UniCODE

Functioneel Programmeren project 3 - René Van Der Schueren

Architectuur van de interpreter

De UniCODE interpreter bouwt een boom van elementen op. Deze worden dan van de top naar beneden geëvalueerd. Doorheen deze evaluatie wordt de ‘Store’ doorgegeven. Dit is een map die de variabele namen (strings) afbeeldt op waarden. Deze waarden kunnen integers, functies of de staat van het spel zijn. Er zijn verscheidene booleaanse operaties ondersteund, deze werken ook met integer waar 0 vals is en alle andere waarden true zijn.

We onderscheiden twee types element in de boom.
‘Commands’ zijn gewone operaties zoals loops of declaraties. Ze nemen als argument de Store en geven als resultaat een al dan niet aangepaste Store terug.
‘IntExpressions’, zijn expressies, deze nemen als argument de store en geven telkens een integer terug deze kunnen de store niet aanpassen.

```
{
    ○ test(n){
        ↪(1);
    };
    🖥(test());
}
```

Zo zal bovenstaande code omgezet worden in onderstaande boom.

```
Scope
    | Seq
        | Function "test" ["n"]
            | Scope
                | Return
                    | :+:
                        | ReadVar n
                        | Lit 1
        | Print
            | Call "test" [Lit 1]
```

Veranderingen aan variabelen in een Scope zullen alleen maar gereflecteerd worden buiten de scope als deze variabelen ook gedefinieerd zijn buiten de scope.

Evaluatie gebeurt niet van de top naar beneden. Functies worden opgeslagen als een variabele met als waarde de gepaste functie. Bij een oproep wordt deze dan telkens geëvalueerd.

Voorbeeldprogramma

UniCODE werkt met een klassieke javascript geïnspireerde syntax waarbij verscheidene operators en built-in functies vervangen zijn door unicode tekens.

```
{
    ○ factorial(n){
        ?((n ⟺ 1)){
            ↳(n);
        }¿{
            a = factorial((n - 1));
            ↳((n * a));
        };
    };

    🖥(factorial(3));
}
```

Bovenstaand voorbeeld beschrijft een functie voor het berekenen van de faculteit. Alle code bevindt zich in de global scope gemarkeerd door de buitenste { en }, hierbinnen mogen geen tekens staan. Op de eerste lijn wordt een functie gedeclareerd. Deze roept hier zichzelf recursief op, mogelijk gemaakt door de implementatie van scopes. Op de onderste lijn wordt deze dan opgeroepen met argument 3 en uitgeprint. Het programma output uiteraard 6.

Verdere ondersteunt UniCODE volgende syntax:

```
integer operators: + - * / %
boolean operators: ¬ ∧ ∨ ⟺ > < ≥ ≤ ≠
if: ?
else: ¿
while: 🔁
assign: =
function: ○
increment: ++
print: 🖥
return: ↳
engine prefix: 🎮
```

(Voor gemakkelijk gebruik kan u deze allemaal vinden in code_examples/syntax_list.xyz.)

Verder ondersteunt UniCODE een set aan game engine functies. Deze bevinden zich allemaal onder de engine prefix die hierboven beschreven staat. Deze simuleert een spel

door middel van de coördinaten voor de speler (PLAYER) en twee lijsten van coördinaten voor de ander actoren (BAD/GOOD).

We bespreken hieronder kort de implementatie van snake. We beginnen met het definiëren van een step functie die elke simulatie stap zal uitgevoerd worden.

```
○ snakeStep(){
    🎮addTo(BAD, PLAYER);
    ?(🎮playerOverlapWith(GOOD)){
        🎮clear(GOOD);
        🎮placeRandom(GOOD);
    }¿{
        🎮removeLast(BAD);
    };
    🎮moveInDirection(PLAYER);
    ?(( (¬🎮playerOnMap) v 🎮playerOverlapWith(BAD) )){
        🎮showScoreScreen(🎮amountOf(BAD));
    };
};
```

Hier zullen we de speler bewegen, kijken of hij voedsel geraakt heeft en kijken of hij de rand of zichzelf niet geraakt heeft.
Verder definiëren we een functie per key die we zullen binden zoals:

```
○ handleRight(){ 🎮setDirection(E); };
```

Uiteindelijk starten we met createGame een game instantie.
We plaatsen het eerste voedsel. We kiezen de kleur van de kop, de staart en het voedsel. We binden alle controles. En we starten het spel met 2 stappen per seconde.

```
createGame(snakeStep);
placeRandom(GOOD);

setColor(PLAYER, 255, 153, 0);
setColor(BAD, 51, 204, 51);
setColor(GOOD, 255, 0, 0);

bind(RIGHT, handleRight);
bind(LEFT, handleLeft);
bind(DOWN, handleDown);
bind(UP, handleUp);
bind(SPACE, resetGame);

startGame(2);
```

De engine ondersteunt verder volgende functies:

```
createGame
startGame
removeGoodBadOverlap
resetGame
boundPlayerToMap

moveInDirection(PLAYER/GOOD/BAD)
bind(LEFT/RIGHT/UP/DOWN/SPACE, ○())
placeRandom(GOOD/BAD)
clear(GOOD/BAD)
addTo(GOOD/BAD, PLAYER)
addTo(GOOD/BAD, x, y)
setDirection(N/E/S/W)
removeLast(GOOD/BAD)
showScoreScreen(score)
setColor(PLAYER/GOOD/BAD, r, g, b)
setColor(PLAYER/GOOD/BAD, r, g, b, a)
removeDuplicates(GOOD/BAD)
parseAddRandomBadToRow(row)
removeOfMap(GOOD/BAD)
forEach(GOOD/BAD, ○(x, y))
```

```
amountOf(GOOD/BAD)
playerOverlapWith(GOOD/BAD)
playerOnMap
getPlayerX
getPlayerY
getMaxX
getMaxY
```

Gebruik monad transformer

De monad stack van de volledige interpreter ziet er uit als volgt:
`String → Parser Command → IO()`
Het gebruik van een monadische parser was uiteraard zeer voordelig, het parsen van werd zo een simpele aaneensluiting van verschillende kleinere parsers.

Het type van de Command evaluator is:
`Command → Store → IO Store`
Het zou hier mogelijks nog een verbetering geweest zijn moesten de Store in een State monad geplaatst worden. Ik vond dit hier echter overbodig aangezien de store een map is waar het opvragen van een waarde nooit zorgt voor een aanpassing. Er wordt dus altijd of een waarde of een nieuwe store teruggeven en nooit een koppel met beide.