Un éditeur de grammaires attribuées modulaires

Antoine Amarilli

École Normale Supérieure

Sommaire

Mon stage en quelques mots

Lieu IRISA de Rennes, équipe S4, sous la direction d'Éric Badouel.

Thème Les grammaires attribuées, un concept introduit par Knuth et utilisé en compilation.

Objet Une manière d'ajouter de la modularité aux grammaires attribuées.

Travail Le développment d'un prototype d'éditeur permettant de construire graphiquement des grammaires attribuées modulaires, des documents, et des règles sémantiques, qui génère du code Haskell permettant d'évaluer les attributs dans des cas simples.

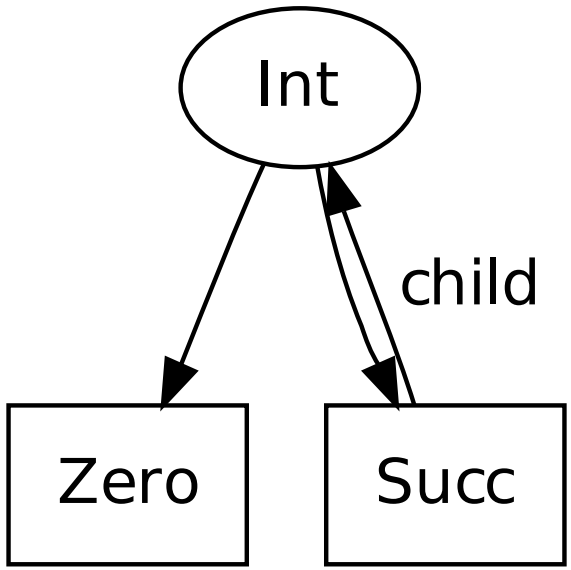
Objectifs Identifier des problèmes dans le formalisme développé par mon maître de stage.

Sommaire

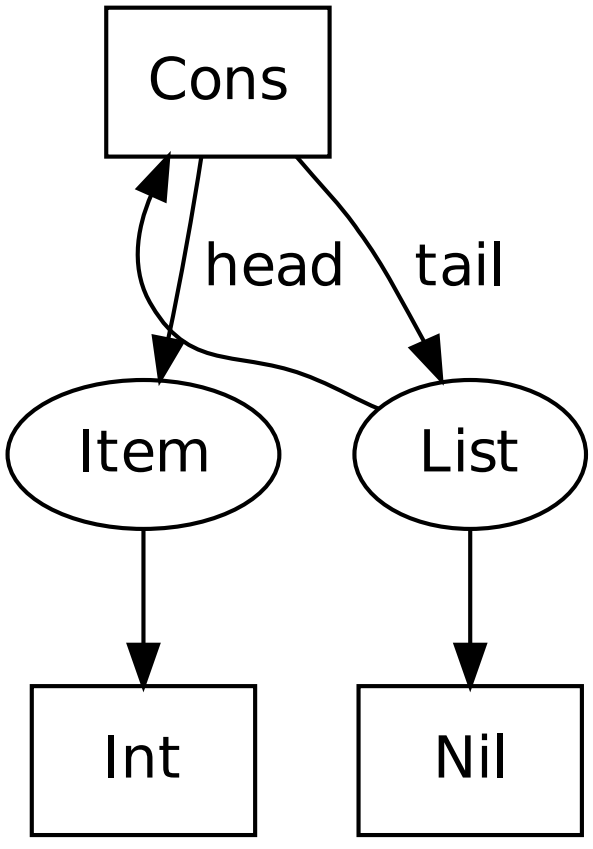
Éléments

- Des sortes (ronds), qui représentent des types.
  - → Produites par un certain nombre de productions.
- Des productions (carrés), qui représentent des constructeurs.
  - → Utilisent un certain nombre de sortes (avec éventuellement des sélecteurs).

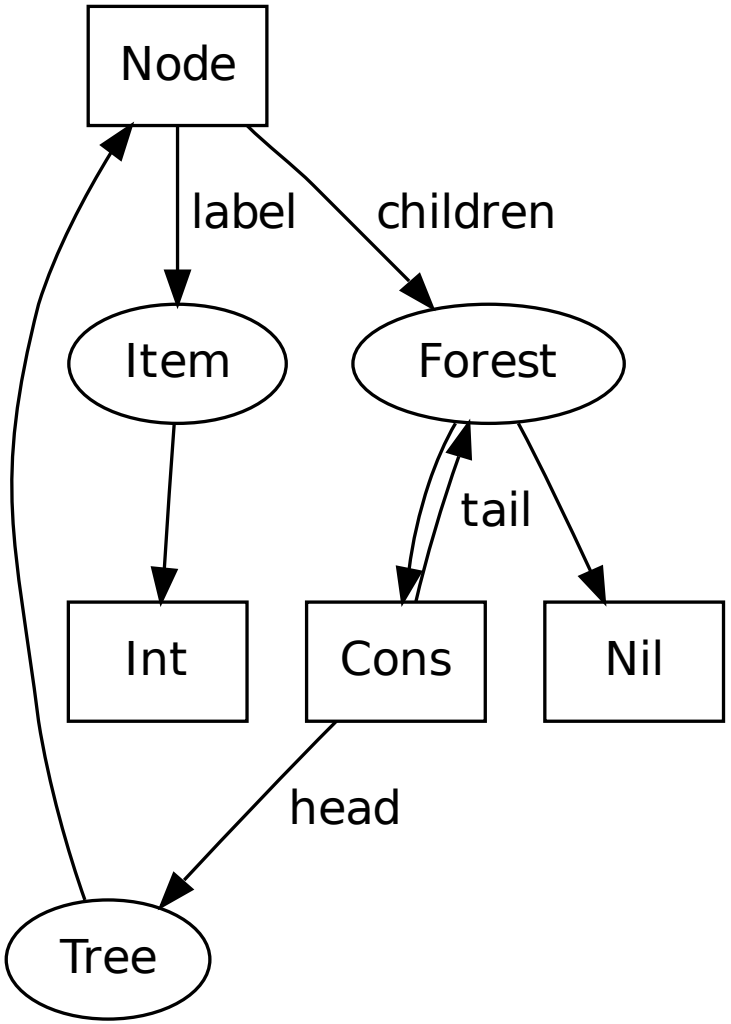
Exemple : représentation simple des entiers



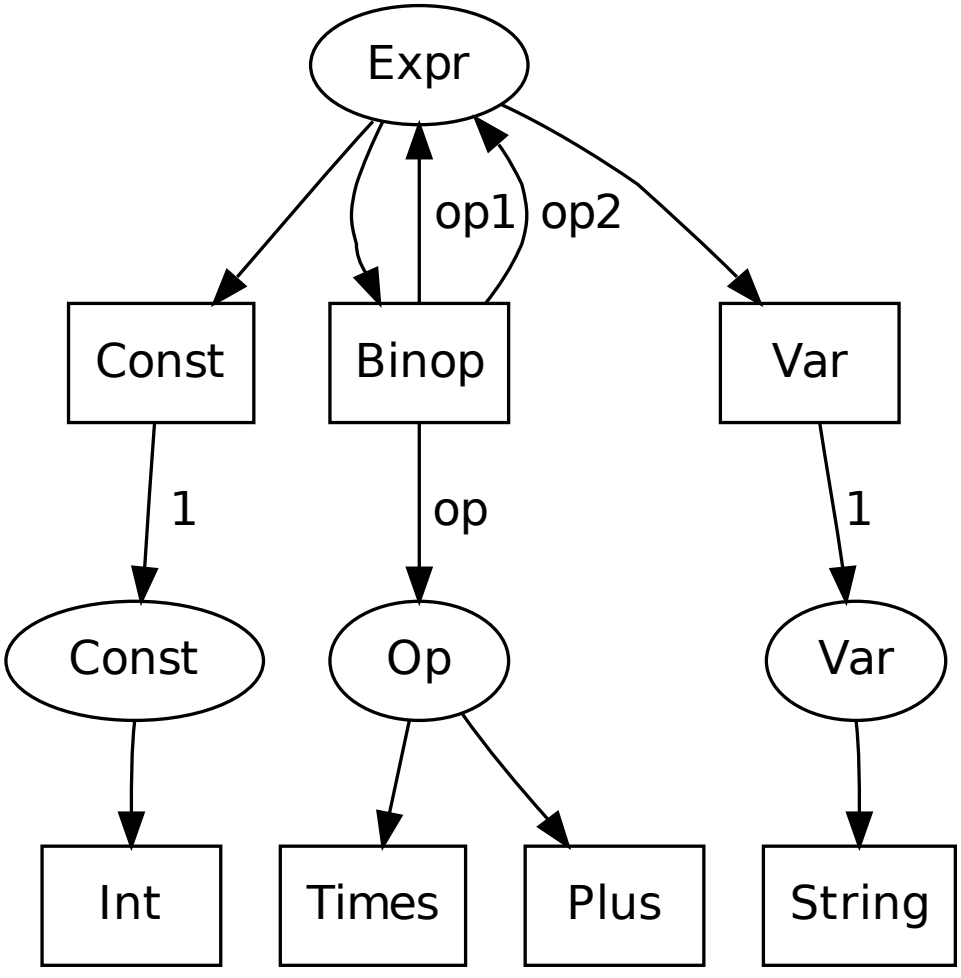
Exemple : liste chaînée d'entiers



Exemple : arbre d'entiers



Exemple : expressions mathématiques

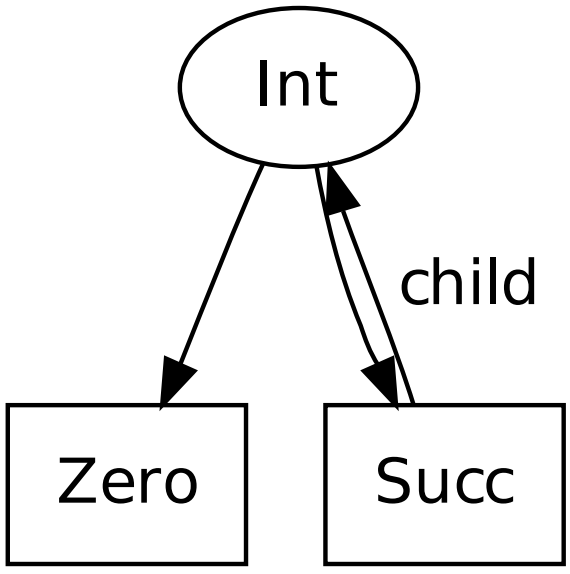


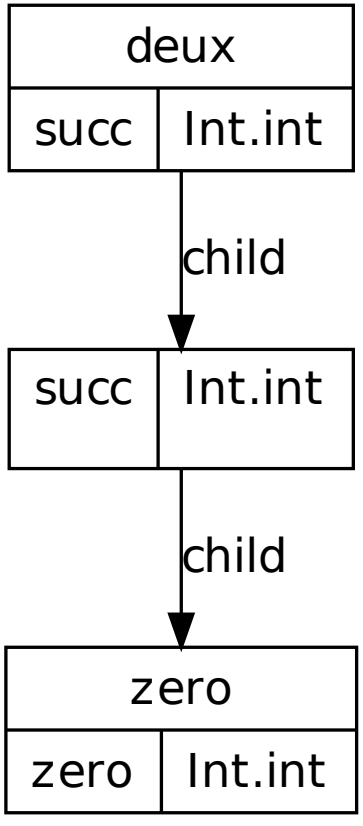
Sommaire

Construction

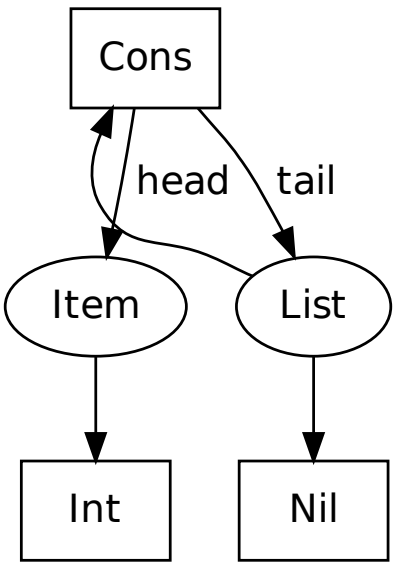
- On part d'une sorte (l'axiome), ce qui crée un noeud.
- On développe ce noeud en choisissant une production.
- Cela crée de nouveaux noeuds ayant la sorte des arguments de cette production.
- On recommence sur ces nouveaux noeuds.

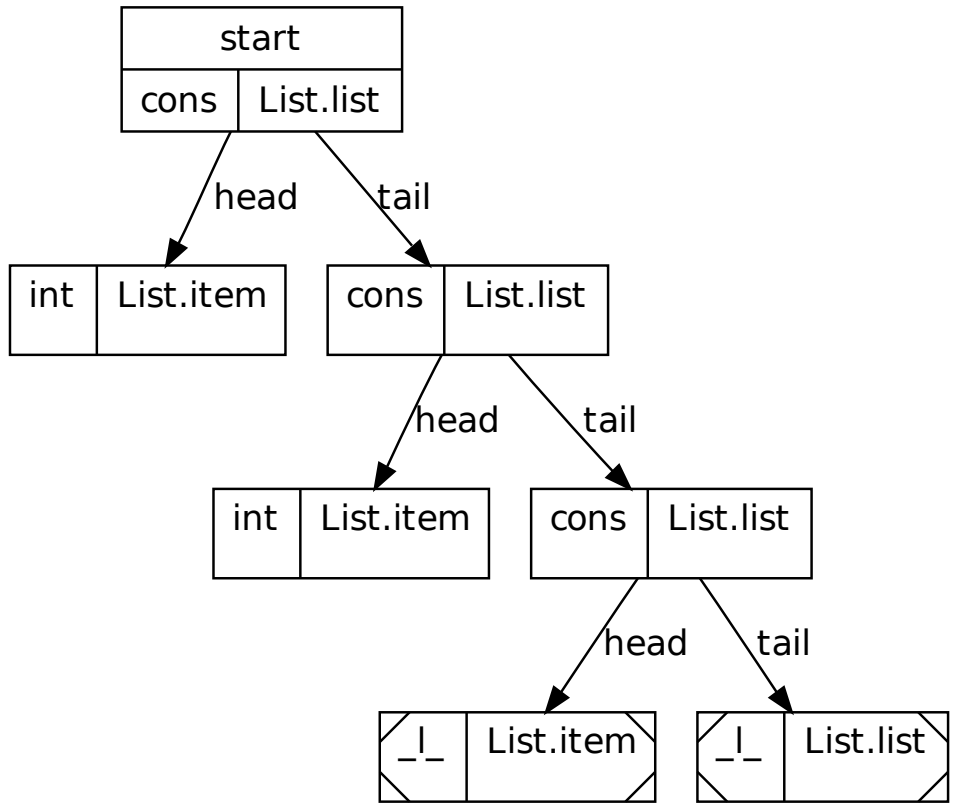
Exemple : représentation simple des entiers



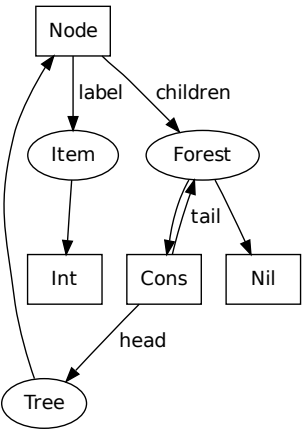


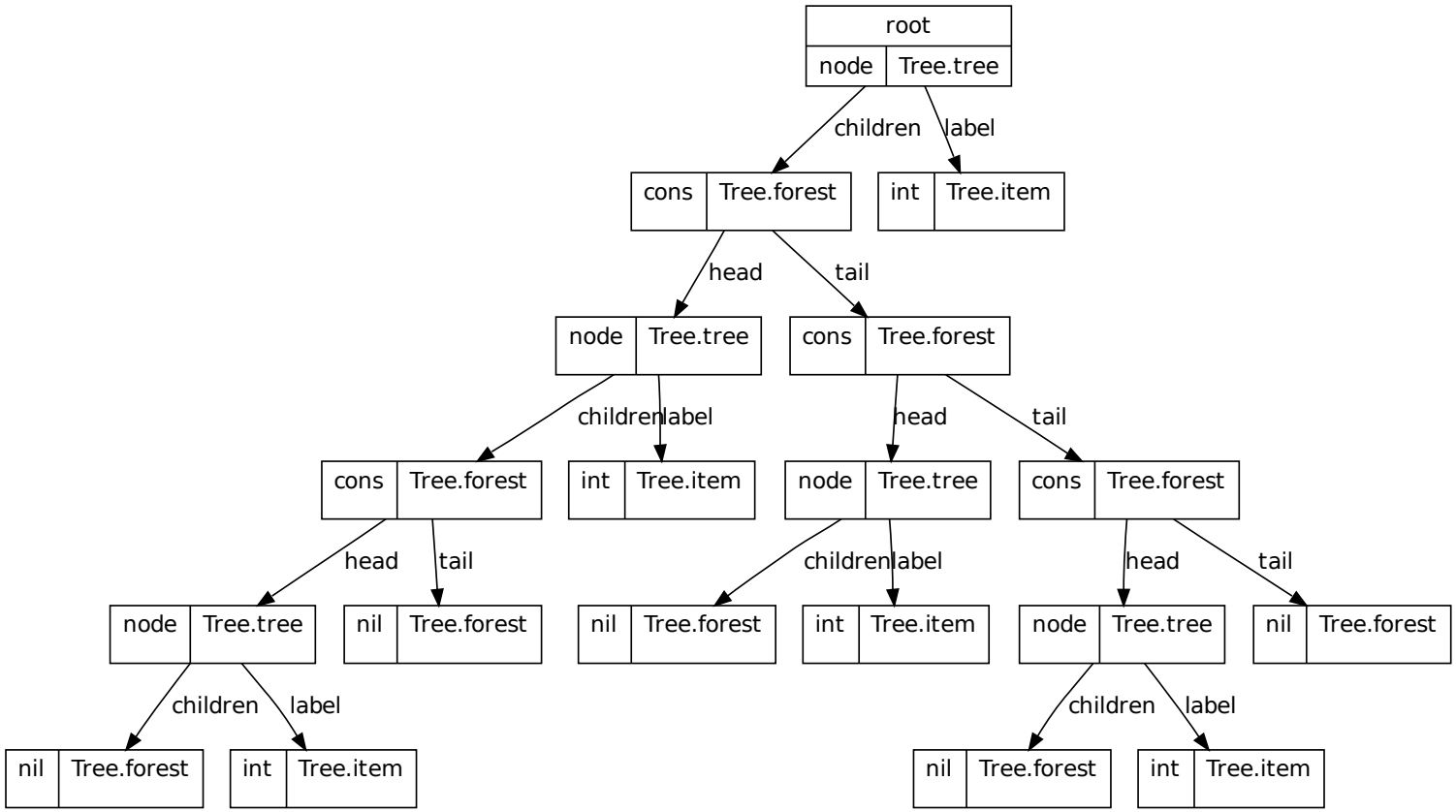
Exemple : liste chaînée d'entiers





Exemple : arbre d'entiers

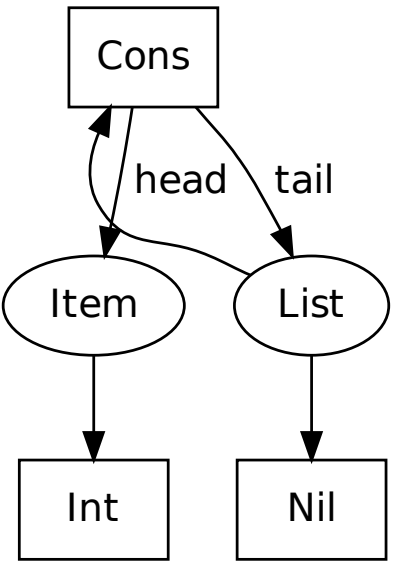


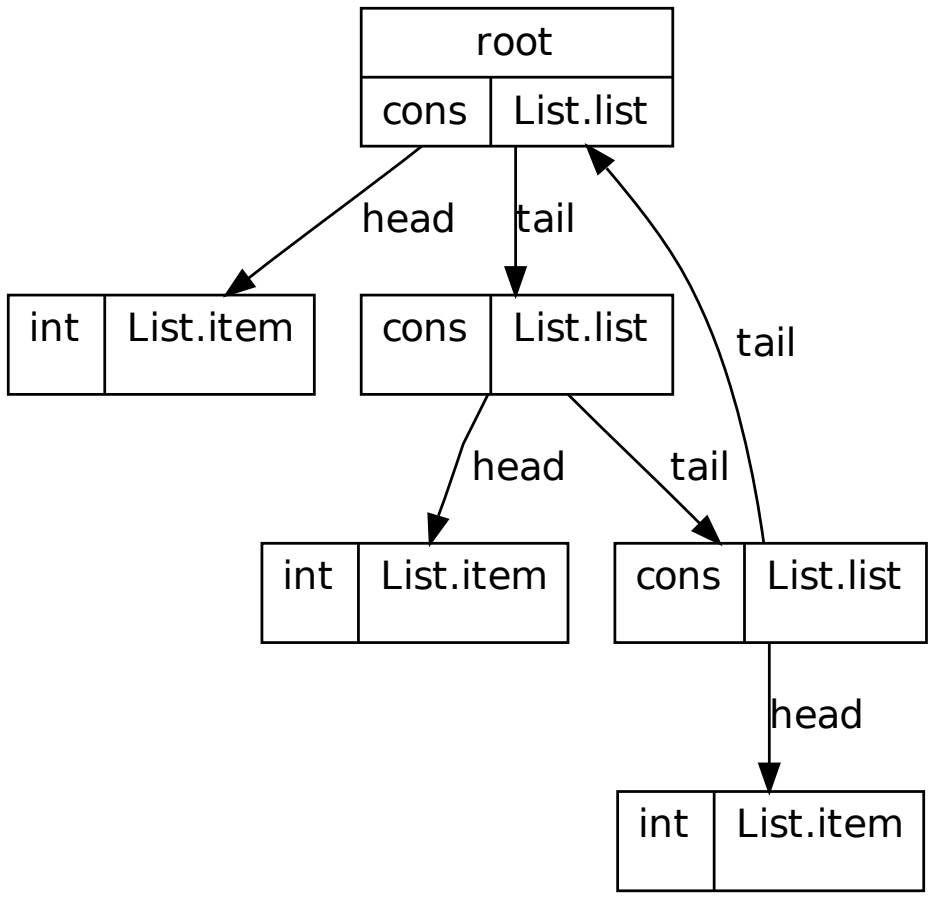


Partage

- Les noeuds du document peuvent être partagés entre plusieurs noeuds parents.
- Il peut donc y avoir des cycles.

Exemple : partage





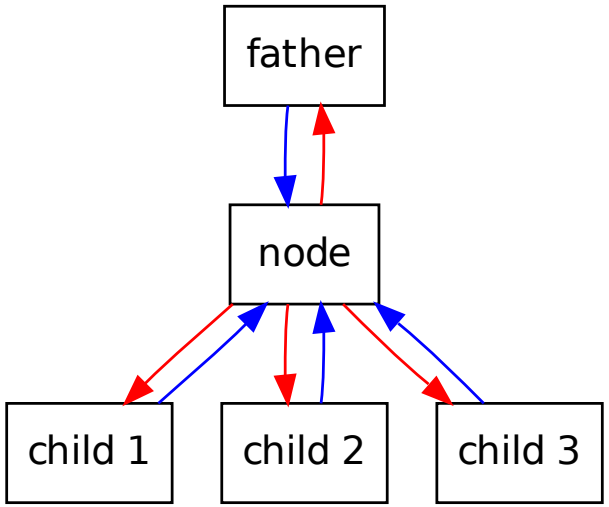
Sommaire

Description générale

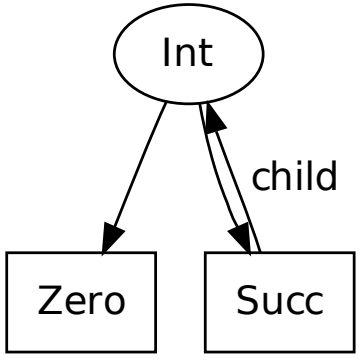
- Les attributs permettent de définir formellement la sémantique d'un document.
- Les sortes sont munies d'attributs.
- Deux cas simples : attributs purement synthétisés (de bas en haut), et purement synthétisés (de haut en bas).
- La version générale permet d'aller et venir dans les deux sens.

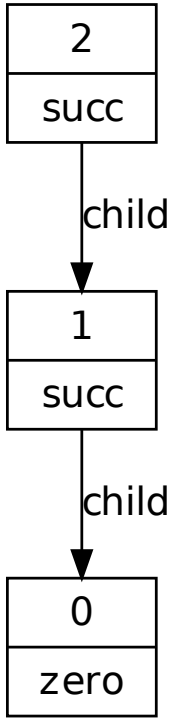
Règles sémantiques

- Les productions sont munies de règles sémantiques permettant de calculer les attributs d'entrée en fonction des attributs de sortie.
- Attributs d'entrée : attributs hérités du père et synthétisés des fils.
- Attributs de sortie : attributs synthétisés du père et hérités des fils.



Exemple : attribut purement synthétisé



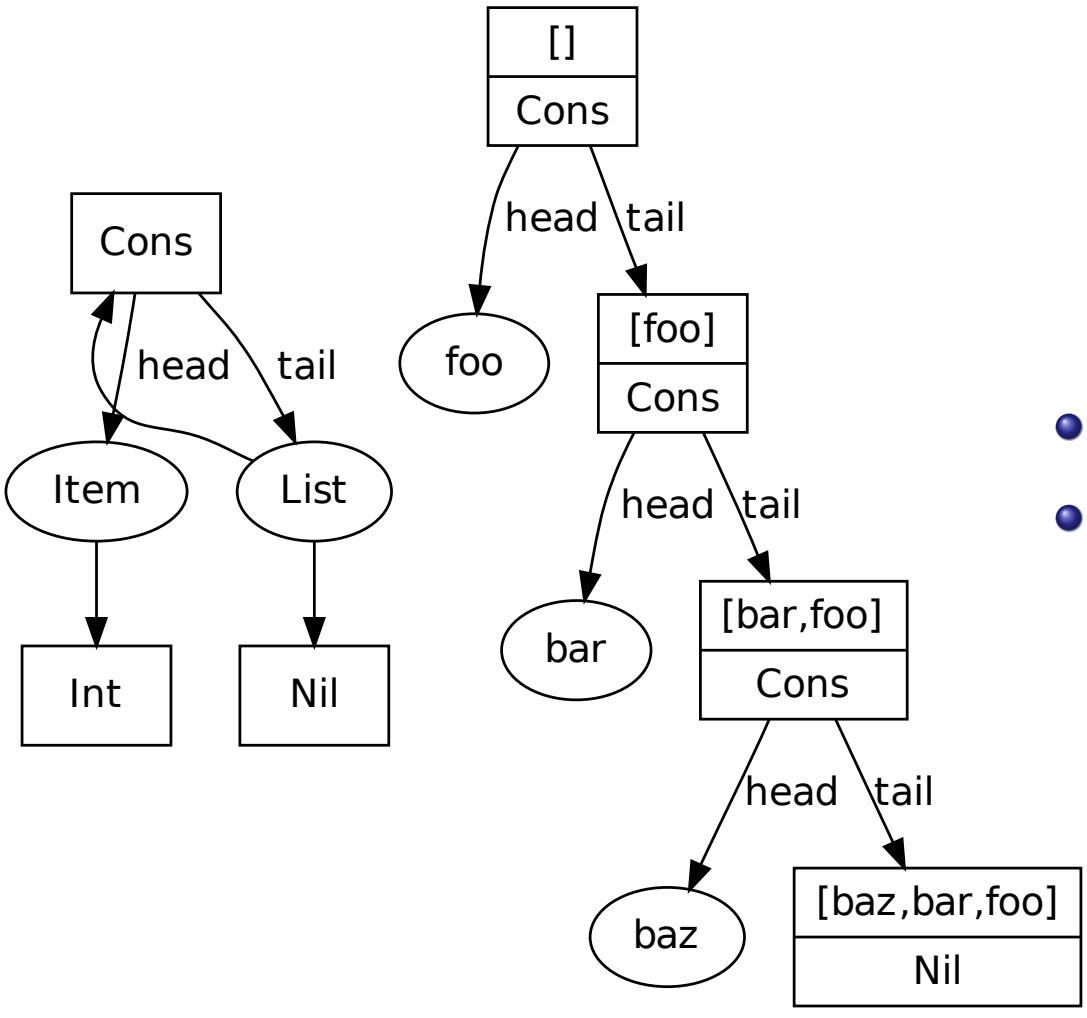


- Zero : $value = 0$.
- Succ :
  $value = 1 + child.value$

Autres exemples

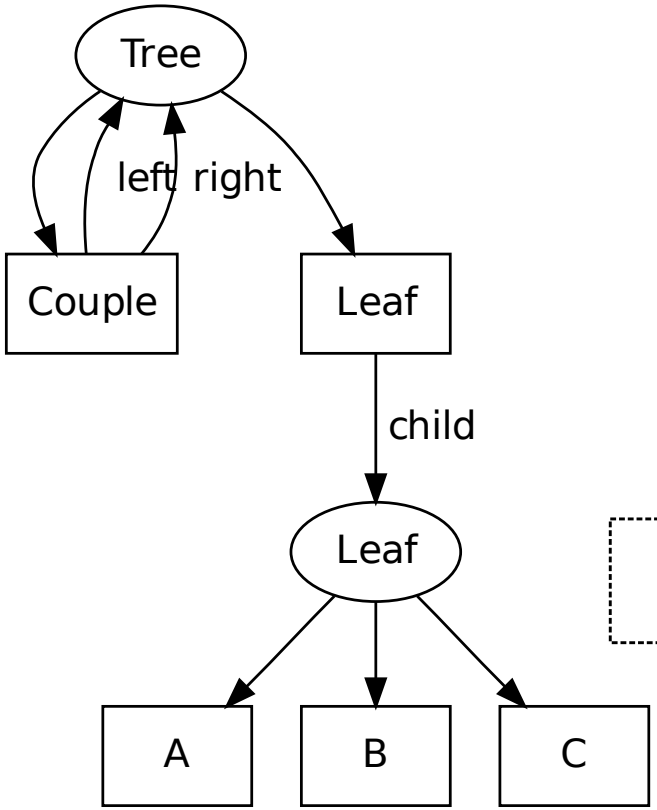
- Longueur d'une liste.
- Nombre d'éléments, profondeur maximale ou minimale d'un arbre.
- Valeur d'une expression mathématique.

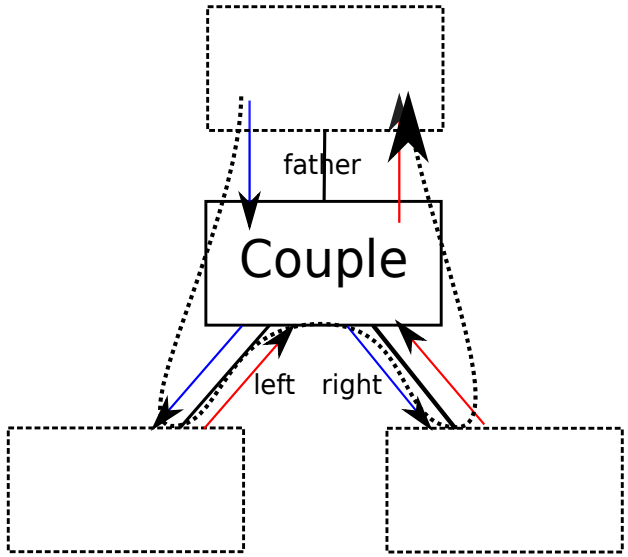
Exemple : attribut purement hérité

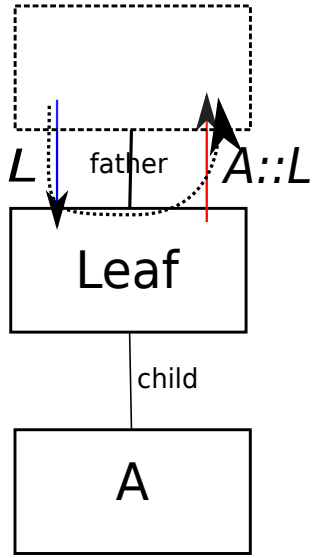


- *root* : *value* = [].
- Cons : *value* = *head* :: *father*.*value*

Exemple plus complexe : la liste des feuilles d'un arbre binaire







Autres exemples

- Valeur d'une expression mathématique avec des `let in`.
- Calcul de la position de blocs imbriqués.

Problèmes

- Partage du nœud du document : on ne peut pas avoir d'attributs hérités dans ce cas.
- Risques de définitions circulaires (même sans partage).

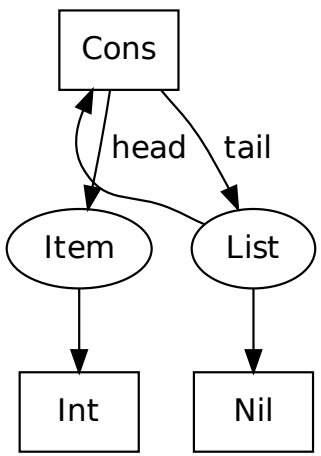
Nature des sémantiques

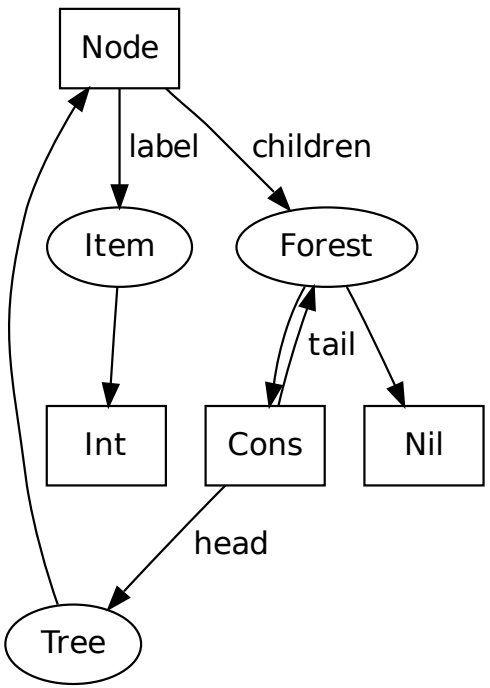
- On convient que les règles sémantiques aboutissent à la construction d'un bout de document où apparaissent les attributs d'entrée.
- C'est relativement naturel pour l'exemple proposé.
- En fait, si l'on suppose que la grammaire dispose des sortes et productions représentant la grammaire Haskell, ce n'est pas vraiment limitant (sauf au niveau des performances).
- La modularité rend une telle contrainte très raisonnable : il suffit d'avoir un module représentant le langage Haskell.

Sommaire

Motivation

Dans les exemples précédents, nous avons vu que certaines structures élémentaires étaient communes à de nombreuses grammaires.





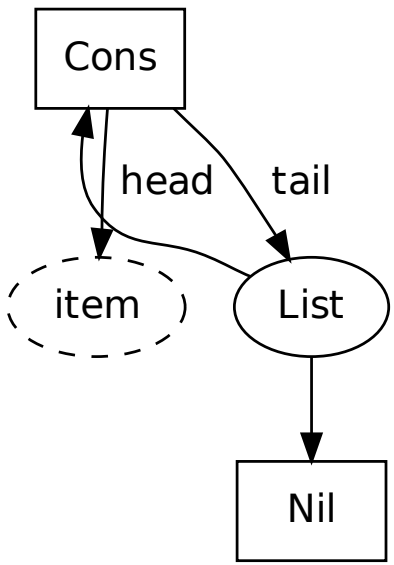
On voudrait trouver un moyen permettant de décrire ces structures une seule fois et les utiliser à plusieurs endroits.

Idée

- On n'a plus une seule grammaire mais un ensemble de modules qui sont autant de grammaires indépendantes.
- La grammaire modulaire est l'ensemble formé par ces modules.

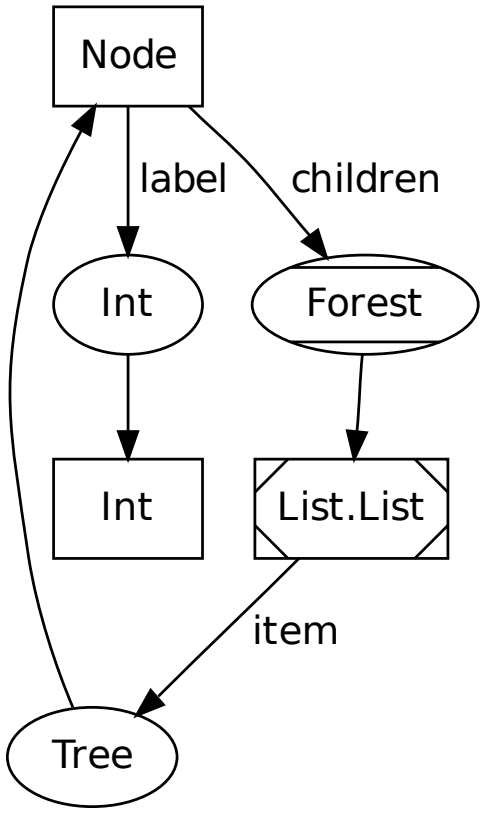
Idée

On ajoute la possibilité d'avoir des *paramètres* où l'on viendra brancher des valeurs.



Idée

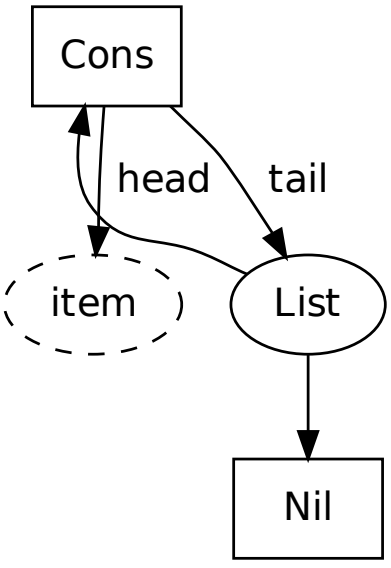
On ajoute ensuite la possibilité d'appeler un module en passant des valeurs pour les paramètres.

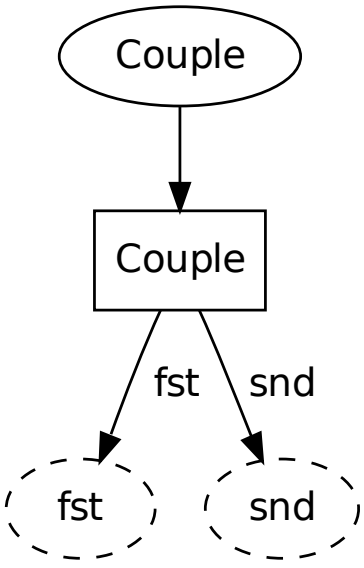


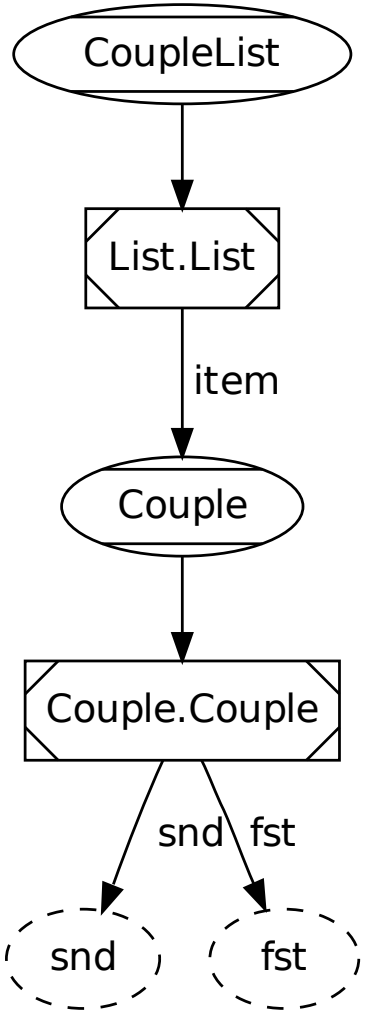
Contraintes

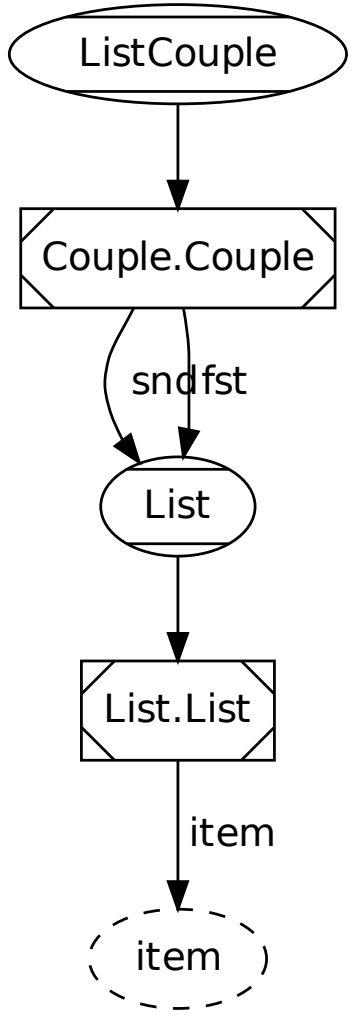
- On impose que le graphe de dépendances entre modules soit acyclique.
- On interdit les cycles de définitions.
- On impose que les fils des indirections correspondent exactement, par leurs sélecteurs, aux paramètres du module appelé.

Exemple : listes et couples

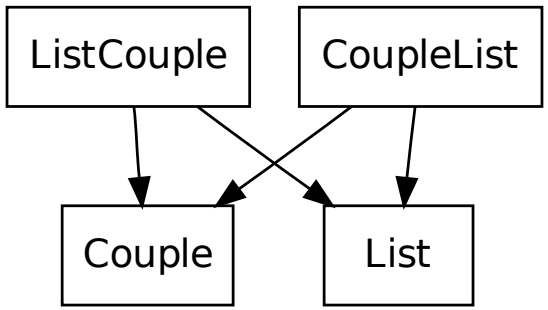








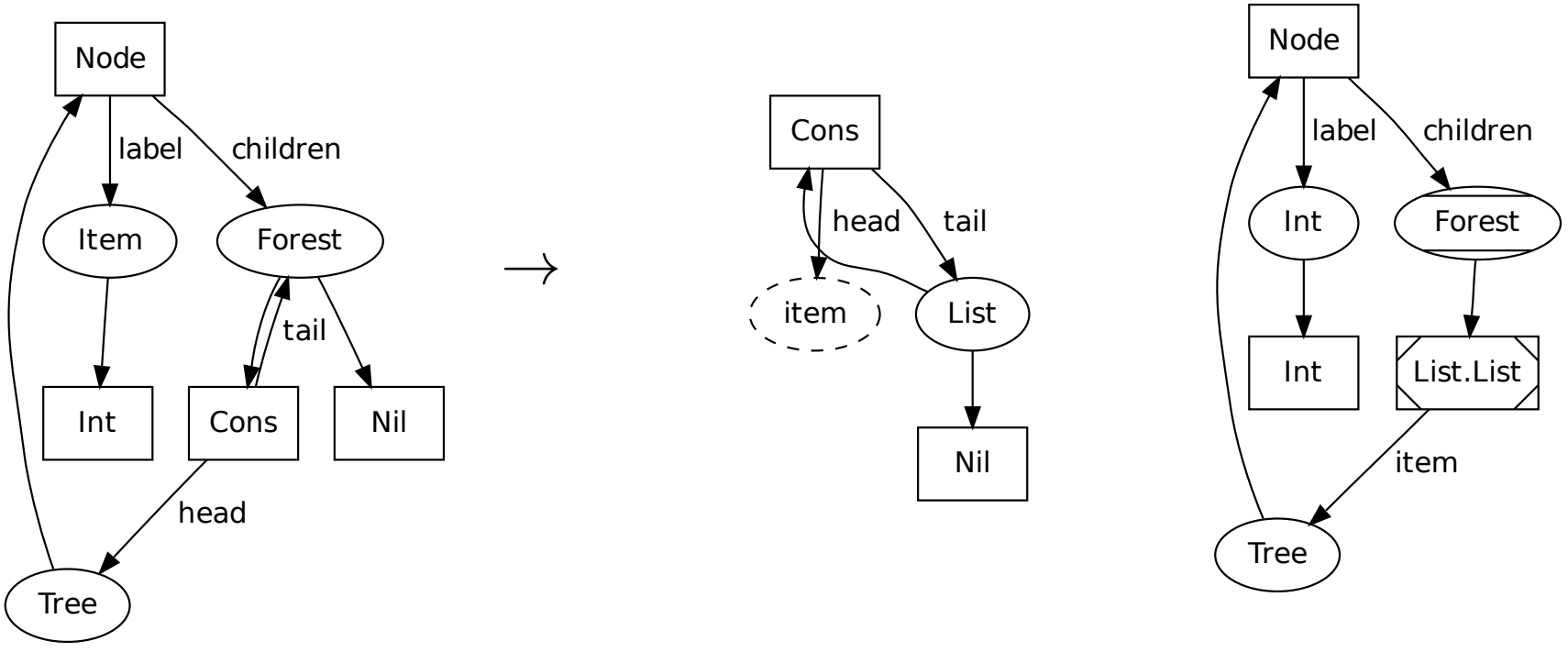
Exemple : listes et couples



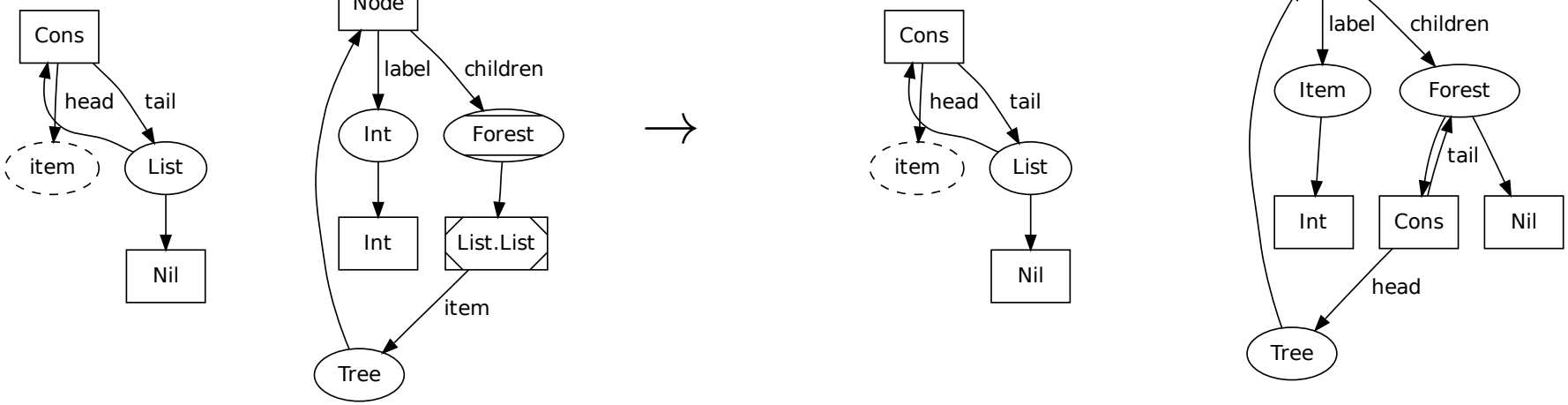
Opérations naturelles

- La *décomposition*, où on prend une partie d'un module et où on la déplace vers un nouveau module.
- La *fusion*, où on résout une indirection en réalisant une copie en dur du contenu du module vers lequel l'indirection pointe.

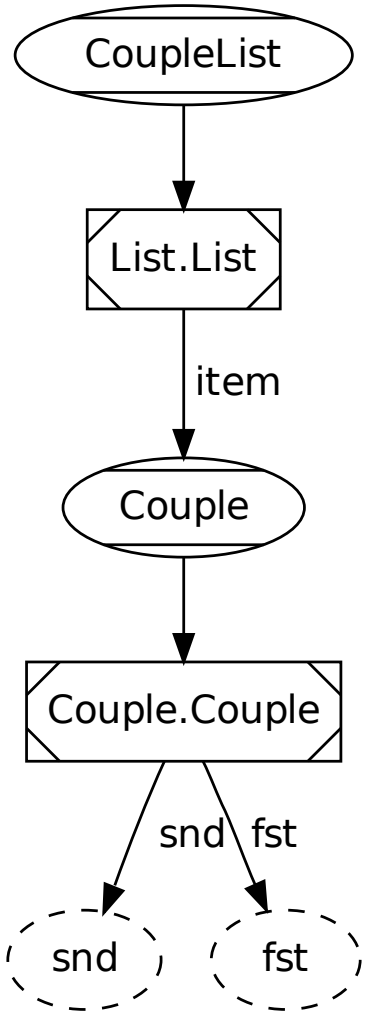
Décomposition

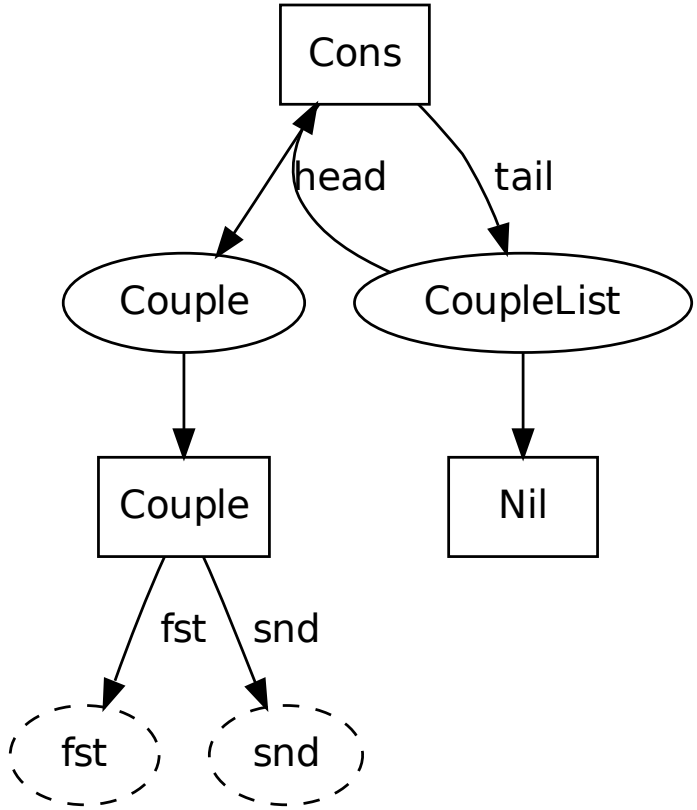


Fusion

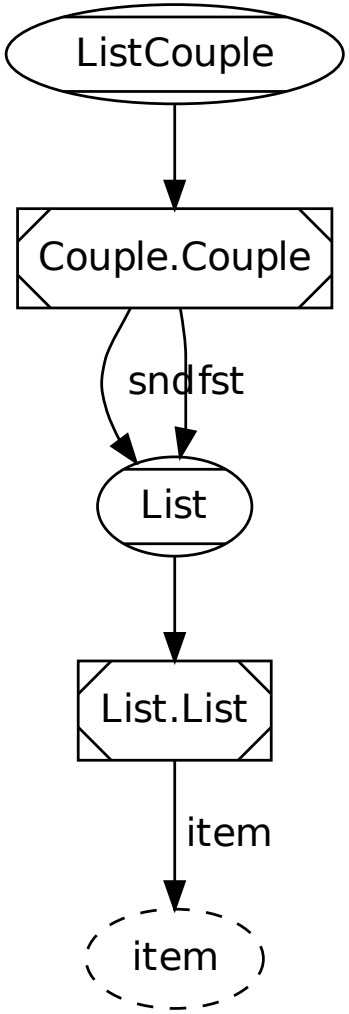

Cons
head
tail
item
List
Nil
Node
label
children
Int
Forest
Int
List.List
item
Tree
Cons
head
tail
item
List
Nil
Node
label
children
Item
Forest
tail
Int
Cons
Nil
head
Tree

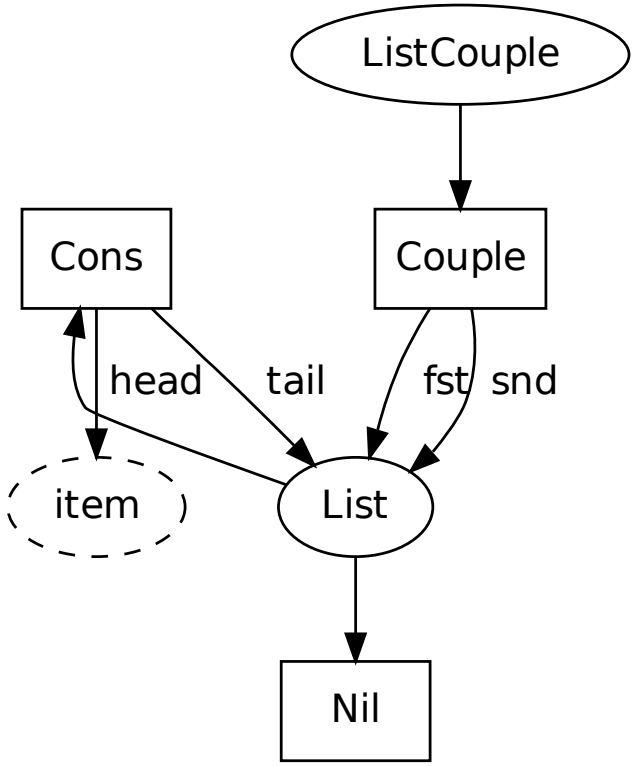
Fusion (CoupleList)





Fusion (ListCouple)



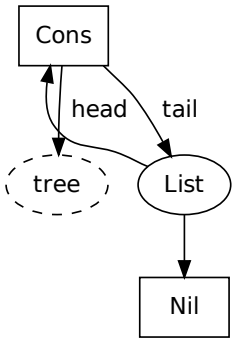


Modularité et documents

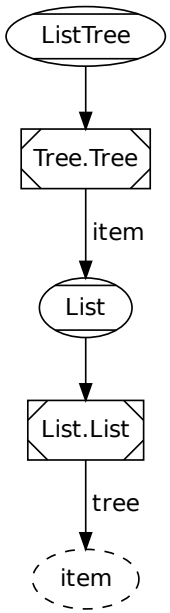
- Lors de la construction du document, on passe de façon transparente d'un module à l'autre.
- On peut masquer les noms des modules appelés lorsqu'on vient d'un module de plus haut niveau, ce qui rend la modularité totalement invisible dans de nombreux cas.

Exemple de document modulaire

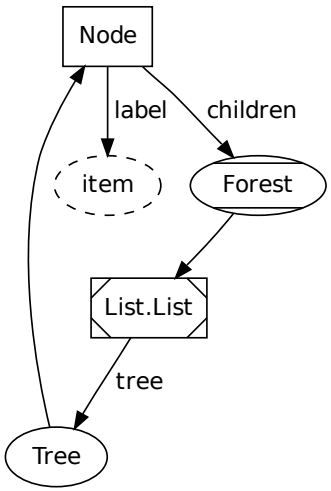
List

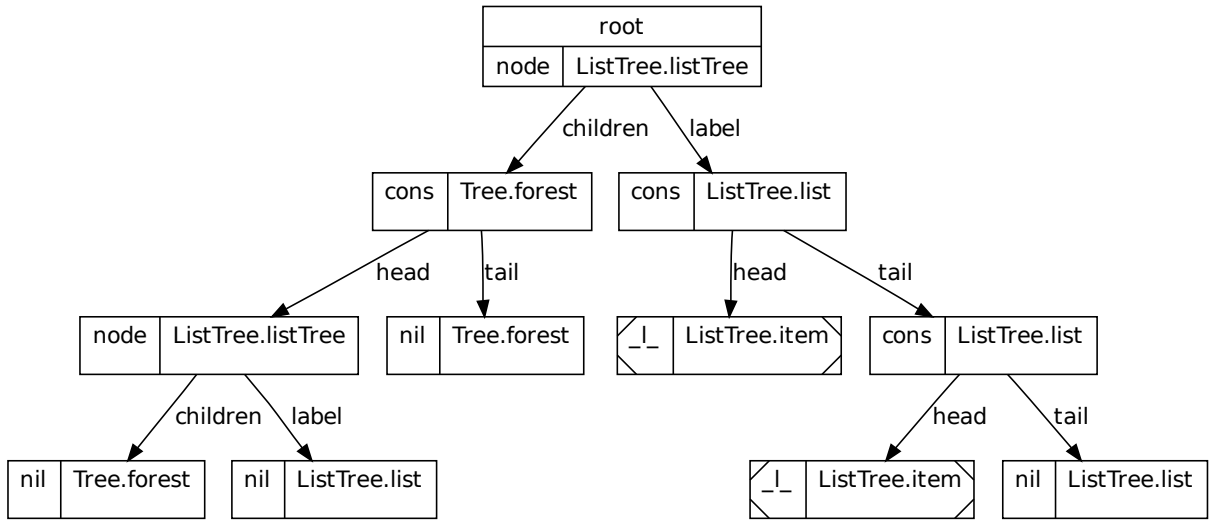


ListTree



Tree





Sommaire

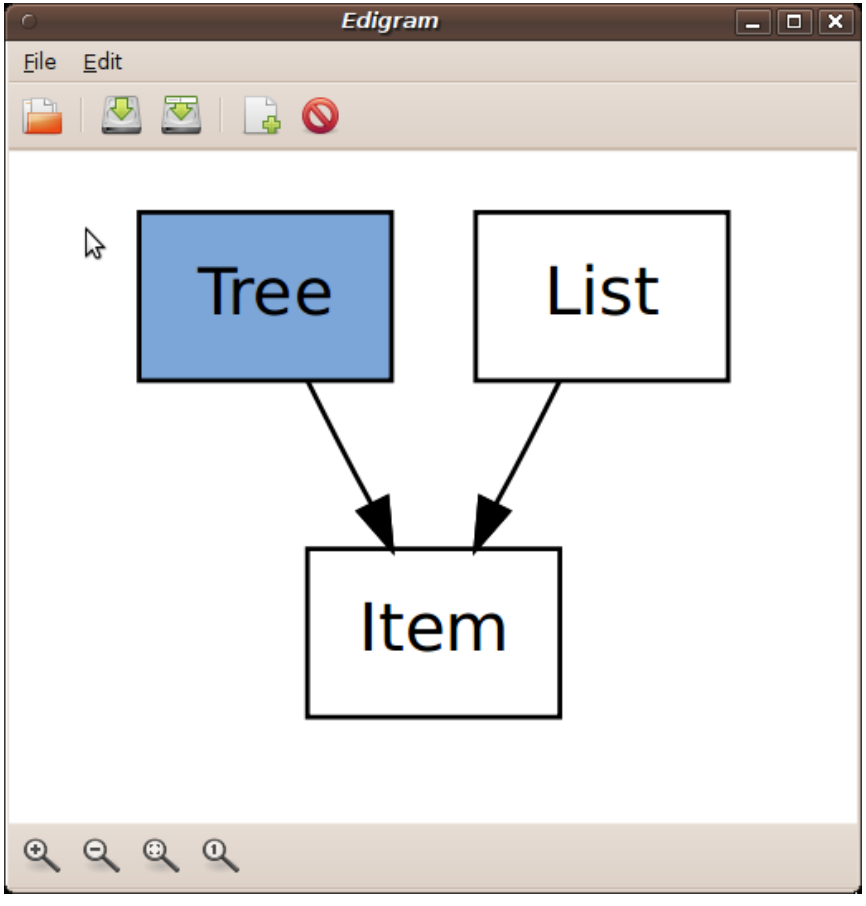
Présentation générale

- On veut pouvoir implémenter en Haskell les grammaires attribuées modulaires et le calcul d'attributs.
- On veut pouvoir construire grammaires, documents et sémantique de façon agréable et intuitive.
- On veut donc un logiciel pour construire ces objets avec une interface graphique et produire du code Haskell en prenant en charge la génération du code répétitif.
- Il s'agit surtout d'un prototype pour identifier les problèmes d'implémentation et faire des tests.

Informations techniques

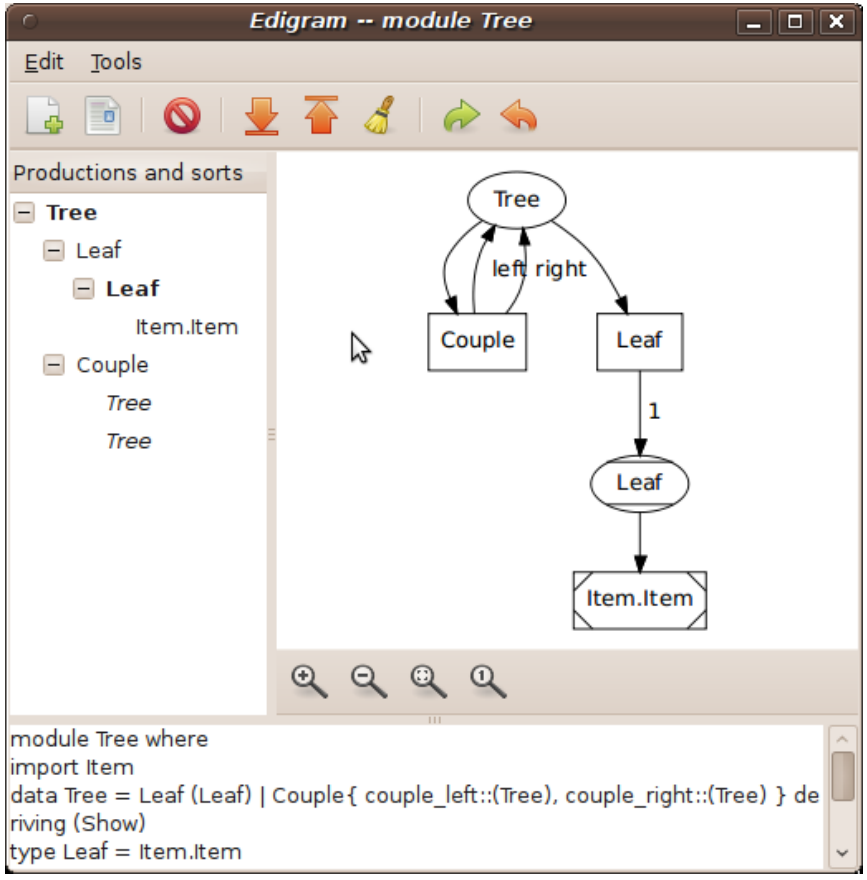
- Langage Python
- Programmation orientée objet
- PyGTK pour l'interface graphique (Python-GTK)
- Xdot pour le dessin de graphes (Python-Graphviz)
- Ply pour parser le code Haskell (Python-Lex-Yacc)
- Environ 8000 lignes de code.

Édition de grammaires



- Création, édition de modules
- Calcul automatique des dépendances

Édition de modules

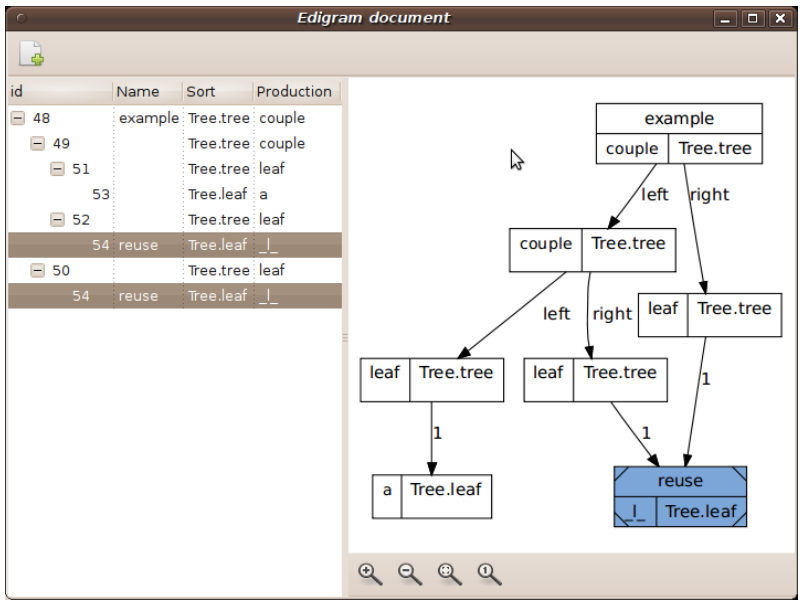


- Création, édition de sortes, de productions
- Décomposition et fusion

Instances d'une sorte

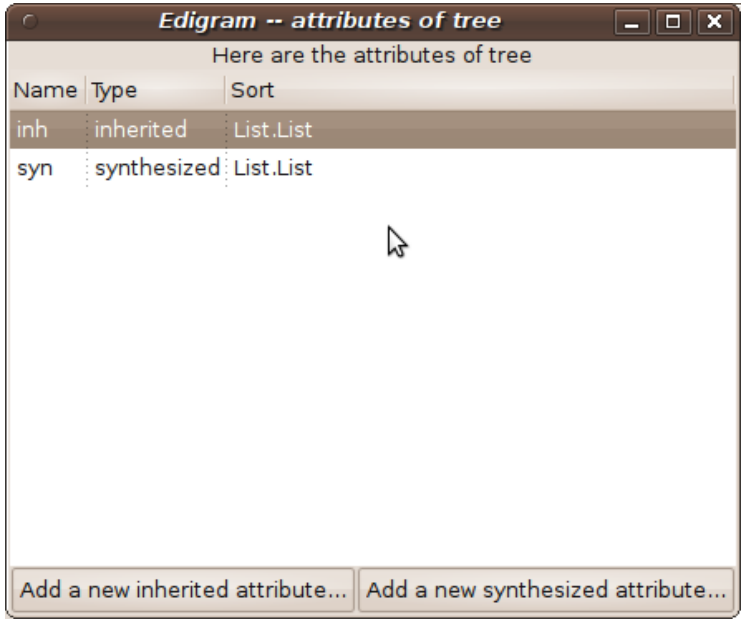
- Liste des instances d'une sorte
- Fonctionne également pour les définitions

Édition du document



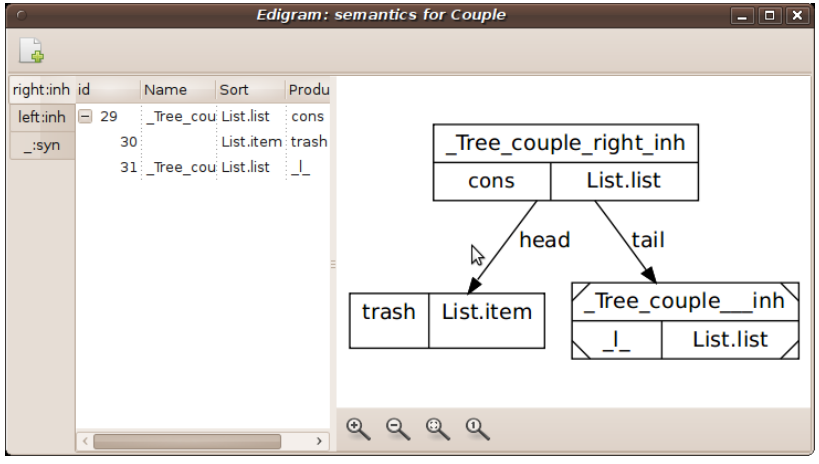
- Nouveaux nœuds
- Construction du document
- Partage de nœuds
- Copie de structures
- Nommage de nœuds
- Déconstruction du document

Attributs



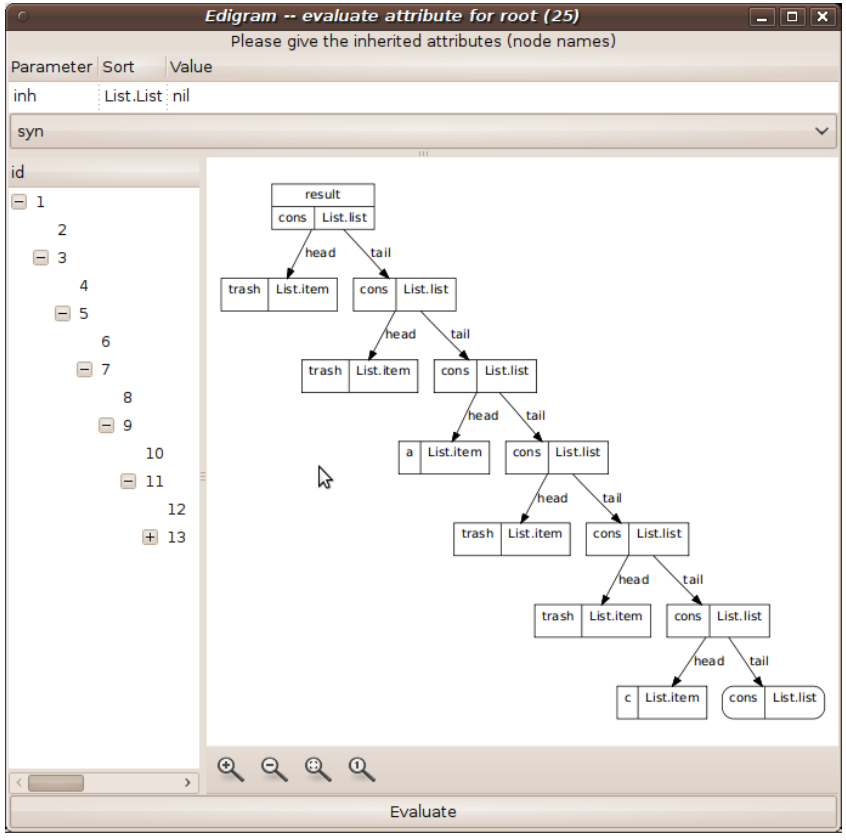
- Liste des attributs d'une sorte
- Création de nouveaux attributs

Construction de règles sémantiques



- Création des règles sémantiques
- (comme des fragments de document)

Construction de règles sémantiques



- Évaluation d'attributs sur des nœuds
- (par la génération de code Haskell)

Fichiers générés

- Déclaration de modules, de types ("data") et abréviations de types ("type") pour la grammaire.
- Déclarations Haskell pour les documents et la sémantique.
- Déclarations de modules avec du code Haskell supplémentaire pour le calcul d'attributs. (Ce code peut être réutilisé par l'utilisateur.)

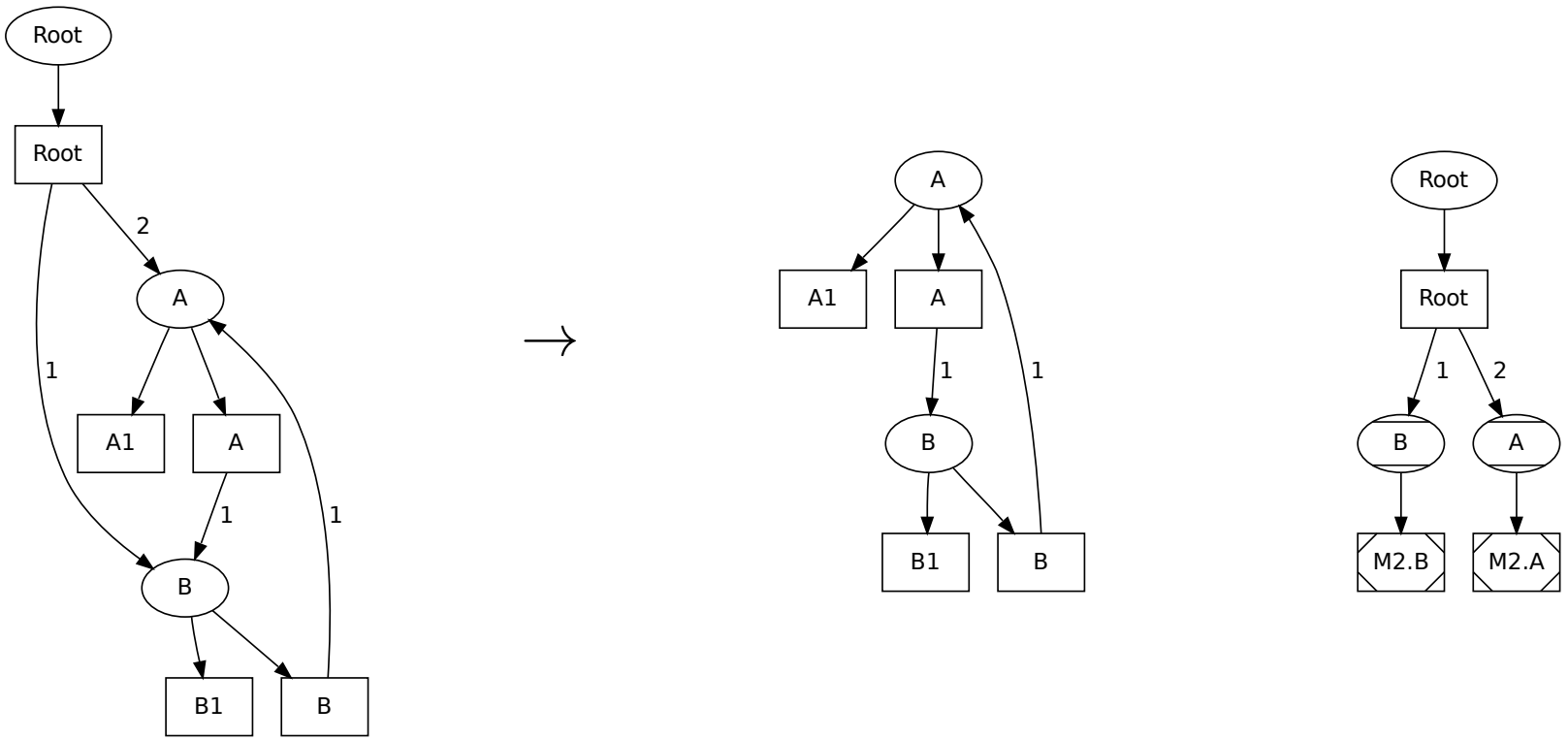
Sommaire

Environnements

- Pour la construction de documents sur des structures modulaires, il faut qu'au niveau de chaque nœud, on stocke les valeurs passées en amont pour les paramètres du module.
- En fait, il suffit de stocker l'historique des indirections par lesquelles on est passé.
- Ces informations doivent être mises à jour lors de la fusion et de la décomposition.

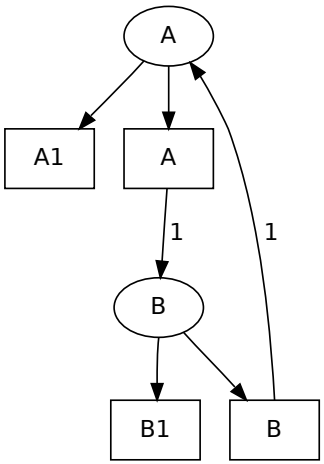
Décomposition et fusion

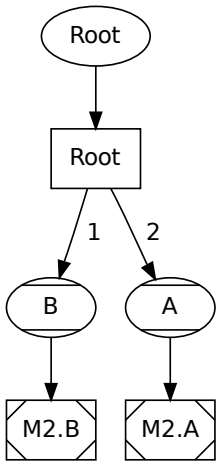
Décomposons...

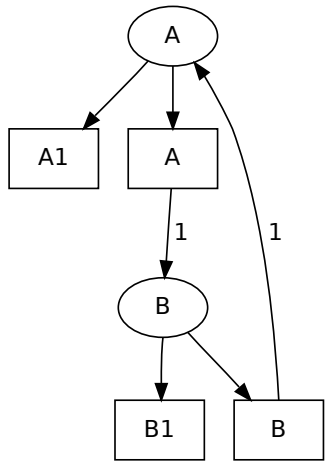


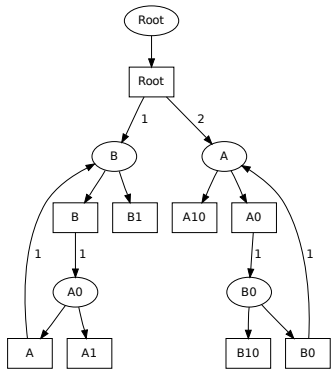
Décomposition et fusion

... et fusionnons...







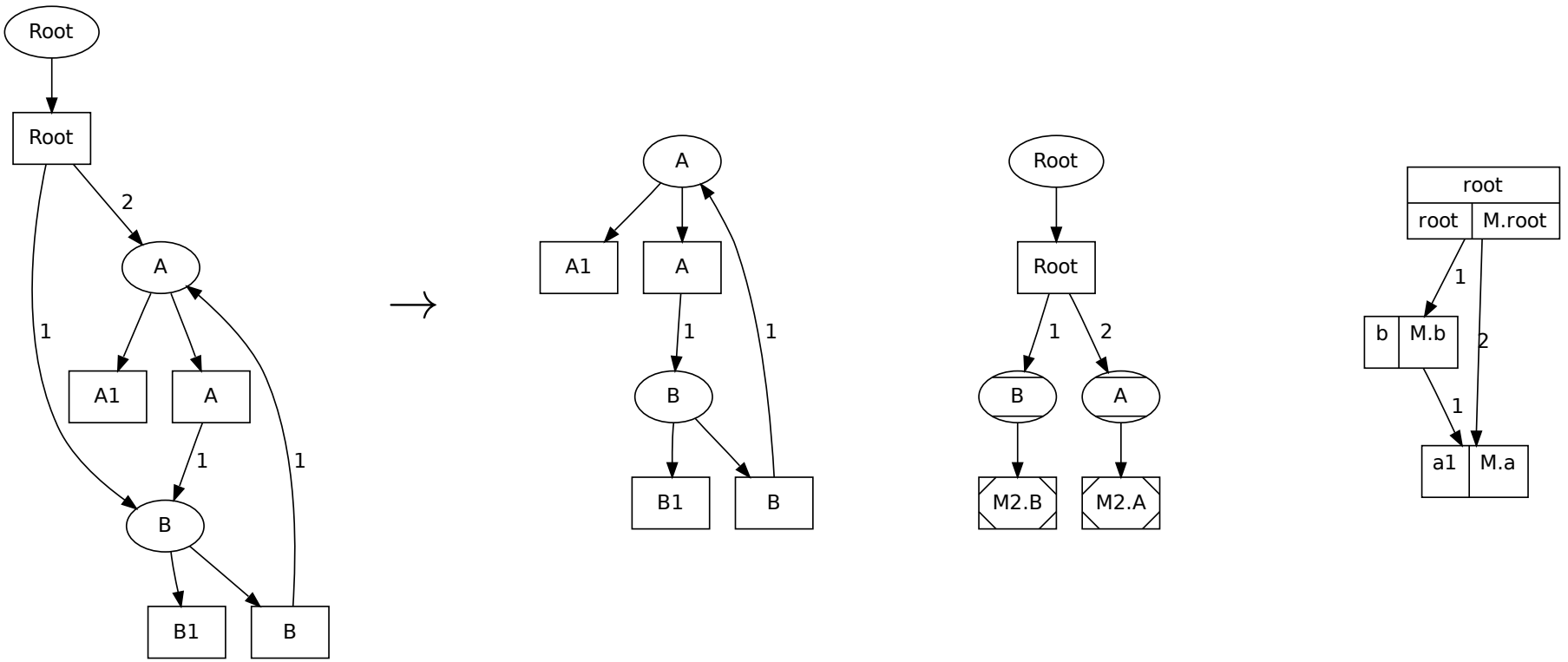


Décomposition et fusion

- On a un problème : la fusion n'est pas vraiment l'opération inverse de la décomposition.
- En fait, le problème est plus subtil...

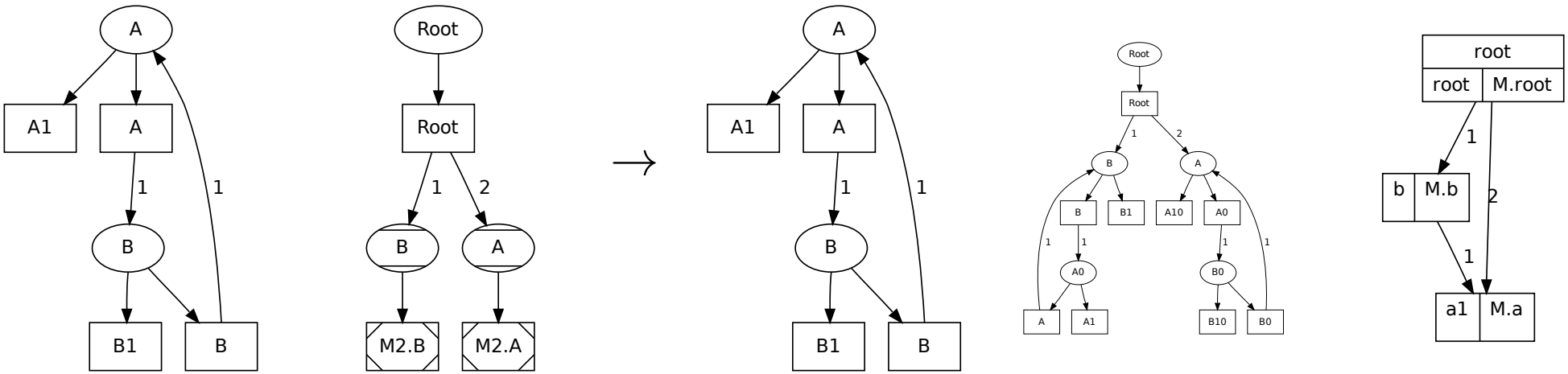
Décomposition et fusion

Décomposons...



Décomposition et fusion

... et fusionnons...



Il y a clairement un problème.

Déconstruction d'un document

- On aimerait laisser à l'utilisateur la possibilité d'annuler un choix effectué lors de la construction du document.
- Cela consiste à ramener un nœud au statut de bourgeon, en le déconnectant de sa descendance.
- Problème : les paramètres passés à la descendance n'existent plus, ce qui invalide une partie de cette descendance.
- La possibilité d'avoir du partage de nœuds et des cycles complique encore les choses.

Sommaire

Améliorations possibles

- Gestion des attributs et de la modularité.
- Module magique avec les mots-clés Haskell.
- Changements de structures de données (l'environnement...)
- Optimisation minimale.
- Correction de bugs, interface graphique...

Perspectives

- Le développement du prototype est poursuivi par Maurice Tchoupé Tchendji, Bernard Fotsing Talla, et Rodrigue Djeumen.
- Le protoype devrait être publié avec l'article présentant le concept de modularité développé par mon maître de stage.

Merci !

Merci pour votre attention !