





```

val x :=
  f (Normal 0) ? {
    | Error => Error
    | Normal rf =>
      g (Normal 1) ? {
        | Error => Error
        | Normal rg => Normal (rf + rg)
      }
  }

```

Pour rendre lisible le code compilé, on peut utiliser les opérateurs suivants (ce sont ceux de la monade dite d'erreur) :

```

val return x := Normal (x)

val bind x f :=
  x ? {
    | Error => Error
    | Normal (y) => f y
  }

```

Le programme compilé précédent s'écrit alors :

```

val x :=
  bind (f' 0) (\x =>
    bind (g' 1) (\y =>
      return (x + y)))

```

(En supposant que  $f'$  et  $g'$  sont les versions compilées de  $f$  et  $g$ .)

1. Comment compiler la construction `error` dans la monade d'erreur ?
2. Comment compiler la construction `try ... orelse ...` dans la monade d'erreur ?
3. Donnez le code de  $P$  compilé avec cette technique.

□

**Exercice 7** (Compilation en style par continuation avec continuation d'erreur)

La compilation par continuation explicite la suite du calcul sous la forme d'une fonction pris en argument de façon systématique par toute sous-expression  $e$  du calcul. Cette fonction attend le résultat de  $e$  pour continuer le calcul. Ainsi, toute expression est transformée en une fonction de la forme :  $\lambda k \Rightarrow \dots k \cdot v \dots$  où  $k$  est une fonction, la continuation du calcul, qui doit être appliquée au résultat  $v$  de l'évaluation de  $e$  pour poursuivre normalement le calcul.

Par exemple, pour compiler

```
f 0 + g 1
```

On écrit :



```

\k =>          ** k is the continuation expecting the result of "f 0 + g 1".
val after_f x := ** x is the result of "f 0".
  val after_g y := ** y is the result of "g 1".
  k (x + y)      ** in the end, the continuation is called with x + y.
in
  g after_g 1    ** The continuation of "g 0" is returning "f 0 + g 1"
in
  f after_f 0    ** The continuation of "f 1" computes "g 1" and returns "f 0 + g 1".

```

Le programme précédent est écrit dans un style dit indirect, ce qui le rend difficile à lire. Encore une fois, une monade peut nous servir à le réécrire de façon plus lisible (Ici, il s'agit de la monade dite de continuation.)

```

val return x :=
  \k => k x
val bind m f :=
  \k =>
    val after_m x := f x k
    m after_m

```

En effet, on obtient alors :

```

val x :=
  bind (f' 0) (\x =>
    bind (g' 1) (\y =>
      return (x + y)))

```

(En supposant que  $f'$  et  $g'$  sont les versions compilées de  $f$  et  $g$ .)

Pour traiter les erreurs, il suffit de se doter d'une continuation supplémentaire : elle représente la continuation à exécuter en cas d'erreur. Ainsi, la monade devient :

```

val return x :=
  \k on_error => k x
val bind m f :=
  \k on_error =>
    val after_m x := f x k on_error
    m after_m on_error
val error :=
  \k on_error => on_error 0
val tryOrElse e on_error :=
  \k old_on_error => e k on_error

```

1. Expliquez le principe de fonctionnement de `error`.
2. Expliquez le principe de fonctionnement de `tryOrElse`.
3. Donnez le code compilé de  $P$  implémenté à l'aide de continuations.



### Exercice 8 (Marquage de pile)

Une autre façon de compiler le mécanisme de lancement/rattrapage d'erreurs nécessite une modification plus profonde du compilateur : on étend chaque langage de HOPIX à RETROLIX avec un mécanisme de lancement et de rattrapage d'erreur.

Comme HOPIX, HOBIX et FOPIX sont des langages à expressions, il n'est pas très compliqué de les étendre avec une construction `error` et une construction `try...orelse` analogues à celles de HOPIX.

Pour RETROLIX, c'est un peu plus compliqué : il faut d'une part étendre l'environnement avec une pile de gestionnaire d'erreurs, et d'autre part, il faut se doter de deux nouvelles instructions.

Tout d'abord, la pile des gestionnaires d'erreurs est constituée de deux types de données : des étiquettes représentant des positions dans le code et des marques d'entrées de fonction. À chaque fois que l'on rentre dans une fonction, on pousse une marque au sommet de la pile. Quand on sort d'une fonction, on dépile tout ce qui se trouve jusqu'à la marque d'entrée de cette fonction.

La première instruction enregistre un gestionnaire d'erreurs. Elle a la forme suivante

```
on_error_jump 11 -> 12
```

Cette instruction met à jour l'environnement d'évaluation de RETROLIX en poussant l'adresse 11 du code du gestionnaire d'erreurs au sommet de la pile des gestionnaires d'erreur puis en continuant l'exécution en 12.

La seconde instruction sert à déclencher une erreur. Elle a la forme suivante

```
error
```

L'exécution de cette instruction procède comme suit :

- Si la pile est vide, le programme est stoppé.
- Si la pile n'est pas vide et que le sommet de la pile est une étiquette 1 de gestionnaire d'erreur alors on saute à cette étiquette.
- Si la pile n'est pas vide et que le sommet de la pile est une marque de début de fonction alors on sort de la fonction courante et on essaie de trouver récursivement un gestionnaire d'erreur dans la fonction appellante.

1. Comment compiler la construction `error` de FOPIX en RETROLIX ?
2. Comment compiler la construction `try ... orelse ...` de FOPIX en RETROLIX ?
3. Proposez une représentation de la pile des gestionnaires d'erreur en MIPS.  
(Indice : Il suffit d'étendre les blocs d'activation.)
4. Comment compiler l'instruction `error` de RETROLIX en MIPS ?
5. Comment compiler l'instruction `on_error_jump 11 -> 12` de RETROLIX en MIPS ?

□

### Exercice 9 (Conclusion)

1. Comparez l'efficacité en espace et en temps de la compilation du mécanisme de lancement d'erreur en fonction des trois techniques de compilation précédentes.
2. Comparez l'efficacité en espace et en temps de la compilation du mécanisme de rattrapage d'erreur en fonction des trois techniques de compilation précédentes.
3. Quelle technique vous semble la plus intéressante à implémenter dans votre projet ? Justifiez votre réponse. N'oubliez pas de prendre aussi en compte le coût de développement de l'extension.

□