Nom :

Prénom :

Uid :

Exercices sur le Front-end

1 Exercice 1. Ajout de l'opérateur $\in$

On souhaite rajouter dans Tiger l'opérateur d'appartenance $\in$. L'exemple ci-dessous montre un cas typique d'utilisation. Au vu de cet exemple il est clair que cet opérateur n'a de sens que lorsque la partie droite représente un ensemble d'éléments (i.e., un tableau).

```
1 let
2   type int_array = array of int
3   var foo := int_array [10] of 51
4 in
5   if 42 ∈ foo
6      then print ("42 is in the array")
7      else print ("42 is NOT in the array")
8 end
```

1. L'introduction de cet opérateur nécessite-t-il de modifier le scanner Flex ? Si oui comment ?

Réponse :

2. Proposez une classe C++ permettant de représenter le nœud de l'AST qui va stocker ce nouvel opérateur. Faites en sorte que cette classe soit le plus simple possible.

Réponse :

3. L'introduction de cet opérateur nécessite-t-il de modifier le parseur ? Si oui comment ?

Réponse :

4. Quel est le type de retour attendu pour l'expression $42 \in foo$?

Réponse :

5. L'opérateur $\in$ ne suit pas les même règles de typage que les opérateurs $+$ ou $<$. Expliquez pourquoi.

Réponse :

6. Donnez un exemple mettant en avant une erreur de typage lors de l'utilisation de l'opérateur $\in$.

Réponse :

7. Proposez un désucrage pour l'exemple donné en début d'exercice.

Réponse :

8. Plutôt que d'intégrer dans le langage le symbole $\in$, une autre idée aurait été d'introduire une fonction *is_in*. Expliquez les problèmes rencontrés si l'on choisit cette approche.

Réponse :

2 Exercice 2. Comme dirait *The Undertaker* Try ... Catch

On souhaite maintenant offrir un mécanisme de try-catch dans Tiger. Pour simplifier le problème on suppose qu'une exception est simplement une expression entière. Le code ci-dessous montre un exemple typique d'utilisation.

```
1  try
2  (
3    /* Some code */
4    throw 42
5  )
6  catch (my_excep)
7  (
8    print_int (my_excep) /* Will print 42 */
9  )
```

1. Avec cette approche, l'instruction *throw* est simplement une expression et peut donc se trouver n'importe ou dans l'AST. Lors de l'étape de liaison des noms que va-t-on maintenant devoir faire ?

Réponse :

2. Une technique similaire est requise pour une autre structure de contrôle dans Tiger : laquelle ?

Réponse :

On décide maintenant que les exceptions ne sont plus des expressions entières mais doivent être déclarées comme un type. L'exemple suivant montre un cas d'utilisation :

```
let
   type out_of_bound = exception
in
  try
  (
    /* Some code */
    throw out_of_bound
  )
  catch (out_of_bound)  ( )
end
```

3. Que doit on maintenant vérifier dans le *throw* et dans le *catch* ? Vous préciserez dans quelle(s) étape(s) du compilateur cette vérification doit être faite ?

Réponse :

4. Dans les langages modernes la présence du try-catch n'est pas requise : cela implique que l'exception peut remonter de un ou plusieurs niveaux jusqu'à trouver un try-catch. Expliquez quels problèmes ont peut rencontrer avec cette stratégie dans notre implémentation.

Réponse :

3 While...Else

Python offre une structure de contrôle un peu particulière le *while ... do : ... else : ....* Cette structure fonctionne de la manière suivante : dès que l'expression conditionnelle devient fausse, la branche *else* (si spécifiée) est exécutée. En revanche, si l'instruction break (ou le déclenchement d'une exception) est exécutée dans le corps du while la branche *else* ne sera jamais exécutée. Ci-après un exemple (a) avec l'affichage associé (b) et une possible traduction (c) dans notre langage préféré (Tiger [1]).

(a)

```
1 n = 3
2 while n != 0:
3   print (n)
4   n -= 1
5 else:
6   print ("Tigrou!")
```

(b)

```
1 3
2 2
3 1
4 Tigrou
```

(c)

```
1  let var n := 3  in
2    while n <> 0 do
3    (
4       print_int(n);
5       n := n -1
6    )
7    else
8    (
9      print ("Tigrou!")
10   )
11 end
```

1. On souhaite introduire cette construction dans notre compilateur. Complétez la BNF suivante en indiquant les mots clefs, les parties qui sont des expressions (*exps*), des instructions (*stmts*) et les parties optionnelles.

```
1 while_else ::=  ..........................................
2                 ..........................................
```

2. Est-il nécessaire de modifier le scanner pour introduire la nouvelle construction ? Si oui comment, si non pourquoi ?

1. Note aux étudiants n'ayant pas fait Tiger : les seules questions "techniques" sont liées au parseur et scanner, tout le reste est lié au cours.

Réponse :

3. Est-il nécessaire de modifier le parseur pour introduire la nouvelle construction ? Si oui comment, si non pourquoi ?

Réponse :

4. Dessinez l'AST pour le programme suivant. (Les noms des nœuds de l'AST doivent être suffisament explicites pour être compréhensibles).

```
let var n := 3  in
  while n <> 0 do
  (
     print_int(n);
     n := n -1;
     break
  )
  else
  (
    print ("Tigrou!")
  )
end
```

Réponse :

5. Que doit faire l'étape de liaison des noms sur la constuction *while ... (...) else (...)* ?

Réponse :

6. Que doit faire l'étape de type checking sur la constuction *while ... (...) else (...)* ?

Réponse :

7. Donnez la nouvelle classe d'AST permettant de représenter la nouvelle construction. Vous penserez à l'équipper pour qu'elle puisse ensuite être utilisable par le reste du compilateur.

Réponse :

8. Proposez désucrage pour *while cond (body) else (elsebody)* qui soit robuste à la présence de *breaks* dans *body*.

Réponse :

On souhaite maintenant augmenter la structure précédente pour capturer les sorties prématurées de body (pour cause de breaks ou d'exceptions) dans une clause **finally** [2].

```
1 while cond do ( body ) else (elsebody) finally ( finallybody)
```

9. Expliquez pourquoi cette construction peut être typée ? **Vous donnerez la règle d'inférence associée.**

2. À note connaissance, aucun langage ne fournit une telle structure

Réponse :