

LogicKit contre le Minotaure



4 mars 2019

Damien Morard

Final date : 24 mars 2019 à 23h59

Files to include : Dans votre dossier privé sur gitlab (où vous aurez déjà bien **pull** le cours, cf.: [semantique/README.md](#)), vous irez dans le dossier **semantique/Exercices/TP1**. Dans ce dossier vous devrez inclure :

- Un rapport au format **PDF**, de la forme **TP1_nom_prenom.pdf**
- La partie code devra être complétée dans le fichier **TPs/TP1/Sources/TP1/main.swift**

Les TP sont des travaux *personnels*, si deux solutions ont des similarités flagrantes les deux personnes auront 0.

La date et l'heure de rendu sont *strictes*, passer le délai d'une minute utilisera un joker (pour une journée supplémentaire). Une fois les deux jokers du semestres consommés et le délai dépassé, vous recevrez une note de 0. Bien entendu la date et l'heure de rendu sont toujours considérées au fuseau horaire de Genève.

Votre rapport contiendra des exemples d'applications de vos règles et des réponses aux questions posées.

Attention à la présentation de votre travail, le rapport comptera pour une partie de la note. De même, les formats de vos fichiers (pdf pour le rapport ...) et la présentation du code (indentation, commentaire ...) sont très importants ! Ces critères rentreront en compte dans l'évaluation de vos TP et pourront vous faire perdre des points.

Au cours de ce TP, vous créerez des programmes LogicKit en Swift. Vous devrez maîtriser l'écriture de règles récursives, et la manipulation de listes. Vous utiliserez aussi des opérations de calcul sur les entiers. Si vous avez besoin de documentation supplémentaire n'hésitez pas à regarder le github de [LogicKit](#).

Dans les sous-sols labyrinthiques de Battelle, le Minotaure demande tous les neuf ans sept étudiants en informatique. Il a depuis longtemps abandonné l'espoir d'obtenir aussi les sept jeunes informaticiennes commandées.

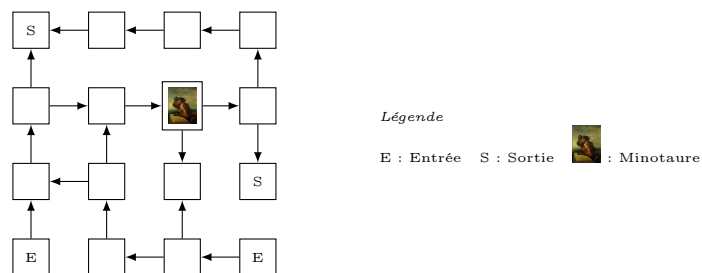


FIGURE 1 – Plan du labyrinthe

Cependant, la raréfaction d'étudiants ne permet pas de conserver cette tradition. Les différentes étapes de cette séance d'exercices nous permettront de trouver le Minotaure, si possible le vaincre, et peut-être trouver la sortie des sous-sols.

Exercice 1 : Description du labyrinthe (1 points)



Le labyrinthe du sous-sol est une succession de pièces carrées, reliées entre elles par des portes. Une pièce a au maximum une porte par mur. Les portes ne s'ouvrent que dans un seul sens ; tout retour en arrière est ainsi interdit. De plus, le labyrinthe ne contient pas de cycles ; il est impossible de revenir dans une pièce après en être sorti.

1. Représentez le labyrinthe de la figure 1 dans une base de faits Prolog. Pour cela, vous remplirez la relation `.fact("porte", de, vers)`, qui indique les portes qui peuvent être ouvertes.
2. Le labyrinthe a plusieurs entrées, plusieurs sorties et un Minotaure. Représentez-les avec les relations `.fact("entree", piece)`, `.fact("sortie", piece)` et `.fact("minotaure", piece)`.

Exercice 2 : Chemins possibles (2.5 points)

1. Nous souhaitons tout d'abord déterminer s'il est possible d'aller d'une entrée au Minotaure, puis du Minotaure (vaincu) à une sortie. Écrivez une règle `.rule("chemin", de, vers)` qui retourne vrai lorsqu'il existe un chemin de la pièce `de` à la pièce `vers`. Cette règle **doit** être récursive, c'est-à-dire s'appeler elle-même, mais ne doit pas utiliser de listes.

Une règle peut s'écrire en plusieurs parties. Par exemple, le code ci-dessous retourne dans `res` la valeur de `n`.

```
let n: Term = .var("n")
let m: Term = .var("m")
let x: Term = .var("x")
let res: Term = .var("res")

var kb: KnowledgeBase = [
  .fact("f", Nat.zero, Nat.from(1)),
  .rule("f", Nat.succ(n), res) {
    .fact("f", n, x) && Nat.mul(x,
      Nat.succ(n), res)
  }
]
kb = kb + KnowledgeBase(knowledge: Nat.axioms)
let ans = kb.ask(.fact("f", Nat.from(5), res))
for binding in ans {
  print(Nat.asSwiftInt(binding["res"]!))
}
```

Plusieurs remarques sur ce code :

- Nous définissons des termes pour nos variables auparavant pour simplifier la lecture du code par la suite.
- L'ordre des opérations est très importante! D'abord appeler la factorielle ou la multiplication n'a pas du tout le même effet!
- Pensez bien à fusionner votre base de connaissance avec les types *builtins* si vous voulez utiliser les opérations sur les `Nat` ou encore les `List`.
- Vous pouvez passer facilement d'un terme `Nat` à un `Int` swift avec la commande `Nat.asSwiftInt(Term)`.

2. Écrivez la règle `.rule("itineraire", de, vers, pieces)` qui retourne dans `Pieces` une liste des pièces à parcourir pour aller de `de` à `vers`.

Quelques exemples de manipulation de listes :

```
let res1: Term = .var("res1")
let res2: Term = .var("res2")

let kb: KnowledgeBase =
  KnowledgeBase(knowledge: List.axioms)

// We create two lists
let list1: Term = List.from(elements:
  [1,2,3].map(Nat.from))
let list2: Term = List.from(elements:
  [4,5,6].map(Nat.from))

// We apply different operations on list
let contains = kb.ask(List.contains(list:
  list1, element: Nat.from(1)))
let concat = kb.ask(List.concat(list1, list2,
  res1))
let count = kb.ask(List.count(list: list1,
  count: res2))

// We print results
print("Does list1 contains 1 ?",
  contains.next() != nil)
for binding in concat {
  print("Result of concat: ",
    binding["res1"]!)
}
for binding in count {
  print("Result of count: ",
    Nat.asSwiftInt(binding["res2"]!))
}
```

Une liste est décomposée par une tête de liste et un reste. Vous avez une structure qui ressemble à `cons(head, tail)` avec `head` un élément de la liste et `tail` une nouvelle liste.

Exercice 3 : Survie dans le labyrinthe (2.5 points)

Nous envoyons le Prof. Buchs vaincre le Minotaure. Pour cela, il se repère dans le labyrinthe au moyen d'un smartphone à la pomme. Sa batterie s'épuise rapidement, à cause de la mauvaise qualité de l'appareil. Lorsque la batterie est vide, votre professeur est perdu.

1. À chaque pièce parcourue, la batterie s'épuise d'une unité. Initialement, celle-ci est chargée de 15 unités.

Écrivez la règle `.rule("batterie", pieces, batterie, reste)` qui retourne dans `reste` la charge restante de la batterie après avoir parcouru les `pieces`.

Testez votre programme en utilisant la règle ci-dessous, qui calcule les pièces franchies pour aller de `de` à `vers`, vérifie que la `batterie` est suffisamment chargée, et retourne dans `reste` sa charge restante.

```
.rule("test_batterie", de, vers, batterie, reste)
{
    .fact("itineraire", de, vers, pieces) &&
    .fact("batterie", pieces, batterie, reste)
},
```

2. Écrivez la règle `.rule("chemin_batterie", de, vers, batterie, pieces, reste)` qui retourne dans `pieces` un chemin allant de `de` à `vers`, si la `batterie` le permet, et retourne dans `reste` sa charge restante.

Pour l'instant, ne vérifiez pas que le chemin passe par le Minotaure, ni que `de` est une entrée et `vers` une sortie.

3. Écrivez la règle `.rule("chemin_reussite", batterie, pieces)` qui retourne dans `pieces` un chemin permettant de vaincre le Minotaure puis sortir. Attention à vérifier le chemin part d'une entrée et arrive à une sortie, et que la batterie est suffisante. Utilisez la règle `chemin_batterie`! Testez votre règle avec différentes charges initiales de `batterie`.

Exercise 4 : Guerrier technophile (Bonus)

Le smartphone servira aussi à éclairer la pièce lors du combat contre le minotaure, ainsi qu'à tweeter le résultat du combat. L'éclairage coûte 5 unités d'énergie, et le tweet 2 unités.

1. Écrivez la règle `.rule("reussite_complete", batterie, pieces)` évaluée à vraie pour les chemins permettant de vaincre le Minotaure, tweeter et sortir. `batterie` contiendra la charge initiale de la batterie.
 - Est-il possible pour le Prof. Buchs de vaincre le Minotaure, tweeter et sortir du labyrinthe avec une batterie chargée à 15 ?
 - Est-il possible d'obtenir à partir de cette règle toutes les charges initiales de batterie valides ?
2. Écrivez la règle `.rule("reussite_tweet", batterie, pieces)` retournant vrai pour les chemins permettant de vaincre le Minotaure, tweeter le résultat de la bataille, mais pas de sortir.
 - Quels chemins permettent de vaincre le Minotaure, tweeter, mais pas de sortir avec une batterie de 15 ?

N'hésitez pas à faire un projet swift de votre côté pour tester le fonctionnement de LogicKit. De même il est conseillé d'essayer d'écrire et d'exécuter les règles à la main. Vous aurez la possibilité de mieux vous rendre compte du fonctionnement des inférences.