TP n° 3

Paradigmes et interprétation
Licence Informatique
Université Côte d'Azur

Dans ce TP, on modifie l'interpréteur avec l'ordre supérieur : repartez du fichier `ordresup.rkt`.

Booléens

On cherche à ajouter les booléens, la comparaison sur les nombres et le branchement conditionnel au langage.

```
<Exp> = ...
      | true
      | false
      | {= <Exp> <Exp>}
      | {if <Exp> <Exp> <Exp>}
```

L'égalité ne fonctionne que sur les nombres, vous devez renvoyer une erreur `"not a number"` si l'une des expressions ne s'évalue pas en un nombre.

Le branchement conditionnel `if` évalue son premier argument. S'il s'évalue à `true` le résultat du branchement est l'évaluation du second argument. S'il s'évalue à `false` le résultat du branchement est l'évaluation du troisième argument. S'il ne s'évalue pas en un booléen, vous devrez renvoyer une erreur `"not a boolean"`. Notez que vous ne devez évaluer qu'une seule des branches de l'expression `if`.

Pour implémenter ces fonctionnalités, il vous faudra ajouter les nouvelles expressions au type `Exp` mais aussi modifier le type `Value` pour qu'il supporte les booléens.

Exemples :
`{= 1 2}` s'évalue à la valeur `false`.
`{= true true}` produit une erreur `"not a number"`.
`{if true 1 2}` s'évalue à la valeur `1`.
`{if {= {+ 1 3} {* 2 2}} 4 5}` s'évalue à la valeur `4`.
`{if 1 2 3}` produit une erreur `"not a boolean"`.
`{if true 1 x}` s'évalue à 1 sans produire d'erreur car l'expression incorrecte `x` n'est pas évaluée.

Délier une variable

On souhaite ajouter une expression `unlet` au langage. Celle-ci permet d'oublier la dernière liaison effectuée sur une variable.

```
<Exp> = ...
      | {unlet <Symbol> <Exp>}
```

L'expression `{unlet s e}` retire la dernière liaison de la variable `s` trouvée dans l'environnement (si elle existe) avant de renvoyer la valeur de `e`.

Exemples :

```
{let {[x 1]}
  {unlet x x}}
```

produit une erreur `"free identifier"`.

```
{let {[x 1]}
  {let {[x 2]}
    {unlet x x}}}
```

produit la valeur 1.

```
{let {[x 1]}
  {let {[x 2]}
    {+ {unlet x x} x}}}
```

produit la valeur 3.

Apportez les modifications nécessaires à l'interpréteur.

Glaçons

Un glaçon[1] est une expression dont l'évaluation a été mise en attente. On peut se les imaginer comme des fonctions sans arguments qui attendent d'être appelées pour produire leur valeur. Ajoutez des expressions `delay` et `force` au langage tels que :

- `{delay e}` crée un glaçon autour de l'expression `e` (celle-ci n'est dont pas évaluée).
- `{force t}` force l'évaluation du glaçon `t`. La valeur obtenue doit être celle qu'aurait eu l'expression retardée au moment de sa définition. Si `t` n'est pas un glaçon, une erreur `"not a thunk"` doit être lancée.

Exemples :

```
{delay {+ 1 {lambda {x} x}}}
```

ne produit pas d'erreur car l'expression erronée n'est pas évaluée.

```
{force {delay {+ 1 {lambda {x} x}}}}
```

produit une erreur `"not a number"` car maintenant l'expression est évaluée.

```
{let {[t {let {[x 1]} {delay x}}]}
  {let {[x 2]}
    {force t}}}
```

produit la valeur 1 qui est bien la valeur de l'expression gelée au moment de sa définition.

1. Traduction non officielle, le terme anglais est *thunk*.