Interpreter języka RPG

Piotr Lewandowski

24 stycznia 2022

Spis treści

1 Opis języka

1.1 Typy proste

Język wspiera 4 podstawowe typy danych:

1. Unit
2. Int
3. Bool
4. String
5. List, listy, w których wszystkie elementy są instancjami tego samego typu T.

1.2 Dice

Dice to typ danych umożliwiający reprezentowanie kości do gry o dowolnej dodatniej liczbie ścian. Wartości typu Dice reprezentują zdarzenie losowe polegające na rzucie pewną liczbą kości i zsumowanie wyników. Dla przykładu wartość „2d6" reprezentuje rzut dwiema sześciennymi kośćmi i zsumowanie liczby oczek na obu. Stałe typu Dice reprezentowane są przez string jak wyżej: najpierw podana jest liczba kości, potem separator „d", a potem liczba ścian każdej z kości. Obie liczby muszą być dodatnie. Przy użyciu jednej wartości typu Dice jesteśmy w stanie wygenerować losową wartość typu Int, przekazując kostkę to funkcji wbudowanej „roll". Przykładowe wartości typu Dice:

```
4d4             // Dice dla przedziału [4, 16]
2d6             // [2, 12]
1d4 + 7         // [8, 11]
6 * (2d3 + 2)   // [24,48]
```

1.3 Operatory

Język *RPG* wspiera następujące operatory:

1. dodawanie (+), odejmowanie (-), dzielenie całkowite (/), mnożenie (*), minus unarny(-), dla Int i Dice
2. operatory równości (==, !=), dla Int i String
3. operatory porównania <, >, <=, >=), dla Int
4. operator konkatenacji (++), dla String i List[T]
5. operatory logiczne koniunkcji (&&) i alternatywy (||) dla Bool

1.4 Obiekty

Możliwe jest zdefiniowanie obiektu, składającego się z pewnej liczby nazwanych pól. Każde pole ma określony typ, którym może być Dice lub jeden z typów prostych. Pola mogą mieć stałą wartość, lub odnosić się do innych pól tego samego obiektu. Nazwa obiektu i wszystkich pól musi zaczynać się od wielkiej litery i może zawierać litery, cyfry i znak podkreślenia „_".

```
Entity {
    Strength: 30,
    Dexterity: 30,
    Willpower: 30,
    Health: (Strength + 2 * Dexterity) / 10,
    Equipment: ["Commoner clothes"]
};
```

Obiekt może rozszerzać istniejący obiekt, poprzez nadpisanie definicji istniejących pól i/lub dodanie nowych pól. W takim przypadku możliwe jest rozróżnienie pól obecnie definiowanego obiektu od pól obiektu rozszerzanego przez użycie prefixów „base." i „this.".

```
Mage extends Entity {
    Willpower: base.Willpower + 10,
    Mana: 10 * this.Willpower
};
```

1.5 Cechy

Możliwe jest też zdefiniowanie cechy dla danego obiektu, która podobnie jak rozszerzający obiekt może modyfikować i dodawać pola. Definicja cechy poprzedzona jest słowem kluczowym "trait".

```
trait Strong for Entity {
    Strength: base.Strength + 10
};
trait BanditEquipment for Entity {
    Equipment: base.Equipment ++ ["SilverCoin", "Dagger"]
};
```

Cech można użyć przy definicji obiektów w nastepujący sposób:

```
BanditMage extends Mage with BanditEquipment and Strong { };
```

1.6 Przypisanie

Przypisania są postaci:

```
set identyfikator = wartość;
```

Jeśli dany identyfikator nie wystąpił wcześniej po lewej stronie przypisania to tworzona jest nowa zmienna z daną wartością, w przeciwnym przypadku nadpisywane są istniejące wartości. Wartością po prawej stronie przypisania może być dowolna wartość kości, typu prostego lub identyfikator obiektu (z opcjonalną listą dodatkowych cech). Użycie identyfikatora obiektu powoduje stworzenie nowej instancji danego typu.

```
set answer = 42;
set fighter = Entity with Strong;
set something = Entity;
```

1.7 Funkcje

Możliwe jest definiowanie własnych funkcji jak na poniższych przykładach. Nazwy funkcji muszą zaczynać się małą literą i mogą zawierać małe i wielkie litery oraz znak podkreślenia „_”. Po nazwie funkcji znajduje się lista parametrów, wraz z ich typami, a następnie typ zwracany i ciało funkcji. Parametry typów prostych i Dice przekazywane są przez wartość. Obiekty przekazywane są przez referencję. Typ zwracany może zostać pominięty dla funkcji, które nie zwracają wartości. Wartością zwracaną jest ostatnie wyrażenie w ciele funkcji.

```
fun flipCoin(): Bool {
    roll(1d2) == 1;
};
fun hello(name: String) {
    print(name);
};
fun add3(x: Int): Int {
    x+3;
};
```

1.8 Funkcje wbudowane

Dostępne są dwie funkcje wbudowane:

1. getString - Zwraca tekstową reprezentację dowolnej wartości
2. print - Wypisuje dany ciąg znaków
3. roll - „Rzuca” podaną kością, generując liczbę losową w odpowiadający jej sposób

```
print("Hello world!");
print(getString(fighter));
print(getString(roll(1d10)));
```

1.9 If-then-else expression

If jest wyrażeniem, jeśli wartość sterująca w nawiasie po słowie „if" ewaluuje się do True, to wartością całego wyrażenia if jest wartość znajdująca się po słowie „then", w przeciwnym przypadku wartość po słowie „else". Zamiast pojedynczej wartości gałęzie ifa mogą zawierać też blok ograniczony klamerkami „{}". W takim przypadku wartość danej gałęzi jest równa wartości ostatniego wyrażenia z bloku.

```
if ((1 * roll(1d10)) - 1 == 7)
    then 3
    else {
        roll(1d40) + 7;
    };
```

2 Wymagania funkcjonalne

1. Interpretacja skryptów zapisanych w plikach tekstowych
2. Możliwość definiowania funkcji i obiektów zgodnie z powyższą specyfikacją języka
3. Przyjazna dla użytkownika obsługa błędów składniowych i semantycznych

3 Wymagania niefunkcjonalne

1. Wraz z interpreterem powinna zostać dostarczona instrukcja obsługi
2. Moduły interpretera powinny zostać przetestowane

4 Plik wykonywalny

Interpreter jest aplikacją konsolową napisaną w języku C#, korzystającą z ASP.NET Core w wersji 6. Implementacja korzysta głównie ze standardowej biblioteki języka C#. Testy wykorzystują także bibliotekę NUnit.

4.1 Obsługa programu

Interpretera będzie można używać albo do wykonania skryptu zapisanego w pliku tekstowym, albo do uruchomienia REPLa, z opcją wcześniejszego uruchomienia skryptu z pliku, co pozwala na korzystanie z poziomu REPLa ze zdefiniowanych w tym pliku typów, funkcji i zmiennych. Pliki źródłowe języka

RPG korzystają z rozszerzenia „.RPG” lub „.rpg”. Wyjście zdefiniowane w ramach skryptów oraz wszelakie błędy związane z interpretacją będą wypisywane na standardowe wyjście. Opcje uruchomienia obrazuje poniższy przykład:

```
./interpreter -h
./interpreter script.rpg
```

5 Testy

Kod źródłowy interpretera korzysta z dwóch sposobów testowania. Poszczególne moduły zostały przetestowane jednostkowo, a całość została także przeszła testy integracyjne na specjalnie przygotowanych plikach źródłowych w języku RPG.

6 Interpretacja

Każdy z poniższych etapów ma odpowiadający mu namespace w kodzie źródłowym interpretera. Oraz osobny typ bazowy wyjątków rzucanych w razie błędu. Główny program łapie te wyjątki i wypisuje użytkownikowi wiadomość mówiącą o rodzaju błędu oraz wskazującą odpowiednie miejsce w pliku źródłowym.

6.1 Tokenizacja

Implementacja tokenizacji wczytuje ze źródła pojedynczy znak i stwierdza, które tokeny mogą rozpoczynać się od tego znaku.

6.2 Parsowanie

Zaimplementowany parser jest parserem typu RD (recursive descent), w wersji bez powrotów, z lookaheadem równym 1. Tworzy on drzewo, które następnie jest wykorzystywane przy analizie typów oraz wykonaniu programu.

6.3 Analiza typów

Na tym etapie sprawdzana jest poprawność semantyczna interpretowanego programu. Wyniki tego etapu są zapisywane jako mapa pomiędzy wierzchołkami wygenerowanego przez parser drzewa a wyliczonymi dla nich typami.

6.4 Wykonanie

Na etapie wykonania to samo drzewo parsowania co w poprzednim etapie zostaje przetworzone po raz kolejny, tym razem korzystając z założenia, że większość błędów semantycznych została wyeliminowana na etapie analizy typów. Kolejne instrukcje z pliku źródłowego są po kolei wykonywane.

Wygenerowane przez program wyjście jest przekierowywane na standardowe wyjście lub zapisywane w pamięci, zależnie od konfiguracji interpretera. Na potrzeby pliku wykonywalnego wykorzystywana jest pierwsza opcja, a z drugiej korzystają testy.

7 Przykładowy skrypt

```
Entity {
    Name: "Unknown",
    Strength: 30,
    Dexterity: 30,
    Willpower: 30,
    Health: (this.Strength + 2 * this.Dexterity) / 10,
    Equipment: []
};

trait Strong for Entity {
    Strength: base.Strength + 10
};

fun fight(e1: Entity, e2: Entity) Entity {
    print(e1.Name ++ " has " ++ getString(e1.Health) ++ " health\n");
    print(e2.Name ++ " has " ++ getString(e2.Health) ++ " health\n");
    set r1 = roll(1d100) - e1.Strength / 10;
    set r2 = roll(1d100) - e2.Strength / 10;
    set result = r1 - r2;
    set loser = if (result > 0)
        then e2
        else e1;
    set winner = if (result > 0)
        then e1
        else e2;
    set damage = winner.Strength / 10;
    set loser.Health = loser.Health - damage;
    print(loser.Name ++ " lost " ++ getString(damage) ++ " health.\n");
    if (loser.Health <= 0)
        then {
            print(loser.Name ++ " has died!\n");
            print(winner.Name ++ " has won!\n");
            winner;
        }
        else {
            print("Fight continues!\n");
            fight(winner, loser);
```

```
        };
};

set fighter = Entity with Strong;
set fighter.Name = "Marcus";
set bandit = Entity;
set bandit.Name = "Otto";

fight(fighter, bandit);
```