Rapport travaux d'études et de recherche : Walnut

QUIGNON Nabyl - MESTAIS Loïc - RAGAVAN Mugilan -
SALEH KASEM Raphaël
M1 IMIS
Année universitaire 2022/2023

Contents

1 Objectifs du TER

L'objectif de ce TER est un travail de refactorisation et d'amélioration de l'outil Walnut du point de vue de son interface avec l'utilisateur. Il s'agit de rationaliser le code source et d'en améliorer les entrées et sorties. Cela nécessite de comprendre son fonctionnement, sa finalité et de développer quelques exemples pour guider la démarche.

2 Walnut, c'est quoi ?

2.1 Description de l'outil

Walnut, développé par J. Shallit et al, est un logiciel conçu pour faciliter la rédaction de preuves mécaniques dans le domaine de la combinatoire des mots. Il s'agit d'une calculatrice spécialisée dans le théorème de Cobham-Semenov, qui établit l'équivalence entre différents formalismes utilisés pour décrire des ensembles d'entiers. Ces formalismes comprennent les automates finis qui reconnaissent les nombres en base k, certaines logiques du premier ordre, ainsi que les substitutions de longueur constante sur un alphabet fini.

2.2 Arithmétique de Büchi

Walnut est très utile pour conjecturer sur l'arithmétique de Buchi qui est la théorie du premier ordre des nombres entiers naturels muni de l'addition et de la fonction Vk(x) qui est la plus grande puissance de k qui divise x.

2.3 Syntaxe de Walnut

Pour bien comprendre la suite du rapport, il est nécessaire de savoir comment sont définis les différents éléments d'une formule du premier odre dans le logiciel.

En ce qui concerne les opérateurs logiques, ils sont définis de la manière suivante :

Les opérateurs d'arité 1 :

- "∼" : NOT
- " ‘ " : INVERSE

Les opérateurs d'arité 2 :

- "&" : ET
- "|" : OU
- "∧" : XOR
- "=>" : IMPLIQUE

- "<=>" : EQUIVALENCE

Les quantificateurs :

- "E" : EXISTENTIEL
- "A" : UNIVERSEL
- "I" : INFINITE (retourne vrai s'il existe une infinité de valeur pour la variable associée vérifiant la formule F)

3 Exercice de compréhension

Walnut est un outil pour manipuler :

- des coloriages automatiques (DFAO, morphismes) ;
- des prédicats (DFA, formules du premier ordre) ;
- les combiner pour définir de nouveaux prédicats ;
- vérifier la valeur de vérité d'un prédicat sans variable libre ;
- calculer les valeurs qui satisfont un prédicat.

Pour bien comprendre le fonctionnement du logiciel et la manipulation de ces objets, M.Ollinger nous a proposer une série d'exercices.

Le but de l'un d'eux état de construire avec Walnut un automate qui reconnait le plus petit k pour lequel Thue-Morse est k-synchronisant.

Définition : Un mot binaire infini est k-synchronisant si chaque sous-mot de k bits est identique à un autre sous-mot de k bits situé à une position de même parité dans le mot. (c-à-d lorsque nous retrouvons un sous-mot de k bits identiques à un autre, ils sont dans le même état : ce qui implique que les prochaines lettres seront les mêmes).

3.1 La suite de Thue-Morse

$$\begin{aligned} t &: a \mapsto ab, \qquad b \mapsto ba \\ \mathcal{T} &= abbabaabbaababbabaababbaabbabaab... \end{aligned}$$

La suite de Thue-Morse est une séquence binaire infinie qui est construite de manière itérative. La construction de la suite de Thue-Morse commence par une séquence de départ de longueur 1, généralement représentée par le symbole '0'. À chaque étape suivante, la séquence existante est doublée en ajoutant sa négation (0 devient 01 et 1 devient 10) de manière récursive. Par exemple,

la deuxième étape donne '0110' (0 suivi de sa négation 1, puis 1 suivi de sa négation 0).

Ce processus se poursuit indéfiniment, en ajoutant chaque fois la négation de la séquence existante à la fin de la séquence en cours de construction. La suite de Thue-Morse complète est donc une séquence infinie de 0 et de 1 qui semble aléatoire, mais qui possède des propriétés intéressantes et des motifs récurrents.

La suite de Thue-Morse est une 2-substitution et est une suite automatique (calculable par un automate fini).

3.2 Les différentes fonctions de Walnut

Walnut met à disposition une série de fonction pour manipuler les objets décris précédemment.

Par exemple, la fonction "def" nous permet de décrire un prédicat et de le garder en mémoire pour l'utiliser plus tard.

Quant à la fonction "eval", elle nous permet d'évaluer un prédicat.

Dans les deux, Walnut traduit ces prédicats en automate qu'il pose dans le dossier Result sous forme de texte et d'image (.gv).

Il existe d'autres fonctions mais elles ne sont pas nécéssaires à la compréhension de la suite de l'exercice.

3.3 Exercice

Le but de l'exercice est de construire avec Walnut un automate qui reconnait le plus petit k pour lequel Thue-Morse est k-synchronisant.

Pour cela, nous allons définir des prédicats grâce à la fonction "def" et au mot T représentant le mot de Thue-Morse intégré dans le logiciel.

Pour définir un **facteur** (qui sont deux sous-mots de k-bits identiques et de même longueur) nous avons définis le prédicat **Factoreq** de la manière suivante :

```
def factoreq "Ak k<n =>T[i+k]=T[j+k]":
```

Factoreq retourne TRUE tel que pour i, j, n donnés il existe 2 séquences partant de i et j égales et de même longueur pour tout k <n.

Ensuite, nous avons définis le prédicat **parite** car, si Thue-Morse est k-synchronisant alors les positions i et j précédentes ont la même parité (c'est à dire pair ou impair).

```
def parite "Ea Eb Ec i=2*a+b & j=2*c+b":
```

En assemblant ces prédicats, nous pouvons définir la synchronisation :

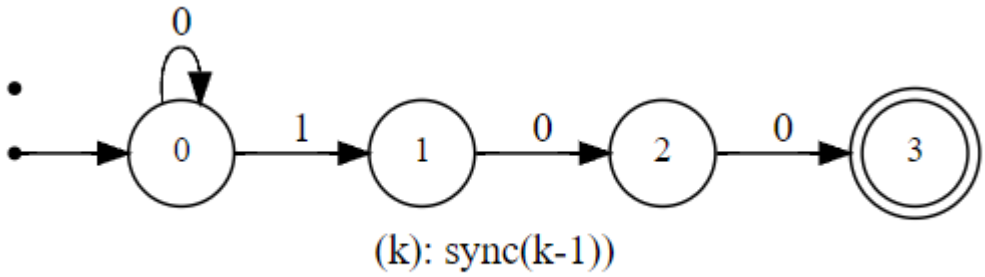
```
def sync "Ai Aj $factoreq(i,j,k) =>$parite(i,j)":
```

Si nous évaluions la fonction sync avec Walnut, il nous dirait d'ores et déjà que la suite de Thue-Morse est synchronisante. Encore faut-il trouver le plus petit k pour laquelle elle l'est.

Pour cela, rien de plus simple : nous évaluions le prédicat suivant :

eval Resultat "$sync(k) & ($sync(k-1))":

En sortie, Walnut nous produit l'automate suivant :



Nous avons utilisé la base msd_2 qui est le binaire avec le bit de poids fort à gauche : nous obtenons une série de 0 inutile et 100 qui représente $2^2 = 4$.

Donc la suite est 4-synchronisante.

4 D'une formule vers un prédicat

Nous savons définir des prédicats et les évaluer, mais que se passe-t-il dans le logiciel pour que, lorsqu'on rentre une commande dans le terminal Walnut, cette commande soit correctement traduite et évaluer.

4.1 La notation postifxée (polonaise inverse)

Pour celà, Walnut utilise la notation postfixée ou polonaise inverse. C'est une méthode de notation mathématique où les opérateurs sont placés après leurs opérandes. C'est à dire, chaque opération est effectuée sur son nombre d'arité de valeurs qui la précèdent immédiatement. Cela permet d'éviter l'utilisation de parenthèses pour indiquer l'ordre des opérations, car l'ordre est déterminé par la séquence des opérations dans l'expression.

$$((x * 3) - ((y + z) - 4)) \mapsto x3 * yz + 4 - -$$

Par exemple, considérons l'expression mathématique en notation infixée suivante : ((x*3) - ((y+z) - 4)). En notation postfixée, cette expression serait écrite comme suit : x 3 * y z + 4 - -. Pour évaluer cette expression, on procède comme suit :

- On dépile les éléments jusqu'à voir une opération. Ici, nous appliquons la multiplication sur x et 3 qui font 3*x.
- En continuant, nous appliquons l'addition sur y et z qui font y+z.
- Ensuite, nous trouvons le moins qui est donc appliqué aux deux derniers éléments : y+z et 4 qui font (y+z) - 4).
- Enfin, nous retrouvons un moins appliqué aux éléments : 3*x et (y+z) - 4) qui font ((x*3) - ((y+z) - 4)).

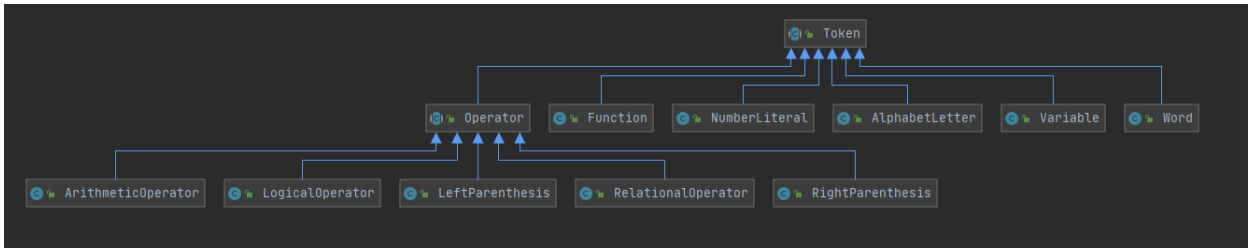
4.2 La transformation

Donc, lorsque nous rentrons une formule en chaîne de caractères, Walnut utilise des expressions régulières afin de déterminer le type de fonction : def, eval, debug... Ce travail est réalisé par la classe **Prover.java** qui découpe la commande en plusieurs groupes : fonction, nom, variables libres etc.. et qui redirige vers les méthodes correspondantes en fonction des éléments.

Elle redirige notemment vers la classe **Computer.java** qui est la classe qui appel la classe **Predicate.java** et qui effectue les calculs pour obtenir un automate à la fin.

Predicate.java est la classe qui va définir le prédicat en notation postfixée à partir de la chaîne de caractères passée en paramètre. Plus précisément, c'est la méthode tokenize_and_compute_post_order() qui se charge de la traduction. Le processus est simple : Toujours à l'aide d'expressions régulières, la méthode cherche les éléments qui compose la formule et crée des **Tokens** qu'elle introduit dans la notation postfixée à l'aide de la fonction put() proposée par la classe **Token.java**.

Token.java est une classe abstraite qui sert à définir les éléments d'une formule. Plusieurs classes concrètes étendent la classe Token.java :



Finalement, lorsque **Computer.java** reprend la main sur l'exécution, il retrouve un prédicat traduit en notation postifxée qui se présente sous la forme d'une liste de Token.

Le travail de **Computer.java** se résume désormais à parcourir la liste des tokens, de les interpréter et de faire évoluer sa pile d'expression (**Expression.java** est une classe permettant de stocker les sous-formules et de les associer à des automates).

4.3 Interprétation

Computer.java parcourt donc la notation postfixée composée de Token et interprète chacun d'eux. Pour cela, Token propose une méthode act() qui est surchargé par chacune des classes concrètes qui l'étende. Évidemment, lorsque nous croisons un opérateur, le travail est plus important que lorsque nous croisons une simple variable. L'opérateur doit dépiler la pile d'expression et d'appliquer son opérateur sur les éléments de la pile d'expression qui le précède (la manière d'appliquer un opérateur et différent en fonction du type de Token qui sont concernés par celui-ci, ce qui est fastidieux à définir).

Prenons l'exemple des classes concrètes suivantes :

- **Variable.java** : c'est un token dont la fonction act est simplement de pousser une nouvelle Expression définie par le nom de la variable dans la pile d'expression.
- **ArithmetiqueOperator.java** : sa méthode act() consiste en dépiler un nombre astucieux (en fonction de l'arité de l'opérator) d'expression et d'appliquer l'opérateur sur ces derniers. A l'issu de la méthode act(), est ajouté à la pile d'expression une autre expression contenant un automate vérifiant le nouveau prédicat. Par exemple, je croise une addition, je dépile 3 et x, je crée une nouvelle expression et je lui associe l'automate vérifiant 3+x.

—

4.4 Documentation

Nous avons crée une documentation html qui reprend plus finement l'explication du passage d'une chaîne de caractère à un prédicat en notation postfixée et son évaluation ainsi qu'une explication plus détaillée sur les méthodes act() et put() en fonction du type de Token. Vous trouverez en annexe, les pages html qui raconte ce qui n'est pas dit dans ce rapport.

4.5 Ajout du mode debug

Nous avons introduit un mode debug qui permet de suivre l'évolution du calcul/interprétation des tokens et leur type dans **Computer.java**.

De manière automatique : L'exécution se fait normalement sans l'intervention de l'utilisateur , toute les deux secondes une itération est effectuée et la pile

d'expression est affichée.

De manière manuelle : Pour cela, à chaque itération, nous ajoutons un scanner qui nous permet de s'arrêter dans l'exécution et de permettre à l'utilisateur de consulter :

- La notation postfixée
- La pile d'expression
- Les deux

De plus, nous avons permis à l'utilisateur de poursuivre le calcul ou de rester dans une configuration. Voici comme cela se présente (retrouvable sur la documentation html) :

Example of computation :

If I enter the command : "debug test "x+y=5";

Reading the token : x type : Variable
expression stack : [x]

Reading the token : y type : Variable
expression stack : [x, y]

Reading the token : +_msd_2 type : ArithmeticOperator
expression stack : [(x+y)]

Reading the token : 5 type : NumberLiteral
expression stack : [(x+y), 5]
computed ~:1 states - 25ms computed ~:2 states - 2ms

Reading the token : =_msd_2 type : RelationalOperator expression : [(x+y)=5]

5 Ajout de fonctionnalitées

5.1 Jline

Il faut savoir que Walnut se présente comme un terminal. Auparavant, il n'était pas possible de naviguer dans une formule avec les touches du clavier, de retracer les commandes entrées, ni d'auto-compléter certaines choses pour faciliter la vie de l'utilisateur.

A l'aide de Jline 3, nous avons donc implémenter ces fonctionnalitées en agissant directement sur la méthode readBuffer et en créant un terminal Jline qui possède ces propriétées.

Pour se faire, on crée un objet Terminal qui va representer un terminal virtuel ce qui va permettre d'avoir un historique de commande et permettre la navigation dans la ligne, on crée ensuite un objet Completer qui va s'occuper

de l'auto-complétion (on écrit les mots clés que l'ont souhaite directement dans le code, dans notre cas "eval", "def", "debug").

On utilise ensuite un objet LineReader qui va utiliser le Terminal et le Completer pour lire les actions de l'utilisateur.

5.2 Le quantificateur S

La partie la plus compliqué arrive ici, nous avons mis en place un nouveau quantificateur S tel que Sx F(x) renvoie la plus petite valeur de x pour laquelle F est satisfaite.

Les quantificateurs sont considérés comme opérateurs logiques. Lorsque **Computer.java** rencontre un opérateur logique, il lui applique la fonction act() décrite précédemment. Cependant, les quantificateurs sont redirigés vers une autre méthode appellée actQuantifier().

La redirection état :
if(op.equals("E") || op.equals("A") || op.equals("I")actQuantifier(S,print,prefix,log);return;

et est devenue :
if(op.equals("E") || op.equals("A") || op.equals("I") || op.equals("S"))actQuantifier(S,print,prefix,log);return

La traduction de Sx, formellement, est : $\forall y(\ (F(y) \Longrightarrow x \leq Y)\ \&\ F(x))$
Donc notre objectif est :

- De créer un nouvelle variable unique correspondant à y.
- De créer l'automate vérifiant F(y) à partir de F(x) défini par l'automate M dans actQuantifier().
- De créer l'automate vérifiant $x \leq Y$.
- De les assembler pour obtenier cette formule.

Création de F(y) : Pour cela, nous créons une variable temporaire "variabletemporaire" correspondant à y. Nous clonons l'automate vérifiant F(x) et créons un nouvel automate à partir de Z. Avec la fonction bind() nous remplaçons l'ancienne valeur x par la nouvelle : "variabletemporaire". Le résultat de bind() nous renvoie un automate avec x et "variabletemporaire" en variable libre, donc nous ajoutons un quantificateur universelle sur x afin de n'obtenir seulement "variabletemporaire" en variable libre. Nous avons donc F(y).

Création de l'automate vérifiant $x \leq Y$: Pour cela, nous créons un nouveau prédicat de type **Predicate.java** qui va traduire en notation postfixée, puis nous poussons nos deux variables x et "variabletemporaire" dans une pile d'expression temporaire et appliquons le RelationOperator $\leq$ créé dans la foulée. Cela nous renvoie une expression dont l'attribut M et l'automate associé. Nous avons donc l'automate reconnaissant $x \leq Y$.

Assemblage : D'aptrès la définition formelle, soit l'automate Z vérifiant F(y) (ou F("variabletemporaire")), l'automate B vérifiant x $\leq$ Y et l'automate M vérifiant F(x). A l'aide des fonctions, il suffit de faire :

Z = Z.imply(B,print,prefix,log); puis Z = Z.and(M,print,prefix,log);

On ajoute à cela le quantificateur universel :

Z.not(print,prefix+" ",log);
Z.quantify(new HashSet¡String¿(bla),print,prefix+" ",log);
Z.not(print,prefix+" ",log);

La fonction quantify concerne l'opérateur existentiel donc il faut passer par la négation pour le quantificateur universel. Le code correspondant est trouvable en annexe.

Vérification :
Nous pouvons vérifier que cela fonctionne en utilisant l'exemple de l'exercice précédent. Pour déterminer le plus petit k pour lequel la suite de Thue-Morse était k-synchronisante nous sommes passés par le prédicat :

eval Tadam "Sk $sync(k)";

et cela fonctionne.

6 Répartition du travail / Fonctionnement du groupe

Le groupe fonctionne principalement avec des réunions sur discord, nous a travaillé à 4 sur l'exercice portant sur Thue-Morse, sur la compréhension du code, le codage du quantifieur S ainsi que la rédaction de ce rapport.Pour le reste du travail,nous nous sommes partagés les tâches :

- La partie Jline avec les améliorations du terminal a été faite par Nabyl
- Le mode Debug a été fait par Mugilan
- La documentation des Tokens en html a été faite par Raphaël et Loïc

7 Annexe

7.1 Site

AlphabetLetter.java

Act()

Adds to the expression stack a new expression whose value is "@"+Integer.toString(value).

Put()

Add the token to List postOrder which contains the postfix notation.

Previous : Token.java

Function.java

Act()

Checks if the Expression Stack S has enough expressions to match the arity of the Function. Creates a temporary stack to store the expressions unstacked from S. Also creates a List of Expressions args and two Lists of String indentifiers and quantify.A is an automaton which represents the function. Then loops on every expression in the temporary stack and if the expression is of type :

- Variable : Adds the variable to an identifier list, if the variable is already in, creates a new identifier for the variable
- Arithmetic : Extracts the Automaton bound to the expression and concatenates it to the M automaton. The quantifiers and identifiers of the extracted automaton from the expression are also added to the identifiers and quantify lists
- Number Literal : Creates an automaton which accepts the value of the literal and concatenates it to M. Also adds the identifiers and quantifiers of the number literal to the corresponding lists
- Automaton : Concatenates the automaton to M. If it is possible, adds its identifiers to the identify list otherwise it throws and Exception.

At the end of the loop, it binds the extracted identifiers to A, concatenates A to M and adds a new Expression to the stack S.

Previous : Token.java

LeftParenthesis.java

Act()

Nothing.

Put()

Add the token to List postOrder which contains the postfix notation.

Previous : Token.java

LogicalOperator.java

Act()

Acts differently depending on the logical operator processed:

- For the quantifiers A, E, I, S: we call the actQuantifier() function. This method implements the processing of universal and existential quantifiers (symbolized by the letters "A" and "E" respectively) for a formal language. Processing these quantifiers requires finding all sets of quantified variables and then applying the appropriate operator on a regular expression. The code starts by unstacking the S-stack and storing the scrolled expressions in a temporary "temp" stack. It then checks that the stored expressions are indeed variables if it is a universal quantifier or a regular expression if it is an existential quantifier. The variables are stored in a list of strings "list_of_identifiers_to_quantify", which then allow to specify the quantified variables. If the operator is "E", the regular expression M is quantified for the identified variables. If the operator is "A", a negation of the regular expression M is performed, followed by the quantization of M on the identified variables, then a new negation. If the operator is "I", M is simplified by removing unnecessary variables, then an infinite regular expression is generated from M. Finally, the complete expression is stored in the stack S and the method returns a string describing the operation performed.
- Else, If the operator is a binary operator, two expressions are unpacked from the stack. If the two expressions are finite automata (Type automaton), the binary operation is applied (and, or, xor, imply, iff) by calling the appropriate method on the automaton. The result is then stacked on the stack as a new expression. If one or both expressions are not finite automata, an exception is thrown. Finally, the method checks if the print option is enabled and if so, it displays the output of the method in the console.

Put()

Add the token to List postOrder which contains the postfix notation.

Previous : Token.java

NumberLitteral.java

Act()

Adds to the expression stack a new expression whose value is Integer.toString(value).

Put()

Add the token to List postOrder which contains the postfix notation.

Previous : Token.java

RelationalOperator.java

Act()

Various cases :

- If a and b are both number or letter literals, the method constructs a new expression by concatenating them with the given operator and creates a finite automaton that matches the evaluation of the expression. This new expression is then stacked on the stack S.
- If a and b are a combination of word and arithmetic/variable, the method constructs a finite automaton that corresponds to the evaluation of the expression using the "comparison" method of "number_system" which returns a finite automaton representing the comparison of the two expressions according to the given operator.
- If a is a number or letter literal and b is of type arithmetic/variable, the method creates a finite automaton representing the comparison between the literal and the expression b, also using the finite automaton corresponding to the expression b.
- If a is of type arithmetic/variable and b is a number or letter literal, the method creates a finite automaton representing the comparison between the expression a and the literal, also using the finite automaton corresponding to the expression a.
- If a and b are both of type word, the method creates a finite automaton representing the comparison between the two words, also using the finite automata corresponding to the expressions a and b to build the final automaton.

Put()

Add the token to List postOrder which contains the postfix notation.

Previous : Token.java

RightParenthesis.java

Act()

Nothing.

Put()

The method performs a while loop as long as the operator stack is not empty. If the operator at the top of the stack is not a left parenthesis, it is removed from the stack and added to the postOrder list. If the operator is a left parenthesis, it is removed from the stack and the method ends. If the stack is empty and there is no left parenthesis to match a right parenthesis, the method throws an exception reporting that the parentheses are not balanced at the position specified in the predicate.

Previous : Token.java

Word.java

Act()

Checks if the expression stack S has enough expressions to match the number of indices, then creates a temporary expression stack to store it.Creates an automaton M which will be used to have the final result and two lists of strings identifiers and quantifiers . Then it loops through the expressions in the temporary stack from the top to the bottom. For each expression, it operates differently depending on the type of the expression.

- Variable : Adds the variable to the identifiers list, if the variable is already in, creates a new identifier for the variable.
- Arithmetic : Extracts the Automaton bound to the expression and concatenates it to the M automaton. The quantifiers and identifiers of the automaton from the expression are also added to the previous lists.
- Automaton : Concatenates the automaton to M. If it is possible, adds its identifiers to the previous lists, otherwise, it throws an Exception.
- Number Literal : Creates an automaton which accepts the value of the literal and concatenates it to M. Also adds the identifiers and quantifiers of the number literal to the previous lists.

At the end of the loop, it binds the extracted identifiers to W and it adds a new Expression to the stack S.

Put()

Add the token to List postOrder which contains the postfix notation.

Previous : Token.java

7.2 Code quantificateur S

```
else if (op.equals("S")){

String c = "variabletemporaire";

List<String> bla = new ArrayList<>();
bla.add(c);

Automaton Z = M.clone();

Automaton A = Z.NS.get(0).equality.clone();
A.bind(list_of_identifiers_to_quantify.get(0),c);
Z = Z.and(A,print,prefix+" ",log);
Z.quantify(list_of_identifiers_to_quantify.get(0),print,prefix,log);
```

```
Predicate p = new Predicate(list_of_identifiers_to_quantify.get(0)+"<="+c);
RelationalOperator test = (RelationalOperator) p.get_postOrder().get(2);
Stack<Expression> s = new Stack<>();

Variable tmp1 = (Variable) p.get_postOrder().get(0);
Variable tmp2 = (Variable) p.get_postOrder().get(1);
s.push(new Expression(tmp1.name));
s.push(new Expression(tmp2.name));

test.act(s,print,prefix,log);

Automaton B = s.pop().M;

Z = Z.imply(B,print,prefix,log);
Z = Z.and(M,print,prefix,log);

Z.not(print,prefix+" ",log);
Z.quantify(new HashSet<String>(bla),print,prefix+" ",log);
Z.not(print,prefix+" ",log);

M = Z.clone();

}
```