`pcar`	`self.pcar.set(NO_PROF);`

Contador de tempo	Partícula	Exemplo
Dias	`days`	`self.expgain.mult(1.2).time.days(1);`
Horas	`hours`	`self.expgain.mult(1.2).time.hours(5);`
Minutos	`minutes`	`self.expgain.mult(1.2).time.minutes(30);`

Operador de propriedade	Partícula	Exemplo
Adição	`add`	`self.hp.add(10);`
Subtração	`sub`	`self.hp.sub(10);`
Multiplicação	`mult`	`self.expgain.mult(1.2).time.minutes(30);`
Divisão	`div`	`self.expgain.div(2).time.minutes(30);`
Atribuição	`set`	`self.sobr.set(PROF);`

Partícula reservada	Uso
`time`	Indicador de início dos contadores de tempo
`.`	Separador de partículas verbais (*tokens*), quando usado fora de expressões matemáticas
`;`	Indicador de final de linha de código
`(`	Indicador de início de argumentos de operador/contador
`)`	Indicador de final de argumentos de operador/contador
`NO_PROF`	Indicador de "**Sem Proficiência**", usado ao atribuir valor a uma proficiência
`PROF`	Indicador de "**Proficiente**", usado ao atribuir valor a uma proficiência
`SPEC`	Indicador de "**Especialista**", usado ao atribuir valor a uma proficiência

Definições técnicas

Estrutura

O coração da DSL será composta basicamente por 3 peças:

- **Lexer**: transforma linhas de código em tokens
- **Parser**: valida a sintaxe a partir dos tokens e extrai os comandos
- **Interpreter**: interpreta os comandos e os executa sobre uma entidade

Lexer

Irá receber um código como uma `string` e irá:

1. Remover todos os white spaces (, `\t`, `\n`, `\r`, etc.) de acordo com o regex `/\s+/g`
2. Extrair os tokens de acordo com a tabela (seguindo a ordem de prioridade):

Nome	Regex
TARGET	`/self|target/`
DICE_EXPRESSION	`/(\+|-)?(\d+)d(\d+)/`

Nome	Regex
NUMBER	`/(\+|-)?\d+(\.\d+)?/`
DELIMITER	`/\./`
OPEN_PARENTHESIS	`/\(/`
CLOSED_PARENTHESIS	`/\)/`
LINE_END	`/;/`
PROPERTY	`/for|des|const|con|car|hp|fp|sp|maxhp|maxfp|maxsp|temphp|tempfp|tempsp|exp|gold|slots|expgain|goldgain|acro|adex|atle|enga|furt|inti|intu|inve|natu|perc|perf|pers|pres|sobr|pfor|pdes|pconst|pcon|pcar/`
OPERATION	`/add|sub|mult|div|set/`
TIME_DELIMITER	`/time/`
TIME_COUNTER	`/days|hours|minutes/`
PROFICIENCY_VALUE	`/NO_PROF|PROF|SPEC/`
UNKNOWN (caso não se encaixe em nenhum outro token)	`/[^\.\(\);]+/`

Parser

Irá receber uma lista de tokens (em ordem) e:

1. Retornar erro em caso de token do tipo `UNKNOWN`
2. Retornar erro em caso de quebra da sintaxe
3. Retornar lista de comandos para serem executados na entidade

Interpreter

Recebe uma lista de comandos e uma entidade (por ora apenas do tipo **User**) e:

1. Traduz o comando para uma operação na entidade
    - caso não seja uma operação válida para aquela entidade, deve retornar um erro
2. Aplica a operação na entidade
3. Agendar rotina de remoção do comando em caso de haver contadores de tempo
    - Estudar possibilidade de um sistema de buffs temporários, nesse caso precisaria apenas fazer uma adição de buff no player, e o sistema de buff lida com o resto

Apêndice

A. Estrutura de uma expressão de dados

```
r#ndl
|||||
||||+-> número de lados
|||+--> caractere constante indicativo de dado (d)
||+---> número de dados
|+----> caractere separador de numero de rolagens (#)
+-----> número de rolagens
Sendo: n, l >= 0
```

- Propriedades:
    - `nd0` sempre resultará em `0`
    - `nd1` sempre resultará em `n`
    - `0dn` sempre resultará em `0`
    - `ndm` sempre resultará em um número entre `n` e `m` (`n <= resultado <= m`)
    - Caso seja feita uma única rolagem, o `r` pode ser omitido: `1#2d12` = `2d12`