LINKÖPINGS UNIVERSITET

TDP019

Ruler

Författare

Max Byrde, maxby769@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2018
Version 1.0

14 maj 2018

Innehåll

1 Inledning

Det här projektet gjordes som ett moment i kursen TDP019 på Linköpings Universitet. Uppgiften gick ut på att designa och implementera ett valfritt programmeringsspråk med hjälp utav språket Ruby, mitt språk är ett regelbaserat system för att skapa dynamiska interaktioner mellan karaktärer i en dataspelsvärld. Idén är att bryta ner spelvärlden till en samling lätthanterlig fakta som assoscieras med en nyckel, och sedan använda denna fakta för att skapa regler som i sin tur styr vilken interaktion som ska ske när och var. Till exempel kanske man vill att en vakt i spelvärlden ska reagera annorlunda mot spelaren beroende på om spelaren är stark eller svag.

2 Användarhandledning

I grund och botten är systemet bara en samling regler med villkor som säger när den en viss regel ska triggas och vad den ska göra när den triggas (s.k. handlingar). Villkoren beror på fakta i en tabell som matas in av en programmerare, och det är meningen att tabellen ska uppdateras med ny fakta med vissa intervall för att på så sätt trigga nya regler som inte varit aktuella innan. Jag kommer gå igenom varje konstruktion i systemet för att förklara hur allt hänger ihop.

2.1 Fakta

Först ska vi prata om hur fakta om spelvärlden representeras i Ruler. Det finns tre olika datatyper i Ruler, dessa är integer (en siffra), string (sträng, en bit text t.ex: ”Hej!”) eller boolean (ett s.k. boolsk värde, kan antingen vara sant eller falskt). All fakta sparas som en av dessa datatyper, och all fakta har dessutom en unik nyckel. Exempelvis kan antalet guldtackor som spelaren äger representeras som en siffra, medans spelarens namn passar bättre som en sträng. Huruvida spelaren har besökt en viss stad passar bättre som boolskt värde (ja spelaren har besökt Linköping, nej spelaren har inte besökt Linköping). En sådan tabell skulle kunna se ut på följande sätt.

Nyckel	Värde
“player_gold”	500
“player_name”	“Max”
“player_visited_linkoping”	**false**

På vänster sida ser vi att vår fakta fått en nyckel, i systemet används denna nyckel för att ta reda på vilket värde ett viss fakta har just nu. För att göra det används s.k. queries. Ett query ser ut på följande sätt, där strängen mellan hakparanteserna är en nyckel.

```
query["player_gold"]
```

Ett query tittar i faktatabellen och ger tillbaka det värde som faktat har, i vårt fall siffran 500.

2.2 Koncept

För att hjälpa oss bygga regler har vi också någonting som kallas för concept (koncept). Ett koncept är ett påstående om någonting i spelvärlden som endast kan vara antingen sant eller falskt, och byggs med hjälp av queries och villkor. Ett exempel på koncept är påståendet “spelaren är vid liv”. Ett sådant koncept kan skrivas på följande sätt i systemet

```
concept PlayerIsAlive
    query["player_health"] > 0 && query["player_alive"] == true
end
```

Vi ser att varje koncept i systemet måste ha ett namn, i vårt fall "PlayerIsAlive". Vi ser också att två queries används för att titta upp fakta i tabellen, "player_health" som antas representera spelarens hälsa som en siffra, och "player_alive" som antas representera huruvida spelaren är vid liv som ett boolsk värde. Men vi ser också konstiga tecken mellan dem, t.ex "==" och "&&". Dessa är logiska operationer och kan föreställas som ett sorts minipåstående mellan två objekt. Till exempel betyder "==" lika med, ">" större än och "&&" och. Konceptet kan alltså läsas som "spelarens hälsa större än 0 och spelaren vid liv är sant". i systemet kan ett påstående närsomhelst anropas för att kontrollera huruvida det är sant eller falskt just nu beroende på den fakta vi har, och används för att beskriva villkoren för att en regel ska triggas.

2.3 Regler

Nu är vi redo att skapa regler. En regel har ett villkor och en samling handlingar som ska ske när regeln triggas, d.v.s när regelns villkor är sant. En regels villkor fungerar precis som villkoret i ett koncept, med skillnaden att även koncept kan användas genom att anropa dem via sitt namn. Som exempel vill jag skapa en regel som får spelaren att klaga på värmen när temperaturen är över en viss gräns och spelaren är vid liv. En sån regel ser ut så här:

```
concept PlayerIsAlive
    query["player_health"] > 0 && query["player_alive"] == true
end

concept IsHot
    query["temperature"] > 30
end

rule PlayerComplainsAboutHeat
    PlayerIsAlive && IsHot
    say "Fasen vad varmt det är"
end
```

Förutom regeln själv använder vi två koncept i regelns villkor. Vi ser också att vi anropar en handling med namnet "say", och som argument till anropet har vi vad spelaren ska säga. En regel kan ha obegränsat med handlingar, och handlingarna i säg är inte definerade av systemet utan av användaren. Detta betyder att man kan skapa nya handlingar med vilket namn som helst som gör vad som helst, och gör systemet mycket flexibelt.

3 Systemdokumentation

Systemet består av en tokenizer, en parser och en runtime. Alla tre är skrivna för hand utan hjälp av frameworks.

3.1 Tokenizer

Tokenizern översätter en sträng av text till en lista helt bestående av tokens som skickas vidare till parsern. Grammatiken för tokenizern ser ut så här:

```
CHAR ::= Any unicode character encoded as UTF-8

BOOL_LITERAL ::= 'true' | 'false'

INT_LITERAL ::= [0-9]+

STRING_LITERAL ::= '"' + CHAR* + '"'

END = ::= 'end'

CONCEPT_T ::= 'concept'

RULE_T ::= 'rule'

QUERY ::= 'query' + '[' + STRING_LITERAL + ']'

COMPARISON_OP ::= '<' | '>'

EQUALITY_OP ::= '==' | '!='

LOGIC_OP ::= '&&' | '||'

IDENTIFIER ::= [A-Za-z][A-Za-z0-9_]+
```

3.2 Parser

Parsern parsar alla tokens till antingen rules eller concepts, som i sin tur består av conditions, anrop till actions och literals. Grammatiken för parsern ser ut så här:

```
PRIMARY ::= BOOLEAN_LITERAL | STRING_LITERAL | INT_LITERAL | QUERY
    | IDENTIFIER

COMPARISON_COND ::= PRIMARY + COMPARISON_OP + PRIMARY

EQUALITY_COND ::= PRIMARY + EQUALITY_OP + PRIMARY

LOGIC_COND ::= (COMPARISON_COND | EQUALITY_COND | PRIMARY) + (LOGIC_OP
    + (COMPARISON_COND | EQUALITY_COND | PRIMARY))+

CONDITIONAL ::= COMPARISON_COND | EQUALITY_COND | LOGIC_COND | PRIMARY

CONCEPT ::= CONCEPT_T + IDENTIFIER + CONDITIONAL + END

ARGUMENT ::= BOOLEAN_LITERAL | STRING_LITERAL | INT_LITERAL

ACTION ::= IDENTIFIER (+ ARGUMENT)*

RULE ::= RULE_T + IDENTIFIER + CONDITIONAL (+ ACTION)* + END
```

3.3 Runtime

Ett kontext är grunden i runtimen, och är vad som returneras av parsern. I parsern fylls kontextet med alla koncept och regler som finns i koden som matas in, men kontextet måste även fyllas i med s.k. actions. Actions är ett dictionary med en sträng som nyckel och en lambda funktion som värde, funktionen bestämmer vad som ska göras när en action med ett visst namn anropas.

Conditions, action anrop och literals valde jag att representera i en nodstruktur där varje konstruktion har en egen nod som kan kombineras med andra noder för att skapa mer komplicerat beteende (ex: LiteralNode, ActionNode, EqualityNode o.s.v).

Contextet fylls seda med en faktatabell och kan sedan exekveras. En exekvering kör alla regler och kollar ifall deras villkor är uppfyllda. Om de är uppfyllda så körs deras actions.

4 Reflektion

4.1 Gick bra

Att designa själva runtimen var förvånansvärt lätt. Även att skriva parsern efter att tokenizern var färdig.

4.2 Gick dåligt

Svårt att komma på en bra språk idé. Mycket pill när tokenizern skulle skrivas, bland annat hur man hanterar whitespace och tokens där inget whitespace fanns emellan dem, t.ex hakparanteser och liknande. Jag hade även problem med att få de logiska operatorerna att köras i rätt ordning, men när jag valde att bygga in ordningen direkt i grammatiken gick det bättre.

4.3 Ruby

Ruby är ett intressant språk, det är väldigt flexibelt och låter programmeraren göra det han vill göra. Finns även massor med olika sätt att lösa samma problem på. Personligen gillar jag dock inte det, utan vill hellre ha ett språk som python där det bara finns ett sätt, och ifall det sättet är dåligt på byts det ut mot ett annat. Experimenterade även lite med metaprogrammering men tyckte det blev för hackigt för att underhålla till slut. Blocks är mycet bra och något jag hoppas andra språk plockar upp i framtiden, men tror inte jag kommer använda Ruby i framtiden igen.