TP n° 9

Paradigmes et interprétation
Licence Informatique
Université Côte d'Azur

Ce TP est basé sur l'interpréteur `inherit.rkt`.

Expression `select`

On cherche à réaliser un branchement conditionnel à partir d'une nouvelle expression `select`.

```
<Exp> ::= ...
        | {select <Exp> <Exp> <Exp>}
```

L'expression `{select v obj arg}` appelle la méthode `select-false` de `obj` si `n` s'évalue en la valeur `(numV 0)`. Sinon, c'est la méthode `select-true` de l'objet qui est appelée. Dans les deux cas, les méthodes sont appelées avec l'argument `arg`. On renvoie une erreur `"not an object"` si `obj` ne s'évalue pas en un objet.

```
(test (interp-expr `{select {* 0 1} {new Selectable} {+ 2 3}}
                   (list `{class Selectable extends Object {}
                             {select-true {+ arg 4}}
                             {select-false {* arg 5}}}))
      (numV 25))

(test/exn (interp-expr `{select 0 1 2}
                       empty)
          "not an object")
```

Expression `instanceof`

Dans les langages autorisant l'héritage, on trouve fréquemment une expression pour déterminer si un objet est une instance d'une classe. C'est le cas lorsque l'objet a été créé à partir de la classe ou de n'importe laquelle de ses classes filles.

Implémentez l'expression `instanceof` telle que `{instanceof expr class}` renvoie `(numV 1)` si `expr` est une instance de `class` ; `(numV 0)` sinon. Notez que tout objet est une instance de la classe `Object` et que les nombres, n'étant pas des objets, ne sont instances d'aucune classe.

```
<Exp> ::= ...
        | {instanceof <Exp> <Symbol>}
```

Exemples :

```
(test (interp-expr `{instanceof {new Triple 1 2 3} Singleton}
                   (list `{class Singleton extends Object {x}}
                         `{class Pair extends Singleton {y}}
                         `{class Triple extends Pair {z}}))
      (numV 1))

(test (interp-expr `{instanceof {new Pair 1 2} Triple}
                   (list `{class Singleton extends Object {x}}
                         `{class Pair extends Singleton {y}}
                         `{class Triple extends Pair {z}}))
      (numV 0))
```

Notez que pour réaliser cet exercice, il faudra que les classes connaissent dynamiquement leur classe mère. Il faut donc modifier le type `Class`.

Les nombres en tant qu'objets

Pour l'instant, nous avons deux types de valeurs possibles : les objets et les nombres. Dans de nombreux langages objets, les nombres sont eux-mêmes considérés comme des objets. On souhaite pouvoir les manipuler comme tels et leur envoyer des messages. On considère qu'ils n'ont pas de champ et disposent uniquement des deux méthodes `plus` et `mult`. Les instructions `+` et `*` du langage sont maintenant du sucre syntaxique pour l'appel des méthodes correspondantes : `{+ 1 2}` est équivalent à `{send 1 plus 2}`. Retirez les variantes `plusE` et `multE` du langage et adaptez la fonction `exp-s->e`.

Même si pour l'utilisateur les nombres sont des objets. En interne, par contre, ils disposent d'un traitement particulier. Une expression s'évaluant en un nombre retourne bien un `numV`. Lorsque l'interpréteur rencontre un appel de message sur un nombre, il réalise le traitement approprié (addition, multiplication ou une erreur `"not found"`).

```
(test (interp-expr `{send 1 plus 2} empty) (numV 3))
(test (interp-expr `{send 1 mult 2} empty) (numV 2))
(test/exn (interp-expr `{send 1 div 2} empty) "not found")
```