# Summary:

Nomic is a unique game proposed by Peter Suber in 1982: «Nomic is a game in which changing the rules is a move. In that respect it differs from almost every other game. The primary activity of Nomic is proposing changes in the rules, debating the wisdom of changing them in that way, voting on the changes, deciding what can and cannot be done afterwards, and doing it. Even this core of the game, of course, can be changed.… » (see [1], [2])

The objective of this document is to describe how a Nomic game can be handled by a computer, to propose a set of requirements and finally a design and an implementation in Haskell.

The proposed game is online and can be played by several players simultaneously. The computer manages the pool of laws, checks automatically their validity, and apply them.

To win, one must achieve an objective fixed by one of the initial laws. Of course this objective can be changed during the game.

# Game description

## Description of a game turn

The game consists into proposing new rules or modifications for the existing ones and voting for them. The rules are written in a special language.

The engine will then check if the proposed rule is valid versus the already active rules. The new rule will be either accepted or rejected. If it is rejected, the player will be able to modify it a re-submit it. If it is accepted, it becomes « active » and is added to the pool of active laws.

Once active, the new rule is executed and can modify the state of the game.

## Examples of rules

1. A rule mustn’t do nothing.
2. The rule number 1 is abrogated.
3. A rule mustn’t suppress another rule.
4. A rule must be voted at the unanimity by the players.
5. A new currency is created: the ECU. Every player as a bank account in ECU.
6. The players wins 1 ECU by rule proposed and 5 ECU by rules accepted.
7. With X ECU the players can buy some powers…
8. The rules must be named. The names have the form RI1, RI2…
9. A player cannot propose more than two rules at once.
10. The first player with 100 Ecu wins the game.
11. A player receives 1 Ecu each day at midnight.
12. The democracy is abolished. God save the King, Player #2 !
13. The laws must be approved by the King.
14. The King can be overthrown if a referendum is thrown and won against him, with an absolute majority of the players.
15. This rule must be executed once and suppressed.
16. The player 1 cannot propose new rules.
17. The rules 1 to 10 cannot be suppressed.
18. The player #1 must sing a song out loud.

## Example of game:

1. Start the game / connect to the web server
2. Register as a player
3. Join an existing game / create a new one
4. Display of the state of the game, including the players and the active rules
5. The player #1 proposes a new rule
6. This rule is scanned by the existing rules. If « voting » rules are present, the players must vote for or against the rule.
7. If the new rule succeeds the test, it is accepted and becomes active. Is then executed.
8. If it fails, it is rejected.
9. Another player can propose a new rule (can be done simultaneously).
10. Etc. until the winning criteria (defined by one rule) is met.

# Analysis

## Analysis of the game

The main object handled by the game is the “rule”. It is so important to give a precise definition to this concept in Nomic. We’ll define a rule as: “A rule is a proposition that allows determining the legality of an object”. The main object manipulated by Nomic being the rule, a rule is then a proposition that allows determining the legality of another rule.

This auto-referential definition is rich and gives a lot of interest to the game. Of course, beside from this, when executed, a rule can also change the state of the game.

## Analysis of the rules

From the analisys of the example rules listed above, we can define several necessary concepts and categories:

### Time :

* Rules with a transient effect: ex. #2
* Rules with a permanent effect: ex. #4
* Rules with a regular effect: ex #11

Indeed rule #2 should be deleted as soon as it is applied: we don’t want to suppress every rule with the number #1, only the current one.

On the other hand, the rule #4 must be permanently added to the rule table.

Some rules are applied on a regular basis, like rule #2.

### Subject:

* The rules that can modify the rule set: ex. #2.
* The rules that gives conditions on the futur rules: ex. 4
* The rules that does not have the other rules as subject.

### Perimeter :

Some rules introduce new notions, like rules #5 or #7. New vocabulary is added. Others are just using the existing vocabulary.

# Requirements

## Requirements on the rule engine

1. The engine must allow the player to compose a new rule in an appropriate language.
2. The engine must allow a rule to be the composition of two other rules.
3. The engine must allow a rule to observe the state of the game to make decisions.
4. The engine must allow a rule to read another rule and make decisions on its legality.
5. The engine must allow a rule to change the state of the game.

## Requirements on the game

The game must be multiplayer.

The game must allow the players to connect and propose their rules, asynchronously.

# Design

## Rule flow

Here is the flow process followed when a new rule is submitted:

***Constitution***

* Rule 1
* Rule 2
* Rule 3

…

New Rule

Legal ?

Submission

Legal ?

Legal ?

Yes

Yes

Yes

No

No

No

Rejected

Accepted, add to the constitution

***Constitution***

* Rule 1
* Rule 2
* Rule 3
* New Rule

…

Execute the new rule (may change the constitution)

A rule is executed immediately after being accepted.

Some part of it must also be executed on a regular basis.

Based on this, rules have three use cases:

## Variables

In order to store values like the amount of Ecu for each player, the game engine shall allow the players to create and manage variables.

The variables are stored inside an array (key, value) inside the game state.

## Cron table / Event table

In order to allow the rules to have some part of them executed on a regular basis or on event, a table (Event, Rule code) is stored inside the game state.

When the rule is executed for the first time after being accepted, some part of its code may be inserted in this table for latter execution.

## Type of a rule

To be able to read and write the game state, the rules should use the Monad State:

Type Rule = MonadState GameState.

A rule must also be able to judge whereas a new rule is legal or not, based on the code of this rule, and the current game state. This leads to the following functional definition of a rule:

:

Type Rule = Rule { ruleNumber :: Int,

ruleName :: String,

ruleDesc :: String,

ruleFunc :: RuleFunc,

ruleCode :: String}

Type RuleFunc = Maybe Rule -> State GameState Bool

A rule contains a rule number, a name, a description in English, a “rule function” and its plain code in the form of a string.

The rule function (“ruleFunc”) is simply the interpretation (by an Haskell interpreter) of the plain text rule (“ruleCode”).

The “ruleFunc”, the most important part, is indeed a function with one argument: it can be a rule, or nothing. This function, when applied to a rule, returns a Boolean and is able to read and write the state of the game, due to the monad state.

After being accepted, the rule is executed with the parameter ‘Nothing’. It can then change

Alternative implementation:

In order to clearly separate the meta rules from the other rules, we can do:

Type RuleFunc = Either (Rule -> StateT GameState Bool) (State GameState)

This implementation has the advantage to get rid of the parameter if the rule is not a meta rule. Thus, it’s a little more elegant during the execution. However

# Implémentation

## Introduction

Les règles formeront une algèbre compositionnelle.

Il faut trouver un certain nombre de règles primitives, et d’opérateurs de combinaisons afin de composer des règles plus compliquées.

D’autre part, les règles devront être dotées d’éléments permettant une analyse des règles.

Une règle est vue comme un arbre. Les éléments d’analyse dont elles sont dotées seront donc les opérateurs valables pour les arbres ainsi que les opérations ensemblistes.

## Headers

> module Nomic where

> import Test.QuickCheck

## Basic data types

On définit d’abord un tour de jeu et le numéro d’un joueur:

> type Turn = Int

> type Player = Int

## Rule type

Voici un type approprié pour les règles :

> data Rule = MustBeEgalTo Rule

> | AlwaysLegal

> | AlwaysIllegal

> | And Rule Rule

> deriving (Eq, Show)

## Primitives for rules

Définissons ensuite un certain nombre de règles de base :

La règle zero ne fait rien.

zero :: Rule

zero = Zero

La régle legal est legale.

legal :: Legal

legal = True

La régle illegal est illegale.

legal :: Legal

legal = False

La règle rAnd compose deux règles pour en former une troisième :

> rAnd :: Rule -> Rule -> Rule

> rAnd = And

La règle cond choisit une règle ou une autre en fonction d’un critère :

> cond :: Obs Bool -> Rule -> Rule -> Rule

> cond = Cond

Le règle mustBeEgalTo oblige à être égale à une règle donnée :

> mustBeEgalTo :: Rule -> Rule

> mustBeEgalTo = MustBeEgalTo

La règle not est legal si une règle ne l’est pas, et inversement :

not :: Rule -> Rule

not = Not

## Observables

Un observable est l’input qu’une règle peut avoir sur le monde extérieur.

> newtype Obs a = IO a

La function vote permet aux joueurs de voter (version un seul joueur pour l’instant)

> vote :: Obs Bool

> vote = do

> putStrLn “Entrez True pour accepter la proposition, False sinon »

> a <- getLine

> return $ read a

## Exemple de règles :

Règle qui disparait d’elle-même si un critère devient vrai :

> autoErase :: Obs Bool -> Rule

> autoErase o == cond o (autoErase o) illegal

Une règle ne doit pas contenir la régle « illegal » :

> noZero :: Rule -> Rule

> noZero == not . included $ illegal

Implémentation de la règle #4.

> voteRule :: Rule

> voteRule r = cond vote r illegal

## Exécution des règles

Exécuter une règle consiste à fournir deux règles, afin de vérifier la légalité de la seconde en fonction de la première.

> isRuleLegal :: Rule -> Rule -> Bool

> isRuleLegal (MustBeEgalTo r) r1 = r == r1

> isRuleLegal AlwaysLegal \_ = True

> isRuleLegal AlwaysIllegal \_ = False

> isRuleLegal (a `And` b) r = (isRuleLegal a r) && (isRuleLegal b r)

Récupération de toutes les règles de base dans une liste:

> baseElements :: Rule -> [Rule]

> baseElements legal == [legal]

> baseElements (a `rAnd` b) = (baseElements a) ++ (baseElements b)

> etc.

## Système de jeu

> main = do

> putStrLn « Welcome to Haskell Nomic »

>

>

## Tests

> r1 :: Rule

> r1 = MustBeEgalTo AlwaysLegal

> r2 :: Rule

> r2 = MustBeEgalTo (MustBeEgalTo AlwaysLegal)

> r3 :: Rule

> r3 = AlwaysLegal

> test1 = isRuleLegal r1 r1 --False: a program cannot contain a whole representation of itself.

> test2 = isRuleLegal r2 r1

> test3 = isRuleLegal (r1 `And` r1) r1

> test4 = isRuleLegal (r2 `And` r2) r1

> test5 = isRuleLegal (r1 `And` r2) r1

> test6 = isRuleLegal (r2 `And` r1) r1

> test7 = isRuleLegal r3 r1

### Instances

> instance Arbitrary Rule where

> arbitrary = sized (arbtree 0 maxkey)

> where maxkey = 1000

> arbtree :: Int -> Int -> Int -> Gen (Rule)

> arbtree lo hi n

> | n <= 0 = elements [AlwaysLegal, AlwaysIllegal]

> | lo >= hi = elements [AlwaysLegal, AlwaysIllegal]

> | otherwise = do{ i <- choose (lo,hi)

> ; m <- choose (1,30)

> ; let (ml,mr) | m==(1::Int)= (1,2)

> | m==2 = (2,1)

> | m==3 = (1,1)

> | otherwise = (2,2)

> ; l <- arbtree lo (i-1) (n `div` ml)

> ; r <- arbtree (i+1) hi (n `div` mr)

> ; return (rAnd l r)

> }

> prop\_autoContain r = not $ isRuleLegal p p

> where p = MustBeEgalTo r

# References

[1] <http://www.earlham.edu/~peters/nomic.htm>

[2] <http://en.wikipedia.org/wiki/Nomic>

* How to mock a mocking bird
* Atoms
* Financial haskell by S. P. Jones