Rapport du projet d'APS
Realisation d'APS0 et APS1

Réalisé par :

HERNOUF Mohamed
BARDOUX Claire

7 octobre 2020

1 Structure du projet

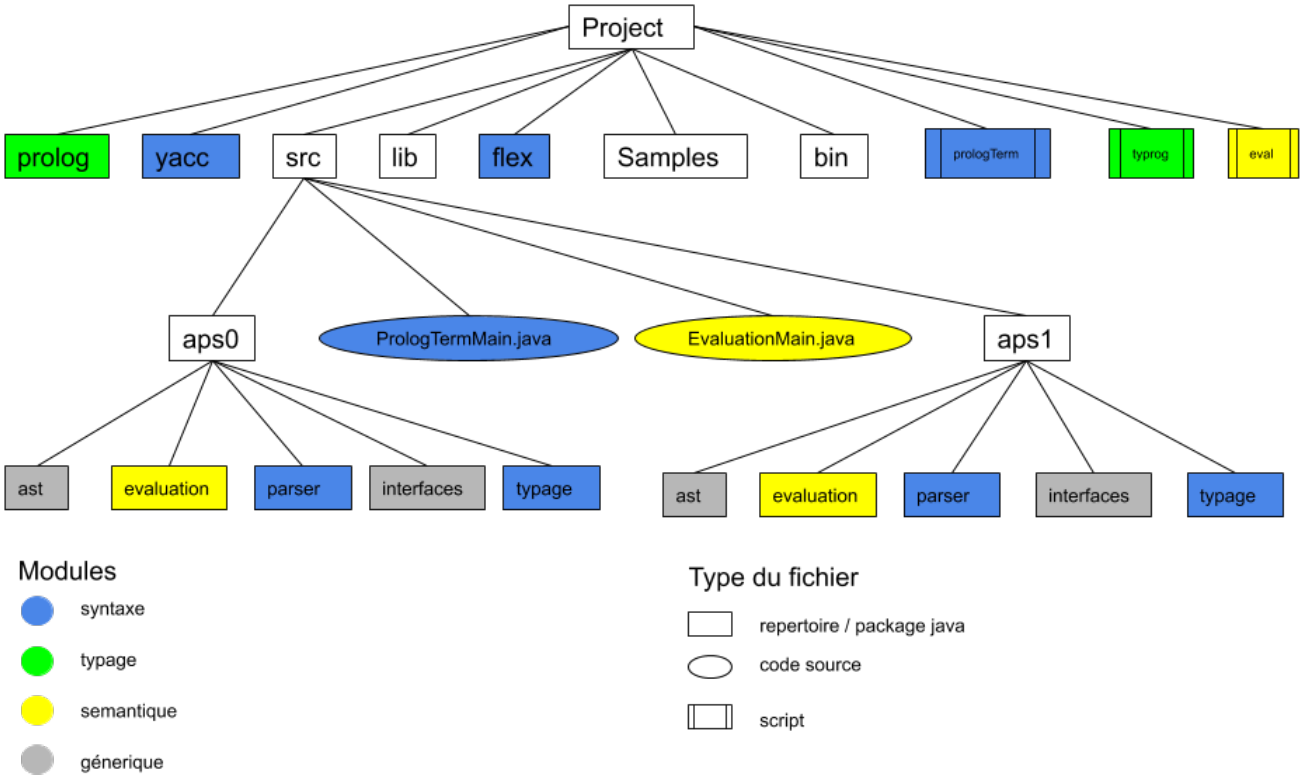

FIGURE 1 – La structure du projet

Dans cette section, on va vous décrire la structure de notre projet, ainsi que les répertoires/fichiers utilisés pour chaque module. La réalisation des modules Syntaxe et Sémantique a été faite entièrement en Java. Pour cela, la structure de notre projet ressemble à celle d'un projet Java ordinaire. Plus précisément, à la racine d'arborescence se trouvent :

- Répertoire *src*, contenant les fichiers sources pour les modules Syntaxe et Sémantique.
- Répertoire *bin*, contenant les fichiers exécutables des fichiers source de Java. Ce répertoire a le même arborescence que celui du src.
- Répertoire lib, contenant les exécutables auxiliaires, comme yacc.linux.
- Répertoire flex, contenant un fichier .flex pour chaque version du APS (APS0 et APS1).
- Répertoire yacc, contenant un fichier .y pour chaque version du APS (APS0 et APS1).
- Répertoire prolog, contenant le fichier judjement.pl donnant les règles de jugement pour le module Typage.
- Les scripts prologTerm, typrog et eval, qui lancent les modules correspondants.

À la racine du répertoire *src*, on trouve deux classes contentant chacune une méthode *main*.La classe *PrologTermMain* affiche à la sortie standard le terme prolog du programme APS, dont le path est spécifié dans les arguments passé au main. La classe *EvaluationMain* exécute un programme APS, en affichant les résultats produits par l'exécution. En plus, on y trouve deux packages : *aps0* et *aps0*. Dans chacun de ces packages, on y trouve :

- Le package **ast**, contenant les types possibles d'AST. Dans *aps0* ce package contient les AST utilisés pour pouvoir supporter la réalisation d'APS0, comme *ASTVar*, *ASTFun*, etc. Dans *aps0* on y ajoute les ASTs requis pour qu'il puisse supporter également APS1, qui ne sont pas présent dans le même package d'*aps0*, comme *ASTProc*, *ASTBlock*, etc. Les classes de ce package sont utilisées dans chaque module.
- Le package **parser**, contenant les classes *Parser* et *Lexer* produite par jflex et yacc. Dans *aps0* ces classes permettent de parser les programmes écrit en APS0 et APS1. Ces classes sont utilisées dans le module Syntaxe.
- Le package **interfaces**, contenant les interfaces Java utiles, permettant de respecter les bonnes pratiques de programmation en Java.

— Le package **typage**, contenant la classe *PrologTerm*. Cette classe est un *visiteur* (cf. section "Choix d'implémentation"), qui parcours récursivement tous les ASTs du programme et construit son terme en prolog. La classe *PrologTerm* du *aps0* hérite de celui d'*aps0* pour réutiliser les méthodes permettant de parcourir les ASTs définis dans le package *aps0*. En plus, il ajoute la possibilité de parcourir de nouveaux ASTs défini dans *aps0*. Cette classe est utilisée dans le module Syntaxe.
— Le package **evaluation**, contenant les classes utilisées lors de l'évaluation du programme APS. Dans le package *value* sont définies tous les valeurs traités par l'évaluateur (comme *Number*, *Function*, *Operation*, etc.). Dans le package **environment** il y a des classes représentant différents états d'environnement (par exemple, *EmptyEnvironnement*, etc.). En plus, ce package contient la classe *Interpreter* qui est lui-même aussi un visiteur. Il parcourt récursivement les ASTs. Lors du parcours, Interpreter évalue les expressions, modifie l'environnement et traite les instructions. Dans *aps0* ce package contient aussi l'implémention de la mémoire du programme (cf. section "Memoire") dans le package **memory**. Les classes de ce package sont utilisées dans le module Sémantique.

2 Choix d'implémentation

2.1 AST

Chaque type d'AST est une classe Java, avec des attributs et des assesseurs. Pour pouvoir parcourir l'ASTs récursivement, chaque AST implémente (indirectement) une interface *IAstVisitable*. Sur la figure 2, on peut voir le diagramme complet des AST existants.

2.2 Le parcours d'un AST

Le parcours d'un AST est effectué grâce à l'utilisation du design pattern *Visitor*. Il existe une interface dans le package interfaces, qui s'appelle *IAstVisitor*, que chaque visitor doit implémenter. Dans le projet, on a défini deux visitors :

1. **PrologTerm** : Il parcourt l'AST et construit son terme prolog.
2. **Interpreter** : Il parcourt l'AST et exécute le programme APS. Chaque méthode *visit* prend en entrée un environnement (cf. section "Environnement"), qu'il, éventuellement, modifie. En plus, chaque méthode *visit* renvoie soit la valeur (cf. section "Les valeurs d'évaluation") soit le nouvel environnement en cas de déclaration. Dans la hiérarchie des classes, ces classes n'ont rien en commun, donc la valeur renvoyé par chaque visite est une instance de la classe Object. Dans le package *aps0* Interpreter contient un attribut *memory* représentant la mémoire, une association entre une adresse en mémoire et une valeur) du programme à un instant donné.

2.3 Jugement des types

Dans le fichier *judjement.pl* se trouve toutes les règles de typage pour le programme en APS1. Pour chacune des six relations de la spécification entre APS0 et APS1 (Prog, Block, Cmds,Dec,Stat,Expr) on a une règle prolog associé (typeProgram, typeBlock, typeCommands, typeDeclaration, typeStatement, typeExpression). Chaque règle traite plusieurs différents cas. Le contexte du jugement des types est implémenté comme la liste des couples. Le premier élément du couple est un symbole, le deuxième est le type qui lui est associé. Le contexte est implémenté comme étant une liste prolog. Le contexte initial correspond au terme "context_init" définit dans judjement.pl. Dans ce fichier, on peut trouver aussi les clauses et termes permettant de manipuler le contexte. La règle *append* permet de concaténer deux listes (donc d'ajouter les symboles avec leurs types à l'environnement). La règle *lookup* permet de trouver le type associé à un symbole. Il y a deux autres règles, permettant de construire la liste des types. Premièrement, c'est la règle *typeArgs* qui permet de récupérer la liste des types à partir de la liste des arguments. Deuxièmement, c'est la règle *typeListExpressions* qui prend en argument la liste des expressions et construit la liste de ces types de même longueur.

2.4 Exception

Dans les cas où quand l'évaluation ou parsing du programme échoue, une exception Java est lancé. On a décidé d'utiliser une classe "Exception" pour indiquer tous les types d'erreur. Le message d'exception indique quel erreur exactement a été produite. Dans la méthode main, on entoure l'endroit susceptible de lancer cette exception avec try/catch. Dans catch on affiche le message associé à cette exception sans afficher le stackTrace.

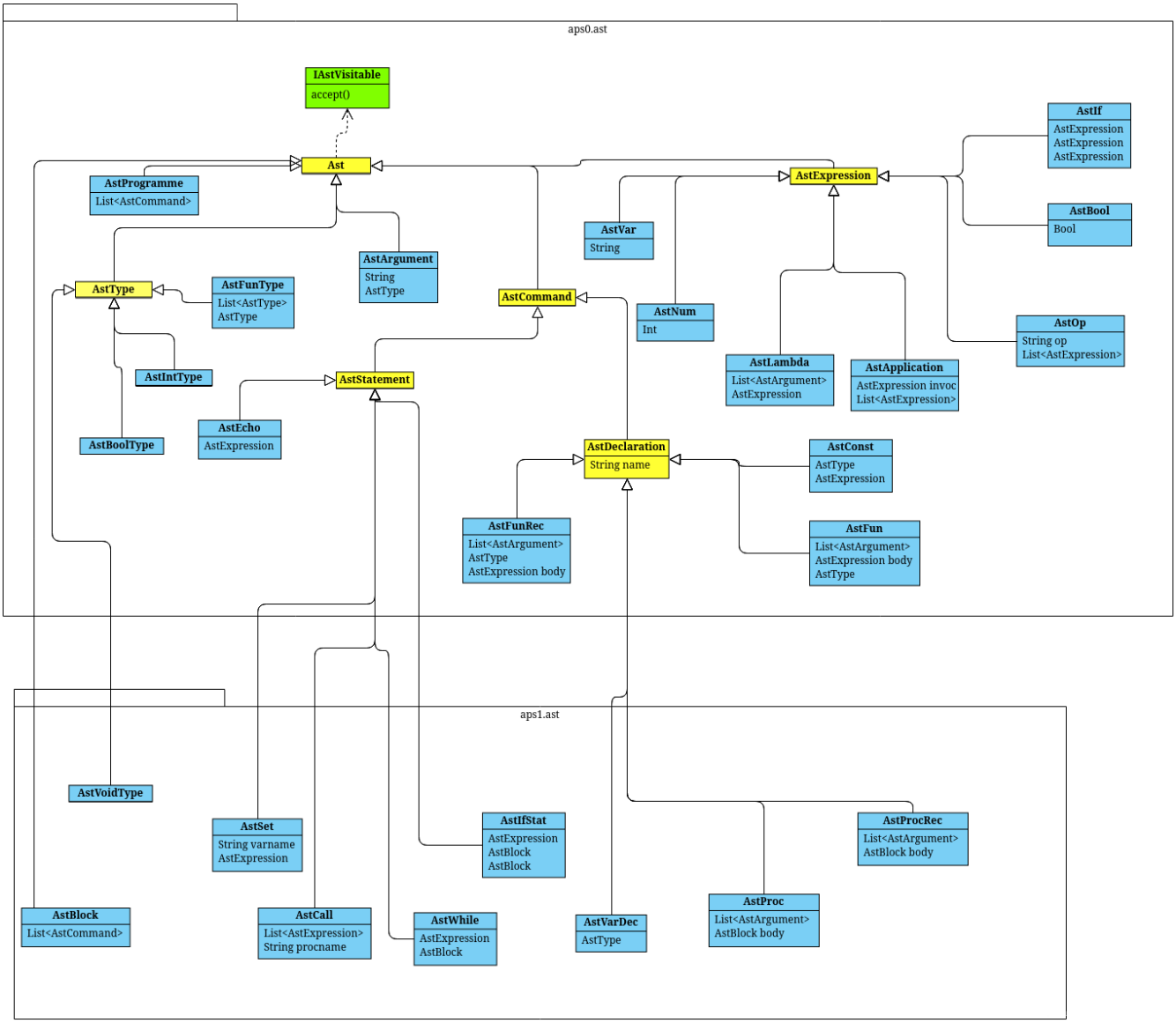

FIGURE 2 – Le diagramme d'AST

2.5 Les valeurs d'evaluation

Lors du parcours de l'AST, Interpreter traite certaines valeurs. Sur la figure 2, on pourrait visualiser la hiérarchie des valeurs existantes. La classe Number correspond à la valeur immédiate. Quand on défini la mémoire du programme à un instant donné, on a besoin d'y sauvegarder deux types de valeurs dedans : la valeur immédiate et la valeur *any* correspondant à la classe Any. Les adresses des cases mémoires sont représentées par la classe Adress.

2.5.1 Invocables

Il y a trois types d'invocables : fonctions, procédures et opérations. Fonctions et procédures peuvent être récursives. Chaque invocable doit définir la méthode getArity() qui retourne nombre d'arguments pour cette invocable. Voici une description de chaque type d'invocable :

— **Fonction** : Valeur fonctionnelle correspond à la classe Function. Au moment de la construction d'une instance de la classe Function, on passe à son constructeur le corps d'une fonction (AstExpression), la liste des arguments (List) et une copie de l'environnement courant (cf. section "Environnement"). Lors d'une application, l'Interpreter cherche la valeur fonctionnelle associé à une variable d'une fonction, et l'applique a la liste des values déjà calculé en appelant la méthode apply. Cette méthode prend aussi l'instance de la classe Interpreter pour pouvoir évaluer le corps de la fonction. L'extension d'environnement pour sauve-

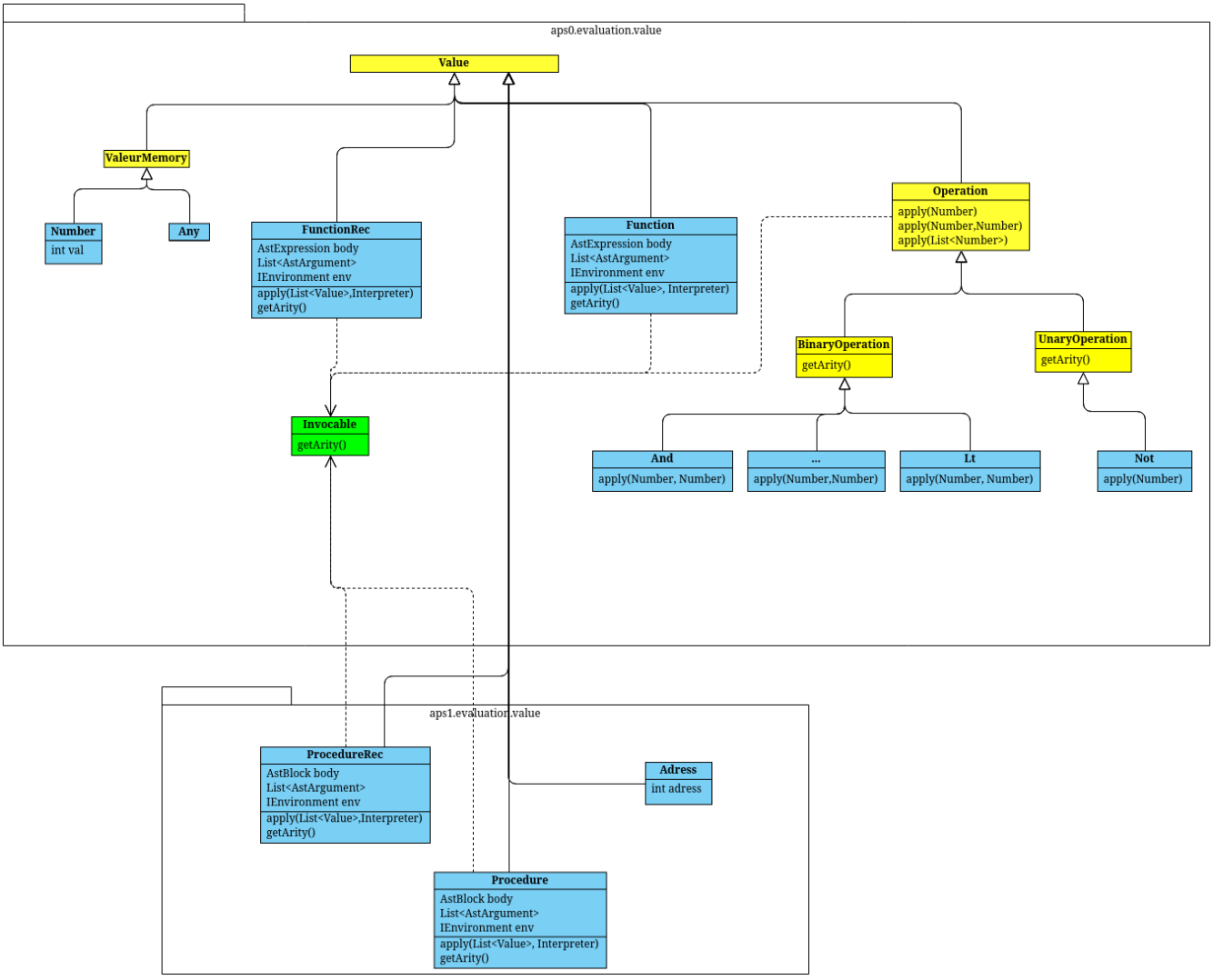

FIGURE 3 – La diagramme des valeurs

garder les valeurs associées à chaque argument est passé dans le corps d'*apply*. Les **fonctions récursives** sont représentées avec la classe FunctionRec. Leurs structure et leur comportement est identique à ceux de classe Function, sauf qu'il possède en plus l'attribut *functionName* et dans le corps d'apply on ajoute dans l'environnement aussi la valeur associé à la fonction elle même (cela est fait avec this).

— **Procedure**. La valeur procédural correspond à la classe Procedure. Procedure récursive est représenté avec classe ProcedureRec. Leur comportement et leur structure ressemblent à ceux de Function et FunctionRec, sauf que leur corps n'est plus AstExpression, mais AstBlock.

— **Operation** : On dispose une classe abstraite Operation. C'est une classe générique permettant de représenter toutes les opérations. Ces sous-classes sont UnaryOperation et BinaryOperation, qui sont aussi les classes abstraites permettant de réunir les opérations avec le même arité. Chaque opération est une classe Java à part, qui hérite une parmi ces classes. Le comportement de chaque opération est défini dans sa méthode apply. La différence entre opération et fonction, c'est que l'opération prend la liste des valeurs immédiates.

2.6 L'environnement

L'environnement représente un contexte au niveau de spécification. L'interface Java dont le nom est "IEnvironement" représente l'environnement avec ces fonctionnalités, comme la recherche d'une valeur et l'extension d'un environnement. Les classes implémentant cette interface sont EmptyEnvironment et Environment. Elles représentent les états remarquables d'environnement. Quand on ajoute les variables dans l'environnement, le nouvel environnement est construit. Cela nous permet de restaurer l'environnement précédent quand on sort de l'appel de fonction,

par exemple. Voici les corps de ces deux classes :

```
public class Environment implements IEnvironment {
  String varName;
  Value value;
  IEnvironment next;

  public IEnvironment extend(String varName, Value value){
      return  new Environment(varName, value,this);
  }

  public Value find(String varName) throws Exception {
      if(getVarName().equals(varName)) return getValue();
      else return next.find(varName);
  }
}
```

Quand on essaie de rechercher la valeur d'une variable dans environnement vide, l'exception est lancé.

```
public class EmptyEnvironment implements IEnvironment {

  public Environment extend(String varName, Value value){
      return  new Environment(varName, value,this);
  }

  public Value find(String varName) throws Exception {
      throw new Exception(varName+ " is not in the environment");
  }

}
```

2.7 La mémoire

La mémoire du programme est représentée par la classe Memory qui implémente l'interface IMemory. C'est une classe qui contient une HashMap et l'utilise pour gérer,ajouter et récupérer les valeurs associé à un adresse. L'objet IMemory est un attribut de l'Interpreter d'APS1. Memory permet de créer l'unicité de chaque adresse grâce à son attribut *adresse_count*.

```
public class Memory implements IMemory{

  private static int adresse_count = 0;
  private Map<Adress,ValeurMemory> memory = new HashMap<Adress,ValeurMemory>();

  public Adress alloc() {
    Adress adr = new Adress(adresse_count++);
    memory.put(adr, new Any());
    return adr;
  }

  @Override
  public void affect(Adress adresse, Number number) throws Exception {
    if(memory.containsKey(adresse)) {
      memory.put(adresse, number);
    }
    else {
      throw new Exception("Segmentation fault");
    }
  }

    @Override
  public Number get(Adress adresse) throws Exception {
    ValeurMemory val = memory.get(adresse);
    if(val != null && val instanceof Number) {
      return (Number) val;
    }
    else {
      throw new Exception("Segmentation fault : Variable at adresse " + adresse.getAdress() + " is
     not initialized");
    }
  }
}
```

Nous avons divergé des spécifications prescrites car au lieu de créer à chaque fois une nouvelle mémoire, nous avons une seule et unique instance de *Memory* que nous modifions à chaque fois qu'on veut allouer un espace en mémoire. Mais cette divergence n'a pas d'impact sur l'évaluation, car dans les spécifications nous n'avons pas la possibilité de libérer une allocation en mémoire.

3 Commentaires

Lors de la dernière phase nous avons effectué des test sur les fichiers se trouvant dans le répertoire *Samples*. Nous avons pu constater qu'une grande majorité des test passent avec succès. Néanmoins il y trois cas qui ne passent pas, voici les raisons de chaque cas :

— **Premier cas :** `./typrog Samples/Samples_APS1/prog114.aps`

```
(lt false true)
```

Ce fichier ne réussit pas nos test étant donné que le typage de l'opération *lt* échoue car dans l'exemple, nous lui transmettons des booléens alors que dans les spécifications *lt* on traite des entiers.

— **Deuxième cas :** `./eval Samples/Samples_APS1/prog105.aps`

```
[
VAR x int;
VAR y int;
SET x 42;
IF (eq x 42)
   [ ECHO x ]
   [ SET y 42 ];
ECHO y          <--
]
```

Ce fichier ne réussit pas car le programme veut afficher la valeur de la variable y alors que nous avons juste initialisé la variable sans lui donner de valeur. Nous obtenons alors cette exception : *"Segmentation fault : Variable at adresse 1 is not initialized"*

— **Troisième cas :** `./eval Samples/Samples_APS1/prog106.aps`

```
[
VAR x int;
VAR y int;
IF (eq x 42)  <--
   [ ECHO x ]
   [ SET y 42 ];
ECHO y
]
```

Ce troisième cas est exactement identique au deuxième : il veut comparer deux valeurs alors qu'une des deux n'a pas encore de valeur. Nous obtenons alors cette exception : *"Segmentation fault : Variable at adresse 0 is not initialized"*

Pour conclure, ces trois erreurs sont tout à fait normales car elles sont là pour vérifier que notre programme signale une erreur quand une de ses expressions ne respectent pas les spécifications.