Compilation

Vérification des types

Jules Chouquet

SOM2IF15 – 2024

Où on en est :

D'où l'on part : Un arbre de syntaxe abstrait généré à travers une hiérarchie de classes que l'on a élaborée à l'aide de visiteurs.
Où l'on va : toujours vers de l'assembleur, à terme. Mais il faut vérifier que le code a **du sens** avant d'essayer de le traduire en exécutable.

Objet de ce cours

Visiter l'AST pour vérifier que les déclarations et utilisations de variables sont cohérentes dans le code, et que toutes les expressions sont bien typées.

Les vérifications correspondent aux dérivations de typage vues au cours précédent, mais avec une gestion des déclarations locales (dans des blocs).

Dans les cours précédents :

La gestion des environnements de typage se faisait à travers une unique liste, ou `map`.
<u>mais :</u> par exemple, le code suivant :

```
if(b){int x;...}else{boolean x;...}
```

est valide dans la plupart des langages.
Mais on a dit au cours précédent qu'un élément de programme ne pouvait être associé qu'à un seul type.
Par ailleurs, dans l'exemple ci-dessus, une utilisation de `x` à la sortie de la conditionnelle n'est pas possible sans erreur.
La variable n'existe que dans le bloc où elle est définie [1]. Elle n'est plus visible après.

1. Donc dans l'exemple ci-dessus, le compilateur se comportera de la même façon que si un nom de variable différent était utilisé.

Mais dis-moi, Jamy, c'est quoi un bloc ?

Mais dis-moi, Jamy, c'est quoi un bloc ?

Une suite d'instructions que l'on souhaite délimiter

Portée des variables

Une déclaration de variable n'est valable que dans le bloc où elle se trouve. Mais, bien entendu, aussi dans les sous-blocs !

```
{...int x;... if(b){x=1;...}else{x=2;...}...print(x);...}
```

Pour définir la portée d'une variable, il faut représenter l'entrelacement des blocs. Si le programme est syntaxiquement correct (bien parenthésé), les blocs forment une arborescence. [2]

2. On peut avoir un nombre arbitraire de blocs dans le même bloc, et un nombre arbitraire d'imbrication de blocs.

Où sont les blocs ?

Au niveau syntaxique : généralement entre les accolades. (mais pas systématiquement). Le corps d'une définition de fonction, d'une boucle while, ou les branches d'une conditionnelle sont toujours des blocs.

La classe `Block` de l'AST

Dans la grammaire, il n'y a pas forcément de règle pour les blocs.[a] On doit rajouter cette catégorie dans l'AST, et transformer toutes les suites d'instructions nécessaires en blocs[b].

---

a. Pourquoi ?

b. Pour cela, deux possibilités : transformer l'AST, ou modifier la première création de l'AST depuis le parseur (`AstBuild`)

Attention, il faut aussi transformer les instructions simples en blocs :

```
if(b) print(b); else print(not(b));
```

Rappel

L'information dont on a besoin : le type des variables apparaissant dans le bloc (comme sous-instructions et sous-expressions).

- Cette information est renseignée au moment de la déclaration de la variable.
- Cette information est utilisée au moment de la visite du bloc qui vérifie que le programme est bien typé.

Cette information, en java, sera représentée par une `Map<String,Type>`.

Si, dans un bloc, on trouve la suite d'instructions `int x;...;y=(x+1)`, alors c'est facile : au moment de vérifier le type de `x+1`, on utilise le `get(“x”)` de notre table, qui aura été enrichie par un `put(“x”,Int)`. Mais si la déclaration de `x` n'est pas dans le même bloc (que son utilisation), alors on a deux possibilités :

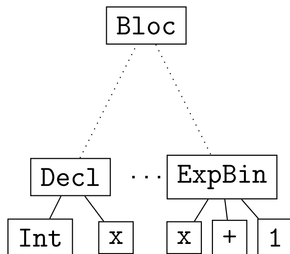
- `x` a été déclarée dans un bloc parent, et il faut donc considérer cette déclaration pour la vérification de types.
- `x` n'a pas été déclarée du tout, et il faut **indiquer une erreur** .

On va commenter la façon de gérer ces deux cas dans la suite du cours.

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

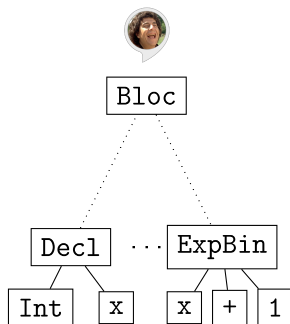


NOM	TYPE
⋮	⋮

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

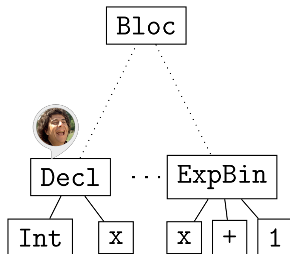


NOM	TYPE
⋮	⋮

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :



NOM	TYPE
⋮	⋮

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

!!! erreur si déjà présent !!!

Bloc

Decl · · · ExpBin

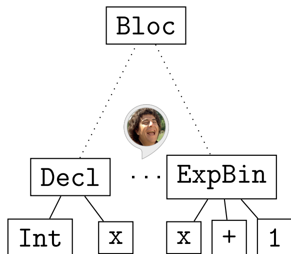
Int x x + 1

NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

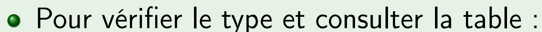
- Pour remplir la table :
- Pour vérifier le type et consulter la table :

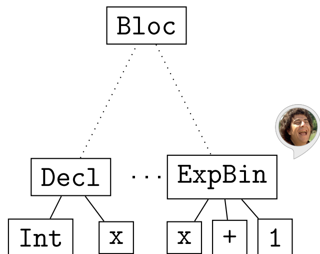


NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table : 

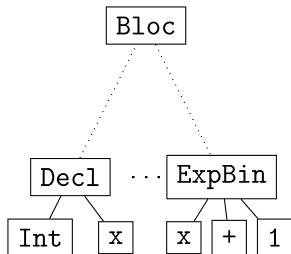


NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

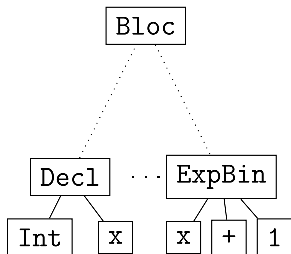


NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

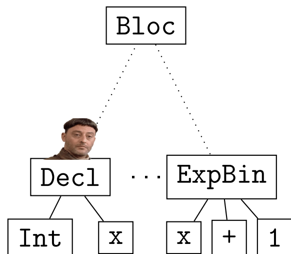


NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

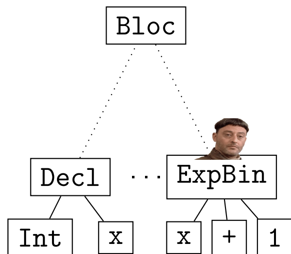


NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

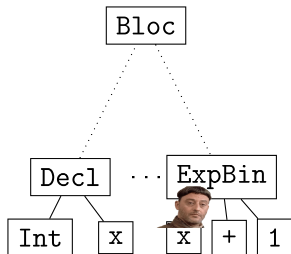


NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

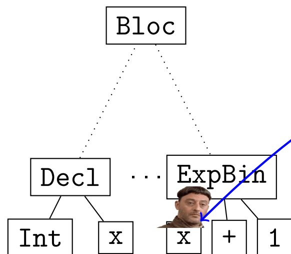


NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

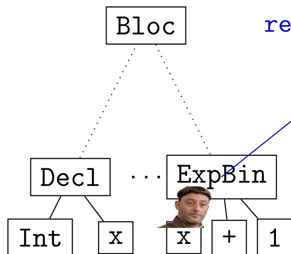


NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :



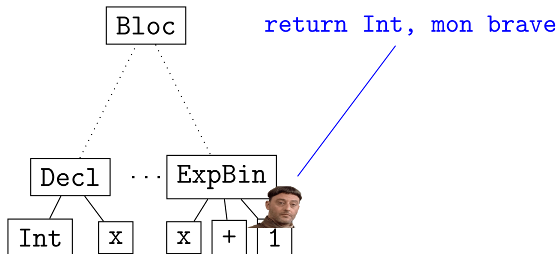
`return Int, mon brave`

NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

- Pour remplir la table :
- Pour vérifier le type et consulter la table :

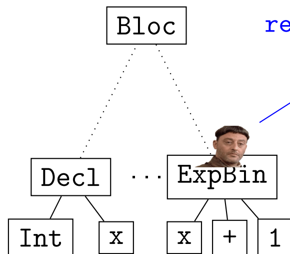


NOM	TYPE
⋮	⋮
x	Int

Dans un seul bloc :

Deux visiteurs vont passer

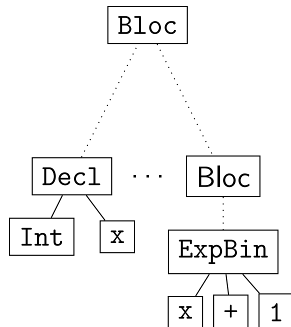
- Pour remplir la table :
- Pour vérifier le type et consulter la table :



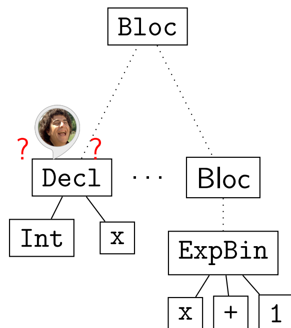
return Int, mon brave

NOM	TYPE
⋮	⋮
x	Int

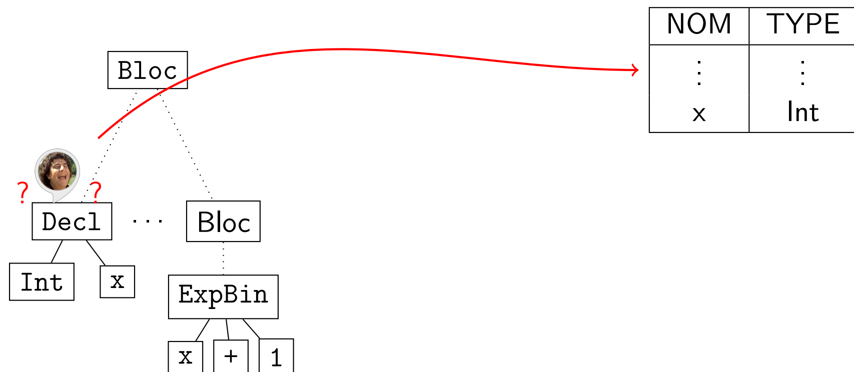
Dans plusieurs blocs ?



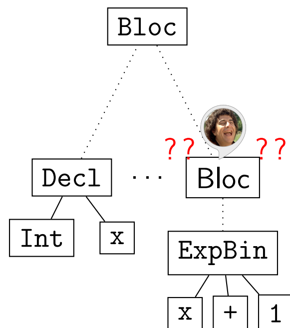
Dans plusieurs blocs ?



Dans plusieurs blocs ?

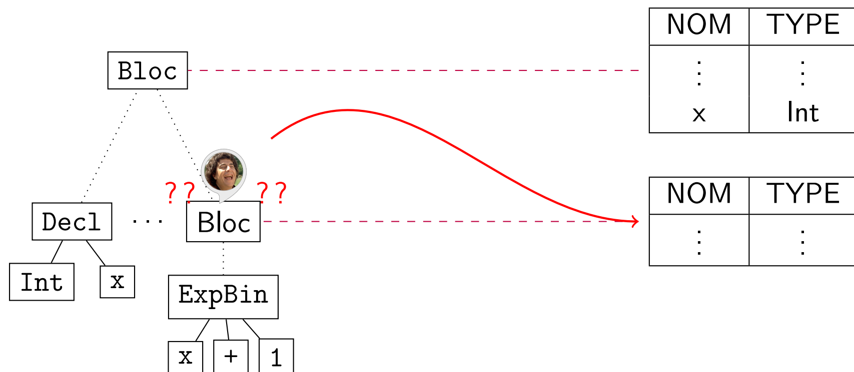


Dans plusieurs blocs ?

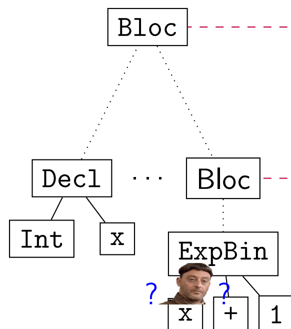


NOM	TYPE
⋮	⋮
x	Int

Dans plusieurs blocs ?



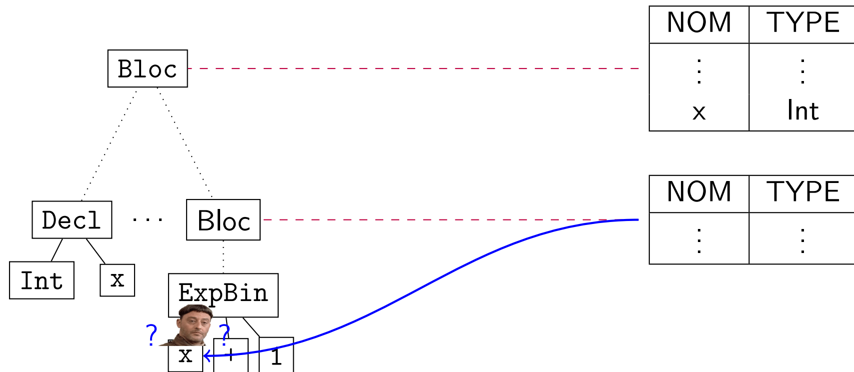
Dans plusieurs blocs ?



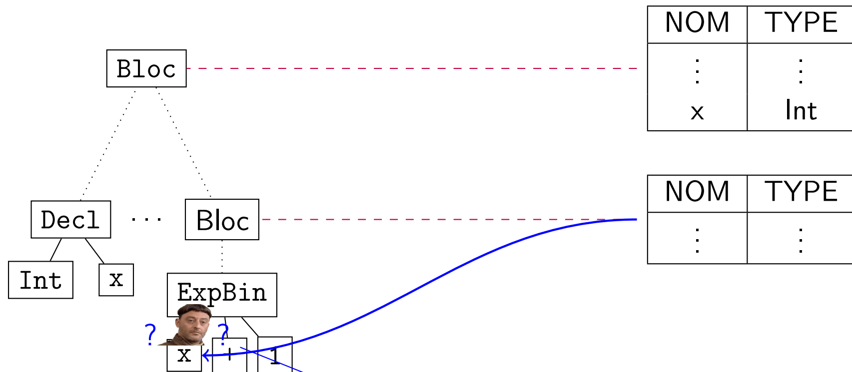
NOM	TYPE
⋮	⋮
x	Int

NOM	TYPE
⋮	⋮

Dans plusieurs blocs ?

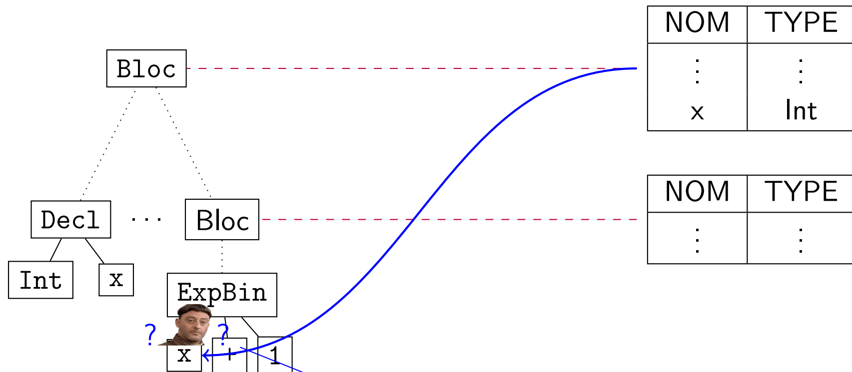


Dans plusieurs blocs ?



Point de x en vue, messire !

Dans plusieurs blocs ?



`Certes, return Int sans plus tarder`

Comment on représente l'imbrication des blocs

On commence par représenter tous les blocs à plat : on a une table qui associe chaque bloc aux déclarations qui lui sont locales :

```
Map <Block,Map <String,Type> >
```

même si les blocs sont imbriqués.

Mais pour savoir si une variable a été déclarée dans un bloc parent, on conserve l'information de la liste des blocs dans lesquels on est entrés, c'est ceux que l'on sera autorisés à consulter pour notre recherche de variable.

Quelle est la structure de donnée la plus adaptée pour cette représentation ?

L'historique des entrées/sorties de blocs

Une solution : les piles de java, (`java.util.Stack`), avec les méthodes `push`, `pop`, et `peek` (retourne le sommet de la pile sans le supprimer).

Ainsi, si l'historique est représenté par une pile p :

```
visit(Block b){
  creerTableLocale(b);//Map<String,Type>
  p.push(b);
  super.visit(b); //visiteur generique
  p.pop(b);
}
```

À tout moment de la visite, `p` représente bien, dans l'ordre, l'imbrication des blocs dans laquelle on se trouve.

Enregistrer les déclarations

```
visit(Declaration d){
  Type t = table.recherche(d.getNom(),p);//a suivre
  if(t!=null){
    //erreur : variable deja declaree
  }
  if(p.isEmpty()){
    //erreur : on n'est dans aucun bloc
  }
  else table.get(p.peek()).put(d.getNom(),d.getType());
}
```

Dans la table des symboles, `blocks` représente l'ensemble des tables locales évoquée plus tôt.

```
recherche(String nom, Stack<Block> p){
  Type t = null;
  for(Block b: p){
    Map<String,Type> table = blocks.get(b);
    if(table == null){
      //erreur
    }
    t=table.get(nom);
    if(t!=null){
      return t;//premier resultat dans la pile des blocs
    }
  }
  return t;
}
```

Un peu de structure

Pour la vérification de type, une articulation naturelle du code consisterait en l'organisation suivante :

- Une classe `SymbolTable` pour la gestion de la table des symboles : table globale `blocks`, méthode de recherche.
- Une classe `TableBuilder` pour la construction de la table, héritant du visiteur générique de l'AST. Ce visiteur comportera un champ `table` correspondant à la classe précédente. Question : quel est le type du visiteur, quelles sont les méthodes de visite implémentées ?
- Une classe `TypeChecker` pour la vérification de types, qui hérite aussi du visiteur générique. Question : idem. Le vérificateur se basant sur la table, il sera instancié avec un champ de la classe `SymbolTable`.

[Pour illustrer cette articulation, un petit coup d'œil au `Main` du compilateur]

Qu'est-ce qui nous manque ?

Qu'est-ce qui nous manque ?

- La gestion des erreurs

Qu'est-ce qui nous manque ?

- La gestion des erreurs
- Les fonctions

Gérer les erreurs de typage

Pour plus de souplesse, les erreurs peuvent être stockées sous formes de listes de chaînes de caractères : une classe `Errors` pourra implémenter cela, avec les méthodes adaptées pour l'ajout, l'affichage.

Un choix possible : la méthode `add` de `Errors` attend une chaîne, et un `Nœud` ; et sa méthode `print` utilise la position du nœud pour préfixer le message.

Une instance de `Errors` sera présente dans tous les visiteurs évoqués plus haut, de façon à ce qu'à chaque phase de l'analyse sémantique, on puisse s'assurer que la liste d'erreurs soit vide avant de continuer.

Et les fonctions ?

De quoi a-t-on besoin pour vérifier que tout se passe bien au niveau du typage, en présence de fonctions ?

Pour sa définition :

Et les fonctions ?

De quoi a-t-on besoin pour vérifier que tout se passe bien au niveau du typage, en présence de fonctions ?

Pour sa définition :

- La valeur retournée est du bon type
- Une valeur est forcément retournée
- Les paramètres sont utilisés avec le bon type

Pour son utilisation (appels) :

Et les fonctions ?

De quoi a-t-on besoin pour vérifier que tout se passe bien au niveau du typage, en présence de fonctions ?

Pour sa définition :

- La valeur retournée est du bon type
- Une valeur est forcément retournée
- Les paramètres sont utilisés avec le bon type

Pour son utilisation (appels) :

- Le nombre d'arguments correspond à la définition
- Ils ont le bon type
- Le résultat est utilisé avec le type de retour de la fonction (`int x = f(...);`)

Et les fonctions ?

En pratique

Représenter la *signature* des fonctions par une classe enregistrant les types et noms des paramètres, le nom de la méthode, et le type de retour.

Dans la tables des symboles : une donnée supplémentaire, (`Map`) associant chaque nom de fonction à sa signature. [3]

3. Pas de pile, car toutes les fonctions sont définies au même niveau, celui du programme.

[Examen du code d'un vérificateur de type pour langage impératif simple avec fonctions]

Bilan d'étape

On a terminé la partie du cours sur la vérification de types.

Bilan d'étape

On a terminé la partie du cours sur la vérification de types.

Bientôt sur vos écrans :

- Représentation intermédiaire, et traduction.
- Assembleur et génération de code