Partie VI : Analyse des types

- Règles de typage
- Grammaires attribuées
- Attributs
- Spécification complète de la grammaire contextuelle de misc
- Comment passer d'une spécification à un compilateur

Règles de typage

- La déclaration des identificateurs n'est pas la seule chose à vérifier dans un compilateur.
- En misc, comme dans la plupart des langages de programmation, les expressions ont un type.
- Il faut vérifier que les types sont corrects.
- Exemples:
  - Les opérandes de `+` doivent être des entiers.
  - Les opérandes de `=` doivent être compatibles. (`Int, Int` est OK, de même que `List[Int], List[Int],` mais pas `Int, List[Int]`)
  - Le nombre d'arguments passé à une fonction doit être égal au nombre de paramètres de cette fonction.
  - etc.
- Comment spécifie-t-on les règles de typage ?

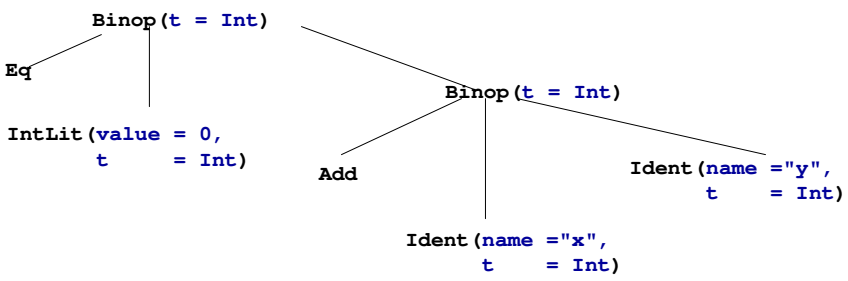
Grammaires attribuées

- On ajoute des attributs et des règles d'attribution aux grammaires non contextuelles.
- A chaque symbole peuvent être attaché des attributs.
- A chaque production peuvent être associées des règles d'attribution qui mettent en relation les attributs des symboles de la production.
- Exemple:

```
E(t) = Binop Add E(t1) E(t2)        t1 = t2 = Int, t = Int
     | Binop Eq E(t1) E(t2)         t1 = t2 = Int, t = Int
     | Ident ident                  sym = lookup(ident), t = sym.type
     | IntLit int                   t = Int
```


Grammaires attribuées (2)

- Une *attribution* est une affectation de tous les attributs d'un arbre syntaxique satisfaisant toutes les règles d'attribution.
- Exemple : `0 == (x + y)` où `x` et `y` sont de type `Int`



Grammaires attribuées (3)

- Un programme est légal si :
  - c'est une phrase du langage décrit par la grammaire non contextuelle, et
  - il existe une attribution pour son arbre.
- Un langage est donc caractérisé complètement par sa *grammaire non contextuelle* et sa *grammaire contextuelle*.
- Toutefois, rien n'est encore dit au sujet de la *signification* d'un programme (qu'on appelle aussi sa *sémantique*).

Attributs

- Des exemples typiques d'attributs sont:
  - Le *type* d'une expression
  - Le *symbole* produit par une déclaration
  - La *table des symboles* (ou *portée*) produit par un ensemble de déclarations.
- Les tables des symboles sont souvent représentées par des variables globales au lieu d'un ensemble d'attributs.
- Voilà une différence entre la théorie et la pratique !
- Il est toutefois important de s'assurer que la pratique ne détruit pas les concepts théoriques. Les tables des symboles peuvent être vues comme un attribut, mais on choisit une représentation centralisée plus performante.

Spécification de la grammaire contextuelle de misc

```
P     = Program { D } B(tb)
```
Crée une portée pour le programme entier, et l'utilise pour analyser les déclarations {D} et l'expression principale B.

```
D     = FunDef ident { F }({ta}) T(tt) E(te)
```
Analyse les arguments {F} dans une nouvelle portée, imbriquée dans la portée courante. Ajoute un nouveau symbole dans la portée courante, de nom ident et de type tf. Analyse ensuite le corps E dans la portée des arguments.

```
F(t)  = Formal ident T(tt)
```
Ajoute un nouveau symbole dans la portée courante, de nom ident et de type tt.

```
B(t)  = Block { S } E(t1)
```
Analyse {S} et E dans une nouvelle portée, imbriquée dans la portée courante.

Spécification de la grammaire contextuelle de misc (2)

```
T(t) = BasicType Unit
     | BasicType Int
     | BasicType Any
     | ListType T(tt)
     | FunType { T }({ts}) T(tt)

S    = VarDef F(tf) E(te)
```
Analyse l'expression E et F dans la portée courante.

```
     | While E(tc) E(te)

     | Exec E(te)

E(t) = UnitLit

     | IntLit int

     | NilLit

     | ...
```

Spécification de la grammaire contextuelle de misc (3)

```
E(t) = ...
     | Ident ident

     | Assign ident E(t1)


     | Apply E(tf) { E }({ta})


     | If E(tc) E(t1) E(t2)

     | ...
```


Spécification de la grammaire contextuelle de misc (4)

```
E(t) = ...
     | Binop Cons E(t1) E(t2)

     | Binop O E(t1) E(t2)                    O != Cons

     | Listop Head E(t1)

     | Listop Tail E(t1)

     | Listop IsEmpty E(t1)

     | B(t1)
```

Le Sous-Typage

- Nouvelle Relation `S <: T`
  - `S` est un *sous-type* de `T`, ou `S` est *compatible* avec `T`.
- Règles:

$$T <: T \qquad T <: Any \qquad Bottom <: T$$

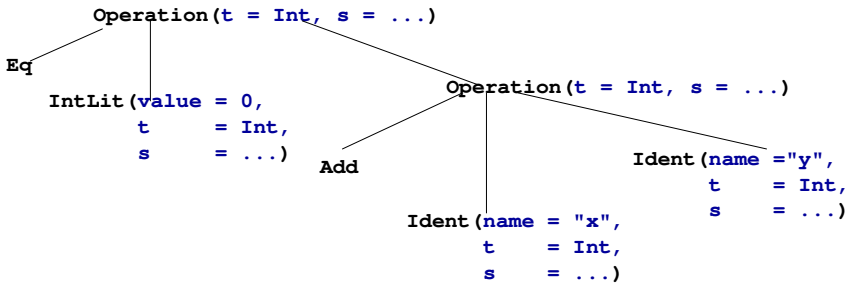
$$\frac{\{T2\} <: \{T1\} \quad R1 <: R2}{(\{T1\}) \Rightarrow R1 \;<:\; (\{T2\}) \Rightarrow R2} \qquad \frac{S <: T}{List[S] <: List[T]}$$

- Exemple: `List[bottom] <: List[Int]`

Comment obtenir un compilateur à partir d'une spécification

- Au lieu de *deviner* les attributs et de *vérifier* ensuite qu'ils satisfont les règles d'attribution, on doit les *calculer*.
- Les attributs sont en général calculés à partir de la valeur d'autres attributs.
- Important : les attributs doivent être affectés une et une seule fois !
- Les attributs peuvent être distingués en fonction de la manière dont ils « traversent » l'arbre syntaxique.
  - Certains le traversent de bas en haut – ils sont dits *synthétisés*.
  - Certains le traversent de haut en bas – ils sont dits *hérités*.
- Dans un compilateur, les attributs synthétisés sont représentés par les type de retour (ou paramètres de sortie en C/C++) des visiteurs.
- Les attributs hérités sont représentés par les paramètres (d'entrée) des visiteurs.

Exemple



- s est hérité
- t est synthétisé
- name, value sont intrinsèques

Représentation des attributs

- Certains attributs doivent être présents dans des phases ultérieures de la compilation – ils sont dits *persistants.*
- D'autres attributs ne sont utiles que lors de la vérification de type – ils sont dits *temporaires*.
- Les attributs persistants peuvent être stockés comme champs supplémentaires dans l'arbre syntaxique.
- Les attributs temporaires sont des paramètres et des résultats des méthodes des visiteurs :
  - Synthétisé, temporaire = résultat de visiteur.
  - Hérité, temporaire = paramètre de visiteur.
- Certains attributs peuvent aussi être représentés par des variables globales. Cela est plus simple si ces attributs changent peu souvent.

Exemple

```
E(t,s) = Binop Add E(t1, s1) E(t2, s2)
       | Binop Eq E(t1, s1) E(t2, s2)
       | Ident name (sym)
       | ...
```

- Les attributs suivants existent :
  - `t` : (type) synthétisé, persistant
  - `sym` : (symbole) intrinsèque, persistant
  - `s`: (portée) hérité, temporaire
- On a donc besoin pour l'analyse d'un visiteur structuré de la manière suivante.

Exemple (2)

```
public class Analyzer implements Visitor {
  private Scope scope;

  Analyzer(Scope scope) { this.scope = scope;}

  Type analyze(Tree tree, Scope scope) {
    tree.apply(new Analyzer(scope));
    return tree.type;
  }

  void caseBinop(Binop tree){
    if (tree.tag == Tree.ADD) {
      Type t1 = analyze(tree.left, scope);
      Type t2 = analyze(tree.right, scope);
      if ((t1 != Type.INT_TYPE) &&
          (t1 != Type.BAD_TYPE)) {
        Report.error(tree.left.pos,
          "expected integer, " +
          "found " + t1);
      }
      if ((t2 != INT_TYPE) &&
          (t2 != BAD_TYPE)) {
        Report.error(tree.right.pos,
          "expected integer, " +
          "found " + t2);
      }
      tree.type = Type.INT_TYPE;
    }
    ...
  }

  public void caseIdent(Ident tree) {
    tree.sym = scope.lookup(tree.name);
    if (tree.sym == null) {
      error(tree.pos,
        tree.name + " undefined");
      tree.type = Type.BAD_TYPE;
    } else {
      tree.type = sym.type;
    }
  }

  ...
}
```

Optimisations :

1. La création d'un visiteur pour chaque nœud visité prend du temps. On a meilleur temps de réutiliser le visiteur courant.

```
public Type analyze(Tree tree, Scope scope) {
  Scope scope1 = this.scope;
  this.scope = scope;
  tree.apply(this);
  this.scope = scope1;
  return tree.type;
}
```

2. L'attribut `scope` ne varie pas fréquemment.
   ⇒ on peut utiliser une méthode d'analyse plus efficace qui ne fait pas la sauvegarde/changement/restauration de la portée.

```
public Type analyze(Tree tree) {
  tree.apply(this);
}
```


Optimisations (2)

3. On doit souvent comparer deux types et signaler une erreur s'ils ne sont pas compatibles.
   ⇒ On peut créer une fonction `checkType` qui se charge de faire cela

```
public Type checkType(int pos, Type found, Type required) {
  if (found.isSubType(required)) return found;
  error(pos, "type error: " +
    "expected " + required + "," +
    "found " + found);
  return Type.BAD_TYPE;
}
```

Dès lors `caseBinop` peut s'écrire de manière plus concise.

```
public void caseBinop(Binop tree) {
  if (tree.tag == Tree.ADD) {
    checkType(tree.pos, analyze(tree.left), Type.INT_TYPE);
    checkType(tree.pos, analyze(tree.right),Type.INT_TYPE);
    tree.type = Type.INT_TYPE;
  }
  ...
}
```

Résumé

- L'analyse de noms et la vérification des types sont parmi les tâches les plus complexes d'un compilateur.
- Tâches: (1) Vérifier que l'arbre donné est légal par rapport aux règles du langage (grammaire contextuelle).
  (2) Déterminer certains attributs de l'arbre (p.ex.: `type, sym`), qui seront nécessaires dans des phases ultérieures.
- La grammaire contextuelle peut être spécifiée au moyen d'une grammaire attribuée.
- Les attributs sont des champs des nœuds de l'arbre, calculés au moyen des règles d'attribution.
  - Trois classes d'attributs : *hérités, synthétisés, intrinsèques*.
- Mise en œuvre : visiteur d'arbre. Les attributs sont représentés par :
  - des champs de l'arbre, ou
  - des paramètres ou des résultats de la méthode du visiteur, ou
  - des champs de la classe du visiteur.

Spécification complète de la grammaire contextuelle de misc

`P = Program { D } B(tb)`

Crée une portée pour le programme entier, et l'utilise pour analyser les déclarations {`D`} et l'expression principale `B`.

`B(t) = Block { S } E(t)`

Analyse {`S`} et `E` dans une nouvelle portée, imbriquée dans la portée courante.

`D = FunDef ident { F }({ta}) T(tt) E(te)`

```
tf = ({ta}) => tt
te <: tt
```

Analyse les arguments {`F`} dans une nouvelle portée, imbriquée dans la portée courante. Ajoute un nouveau symbole dans la portée courante, de nom `ident` et de type `tf`. Analyse ensuite le corps `E` dans la portée des arguments.

`F(t) = Formal ident T(t)`

Ajoute un nouveau symbole dans la portée courante, de nom `ident` et de type `t`.

Spécification complète de la grammaire contextuelle de misc (2)

```
T(t) = BasicType Unit                t = Unit
     | BasicType Int                 t = Int
     | BasicType Any                 t = Any
     | ListType T(tt)                t = List[tt]
     | FunType { T }({ts}) T(tt)     t = ({ts}) => tt

S    = VarDef F(tf) E(te)            te <: tt
```

Analyse l'expression `E` et `F` dans la portée courante.

```
     | While E(Int) E(te)

     | Exec E(te)

E(t) = UnitLit                       t = Unit

     | IntLit int                    t = Int

     | NilLit                        t = List[Bottom]

     | ...
```


Spécification complète de la grammaire contextuelle de misc (3)

```
E(t) = ...

     | Ident ident                   sym = lookup(ident)
                                     t = sym.type

     | Assign ident E(t)             sym = lookup(ident)
                                     sym.isVariable()
                                     t <: sym.type

     | Apply E(tf) { E }({ta})       tf = ({ts}) => t
                                     {ta} <: {ts}

     | If E(Int) E(t1) E(t2)         t1 <: t2 or t2 <: t1
                                     t = max(t1, t2)

     | ...
```

Spécification complète de la grammaire contextuelle de misc (4)

`E(t) = ...`	
`| Binop Cons E(t1) E(t2)`	`t2 <: List[t1] or List[t1] <: t2` `t = max(List[t1], t2)`
`| Binop O E(t1) E(t2)`	`O != Cons` `t = t1 = t2 = Int`
`| Listop Head E(t1)`	`List[t] = t1`
`| Listop Tail E(t)`	`t <: List[Any]`
`| Listop IsEmpty E(t1)`	`t1 <: List[Any]` `t = Int`