

LINFO1104: TP 1

L'objectif de cette séance est de se familiariser avec

- l'environnement Mozart;
- le langage de programmation Oz.
- les *fonctions et procédures récursives*.

Pour cela, vous écrirez de petits programmes et les ferez fonctionner avec Mozart. La dernière version de Mozart est disponible ici: <https://github.com/mozart/mozart2/releases/tag/v2.0.1>. Utilisez le binaire d'installation selon votre système d'exploitation.

- *.deb: paquet destiné aux distributions Linux basés sur Debian/Ubuntu
- *.rpm: pour les distributions linux Fedora/CentOS
- *.dmg: pour MacOS
- *.exe: pour Windows

Pour Linux, il faut manuellement installer emacs et tk (selon la distribution Linux, le nom des paquets à installer changent. Renseignez-vous sur la façon de les installer). MacOS demande également d'installer emacs (ou aquamacs si emacs ne fonctionne pas). Seule la version Windows ne nécessite pas d'intervention manuelle avant l'installation de Mozart2.

Exercices: Premiers pas en Oz

Voici quelques exercices pour vous familiariser à l'environnement et au langage.

1. **Editer et compiler.** Introduisez le code suivant dans l'environnement Mozart :

```
{Browse 1}  
{Browse 2}  
{Browse 3}  
{Browse 4}
```

Ce programme appelle quatre fois la procédure **Browse**. Cette procédure affiche la valeur qui est donnée en argument.

- Exécutez ce programme.
 - Exécutez maintenant ce programme une ligne à la fois. (En sélectionnant la ligne voulue et en cliquant sur "Exécuter la sélection"). Exécutez les lignes dans l'ordre inverse, sans changer le programme. Exécutez les deux dernières lignes en une fois.
2. **Expressions arithmétiques.** Nous pouvons utiliser la procédure **Browse** pour évaluer des expressions arithmétiques plus complexes. Par exemple:

```
{Browse (1+5)*(9-2)}
```

Calculez en vous basant sur l'exemple précédent:

- 19 - 970
- 192 - 980 + 191 * 967
- (192 - 780) * (~3) - 191 * 967

Attention: l'opérateur de négation se note "~".

3. **Entiers et flottants.** Voici un exemple de code incorrect:

```
{Browse 123 + 456.0}
```

Essayez de l'exécuter, puis trouvez deux manières de corriger ce programme.

4. **Variables.** L'instruction **declare** permet de créer une variable et de lui assigner une valeur. Comme dans l'exercice précédent, la valeur peut être donnée par une expression. Par exemple:

```
declare  
X=(6+5)*(9-7)
```

On peut alors utiliser l'*identificateur* de la variable comme synonyme de la valeur. Par exemple:

```
{Browse X}  
{Browse X+5}
```

De même, l'instruction `local ... in ... end` permet de déclarer des variables qui ne sont accessibles que entre le `in` et le `end`.

```
local
  X
in
  X=1
  {Browse X} % Affiche 1
end
{Browse X} % ERREUR: X n'est pas accessible
```

Réeffectuez les calculs de l'exercice précédent, en utilisant des variables là où vous le jugez utile.

5. **Déclarations multiples.** Encodez le programme suivant dans Mozart :

```
declare
X=42
Z=~3
{Browse X}      % (1)
{Browse Z}

declare
Y=X+5
{Browse Y}      % (2)

declare
X=1234567890
{Browse X}      % (3)
```

Ce qui se trouve après le signe `%` est ignoré par le compilateur. Il s'agit d'un commentaire.

- Exécutez le programme *paragraphe par paragraphe* (c'est-à-dire les cinq premières lignes d'abord, puis les trois suivantes, puis les trois dernières). Après cela, le premier `X` est-il toujours accessible ? À votre avis, sa valeur est-elle toujours en mémoire ? Qu'en est-il de `Z` ?
- Réexécutez uniquement la ligne `%(1)`. Quel `X` est affiché ? Où s'est fait le lien entre le `X` et cette valeur ?
- Réexécutez la ligne `%(2)`. La valeur de `Y` a-t-elle été affectée par la seconde déclaration de `X` ? Pourquoi ?

6. **Expressions booléennes.** Outre les expressions arithmétiques, Oz propose des expressions booléennes. Les deux valeurs possibles de telles expressions sont `true` et `false`. Ainsi, la comparaison de deux valeurs entières renvoie un booléen.

```
{Browse 3 == 7}      % egaux
{Browse 3 \= 7}      % différents
{Browse 3 < 7}       % plus petit
{Browse 3 =< 7}      % plus petit ou égal
{Browse 3 > 7}       % plus grand
{Browse 3 >= 7}      % plus grand ou égal
```

Faites quelques tests et observez les valeurs booléennes renvoyées.

7. **Fonctions et expressions.** Nous pouvons aussi utiliser des appels de fonctions dans les expressions arithmétiques et booléennes. Par exemple :

```
{Browse {Max 3 7}}
{Browse {Not 3==7}}
```

La fonction `Max` prend deux arguments et renvoie le maximum de ceux-ci. La fonction `Not` prend un argument booléen et renvoie sa négation. Notez que les arguments des fonctions sont aussi des expressions arithmétiques ou booléennes.

- Utilisez la fonction `Max` dans une expression qui renvoie le maximum entre trois nombres `X`, `Y` et `Z`. Essayez avec `X=7`, `Y=5`, `Z=6`. Essayez également avec d'autres valeurs.
- Construisez une fonction qui renvoie le signe d'un entier `N` passé en argument. Les valeurs possibles du signe sont `0`, `1` et `~1` suivant que `N` est nul, positif ou négatif. Testez votre fonction pour différentes valeurs de `N`.

8. **Portée lexicale et environnement.** Pour rappel, la *portée* d'une déclaration est la zone d'un programme où un identificateur est défini et correspond à cette déclaration. L'*environnement* à un moment donné de l'exécution est l'ensemble des identificateurs définis et leurs variables correspondantes en mémoire.

- Dans le programme suivant, quelle est la portée de chacun des identificateurs déclarés ? Ou, de manière duale, à quelle déclaration se rapporte chaque occurrence d'identificateur ?

```
local P Q X Y Z in      % (1)
  fun {P X}
    X*Y+Z                % (2)
  end
  fun {Q Y}
    X*Y+Z                % (3)
  end
  X=1
  Y=2
  Z=3
  {Browse {P 4}}
  {Browse {Q 4}}
  {Browse {Q {P 4}}} % (4)
end
```

- En utilisant les informations précédentes, prédisez le résultat de l'exécution du programme. Si cela s'avère nécessaire, exécutez le programme à la main, en notant les instructions effectivement exécutées et leur environnement. Finalement, exécutez le code pour confirmer votre bonne compréhension.
- Modifiez le code pour que l'identificateur Y à la ligne *%(2)* corresponde à la variable dénotée par X à la ligne *%(1)*, et ait donc la valeur 1. Vérifiez que l'appel à Browse de la ligne *%(4)* du programme modifié affiche 10.
- En partant du résultat du point précédent, faites de même pour que l'identificateur Z à la ligne *%(3)* corresponde à la variable dénotée par Y à la ligne *%(1)*, et ait donc la valeur 2. Vérifiez que le programme modifié affiche cette fois 9.

9. **L'abstraction procédurale.** Dans cet exercice, vous allez explorer les possibilités des fonctions, afin d'en améliorer votre compréhension. Considérez le programme suivant :

```
declare
X=3

{Browse X+2}      % (1)
{Browse X*2}      % (2)
```

- Construisez une fonction Add2 sans argument, à partir de l'expression X+2 à la ligne *%(1)*. Ensuite, remplacez l'expression de la ligne *%(1)* par un appel à la fonction Add2.
- Construisez maintenant une fonction Mul2 à partir de l'expression X*2 de la ligne *%(2)*. Cette fonction prendra X en paramètre. Comme dans le point précédent, remplacez l'expression de la ligne *%(2)* par un appel à la fonction Mul2.
- Décrivez les fonctions Add2 et Mul2. Quel est leur *environnement contextuel*, c'est-à-dire leurs identificateurs libres et leurs valeurs ?
- Déclarez à nouveau l'identificateur X par l'instruction ci-dessous, *sans* redéfinir les procédures Add2 et Mul2 et sans relancer Oz.

```
declare
X=4
```

Que se passe-t-il si l'on appelle maintenant les fonctions Add2 et Mul2 ?

Exercices: La programmation récursive avec des entiers

Certains exercices mentionnent le concept d'*accumulateur*. Un accumulateur est un argument d'une fonction récursive qui sert à stocker un résultat intermédiaire. Par exemple, la fonction factorielle peut être définie par :

```

fun {Fact N}
  if N==0 then 1 else N*{Fact N-1} end
end

```

Remarquez que la multiplication par N se fait *après* l'appel récursif. Retenir cette multiplication peut consommer beaucoup de mémoire, parce que chaque appel récursif doit retenir une multiplication à faire. On peut éviter ce phénomène en ajoutant un argument à la fonction, qui contient les valeurs successives N , $N*(N-1)$, $N*(N-1)*(N-2)$, etc. Cet argument est appelé un accumulateur.

```

fun {FactAux N Acc}
  if N<=1 then Acc else {FactAux N-1 Acc*N} end
end

```

```

fun {Fact N} {FactAux N 1} end

```

L'appel `{Fact 5}` va se réduire en `{FactAux 5 1}`, puis `{FactAux 4 5}`, puis `{FactAux 3 20}`, puis `{FactAux 2 60}`, puis `{FactAux 1 120}`, puis enfin `120`. Les valeurs successives des paramètres satisfont à l'invariant $N! * \text{Acc} = 5!$. Cette stratégie d'exécution est préférable (voir cours).

1. **Somme de carrés.** Écrivez une fonction récursive `Sum` qui calcule la somme des carrés des n premiers nombres entiers, en commençant par 1. Par exemple, `{Sum 6}` retourne $91 = 1 + 4 + 9 + 16 + 25 + 36$.
 - Donnez une définition mathématique récursive de la fonction *sum*.
 - Traduisez cette définition en une fonction Oz appelée `Sum`.
 - Écrivez une version de la fonction `Sum` qui utilise un accumulateur `Acc`. L'invariant pour cette fonction est $\{\text{Sum } N\} = \{\text{Sum } I\} + \text{Acc}$.
2. **Miroir, mon beau miroir...** Il y a deux opérateurs en Oz pour faire la division des nombres entiers: `div` et `mod`.
 - `div` renvoie la partie entière de la division de deux entiers. Donc, `11 div 4` est 2.
 - `mod` renvoie le reste de la division de deux entiers. Donc, `11 mod 4` est 3.

Écrivez une fonction récursive `Mirror` qui prend un entier positif n et renvoie les chiffres du nombre en ordre inverse. Par exemple, l'appel `{Mirror 12345}` retourne `54321`. Considérez que n ne commence ni ne finit par 0 (zéro). Naturellement, `div` et `mod` sont utiles pour écrire cette fonction.

Aide: utilisez un accumulateur. Quel est l'invariant satisfait par la fonction avec accumulateur?

3. **Univers parallèle (première partie).** Dans un univers parallèle, l'un d'entre vous écrit ce code en réponse à la question 3.

```

declare
fun {Foo N}
  if N<10 then 1
  else 1+{Foo (N div 10)}
  end
end

```

Sa réponse est correcte...

- Quelle était la question?
 - Que renvoie l'appel `{Foo 123456}`?
 - Que renvoie l'appel `{Foo 3211}`?
 - Que renvoie l'appel `{Foo 0}`?
 - Que renvoie l'appel `{Foo ~666}`?
4. **Univers parallèle (deuxième partie).** Une autre étudiante a répondu avec un code alternatif. Son code est le suivant.

```

declare
local
  fun {FooHelper N Acc}
    if N<10 then Acc+1
    else {FooHelper (N div 10) Acc+1}
    end
  end
in

```

```

    fun {Foo N}
      {FooHelper N 0}
    end
  end
end

```

- Cette réponse est-elle correcte également?
- Quelle est la principale différence entre les deux réponses?

5. **Un, deux, trois, nous irons au bois (première partie).** Une procédure est une suite d'instructions qui ne retourne pas de valeur. Par exemple, une procédure qui affiche un nombre n et son successeur peut être définie par

```

declare
proc {BrowseNumber N}
  {Browse N}
  {Browse N+1}
end

```

Écrivez une procédure récursive `CountDown` qui reçoit un entier n et qui compte de n à 0, en affichant chaque nombre compté. Par exemple, `{CountDown 3}` va afficher

```

3
2
1
0

```

Si elle ne l'est pas déjà, rendez votre solution récursive terminale.