Projet du premier semestre L3

Sébastien Briais

ENS Lyon

12 octobre 2007

À propos de l'analyseur lexical

Les difficultés :

- les nombres !
  001234567890 sont trois mots de la micro-syntaxe de `drei`.
- Les commentaires et la division
  Un commentaire va jusqu'au bout de la ligne ou EOF

Le code doit être lisible (indentation, commentaires, ...)
L'abstraction en terme de flux (streams) n'a pas été comprise par tous.

Les flux

Un type de donnée pour représenter les listes potentiellement infinies :

- On peut tester si le flux est vide ou non.
- On peut connaitre la première valeur du flux.
- On peut consommer la première valeur du flux.

```
type 'a t

val is_empty : 'a t -> bool
val peek : 'a t -> 'a option
val drop : 'a t -> unit
```

La suite du projet

- Chacun s'est vu attribué une couleur parmi R, Y, G, C, B
- 29 élèves = 13 binômes + 1 trinôme
- Contraintes :

	R	Y	G	C	B
R	o	o	n	n	n
Y	o	o	o	n	n
G	n	o	o	o	n
C	n	n	o	o	o
B	n	n	n	o	o

- Il est *très fortement* conseillé aux groupes n'étant pas teintés B de repartir du corrigé disponible sur le site web (version ocaml ou C au choix)...
- C'est même *obligatoire* pour les groupes teintés R
- Former les groupes d'ici mercredi 10 octobre minuit et m'envoyer un mél par groupe pour m'en donner la composition.

Actions sémantiques

Un analyseur syntaxique fait généralement plus que simplement reconnaitre la syntaxe. Il peut :

- Évaluer du code (interpréteur).
- Émettre du code (compilateur simple passe).
- Construire une structure de données internes (compilateur multi passes).

De façon générale, un analyseur syntaxique réalise des actions sémantiques :

- analyseur à descente récursive : intégrées aux routines de reconnaissance.
- analyseur ascendant généré automatiquement : ajoutées à la grammaire soumise au générateur d'analyseurs.

Exemple

Interpréteur d'expressions arithmétiques :

$$E = T\{\text{“}+\text{”}\ T|\text{“}-\text{”}\ T\}$$
$$T = F\{\text{“}*\text{”}\ F|\text{“}/\text{”}F\}$$
$$F = number|\text{”}(\text{”}\ E\text{”})\text{”}$$

...

Arbres de syntaxe

Dans un compilateur multi-passes, l'analyseur construit explicitement un arbre de syntaxe.
Les phases suivantes travaillent sur l'arbre et non sur le source.
Un arbre de syntaxe est

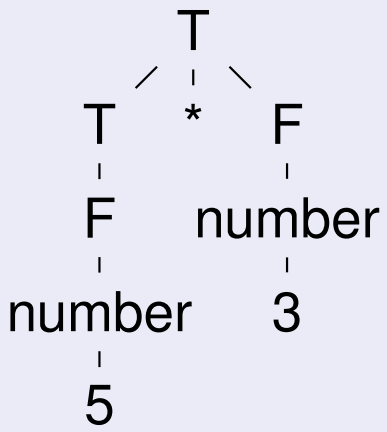
- *concret* : s'il correspond directement à la grammaire du langage
- *abstrait* : s'il correspond à une grammaire simplifiée.

On utilise généralement un arbre de syntaxe abstraite.

Arbres de syntaxe concrète

Exemple : Un arbre de syntaxe concrète pour une E.A.

En utilisant la grammaire non-contextuelle des expressions arithmétiques définies précédemment, on obtient pour l'expression `5 * 3` l'arbre de syntaxe concrète suivant :



La grammaire abstraite

On peut simplifier en omettant les informations redondantes.

- Les parenthèses sont inutiles.
- Les terminaux peuvent être implicites

Ces simplifications amènent à la définition d'une nouvelle grammaire non-contextuelle plus simple : la grammaire abstraite.

Exemple pour les E.A.

$$\begin{aligned} \mathit{Expr} &= \mathit{BinOp}\ \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\ &\mid \mathit{IntLit}\ \mathit{int}. \end{aligned}$$

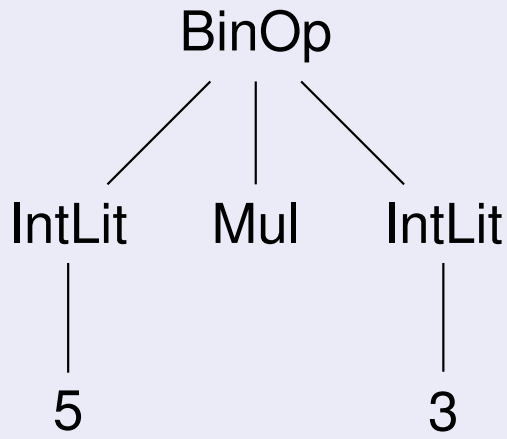
$$\mathit{Op} = \mathit{Add} \mid \mathit{Sub} \mid \mathit{Mul} \mid \mathit{Div}.$$

D'autres grammaires abstraites sont possibles...

Arbres de syntaxe abstraite

Exemple : Un arbre de syntaxe abstraite pour une E.A.

Pour l'expression `5 * 3`, on obtient l'arbre de syntaxe abstraite suivant :



En caml

```
type binop = Add | Sub | Mul | Div

type expr =
  | BinOp of binop * expr * expr
  | IntLit of int
```

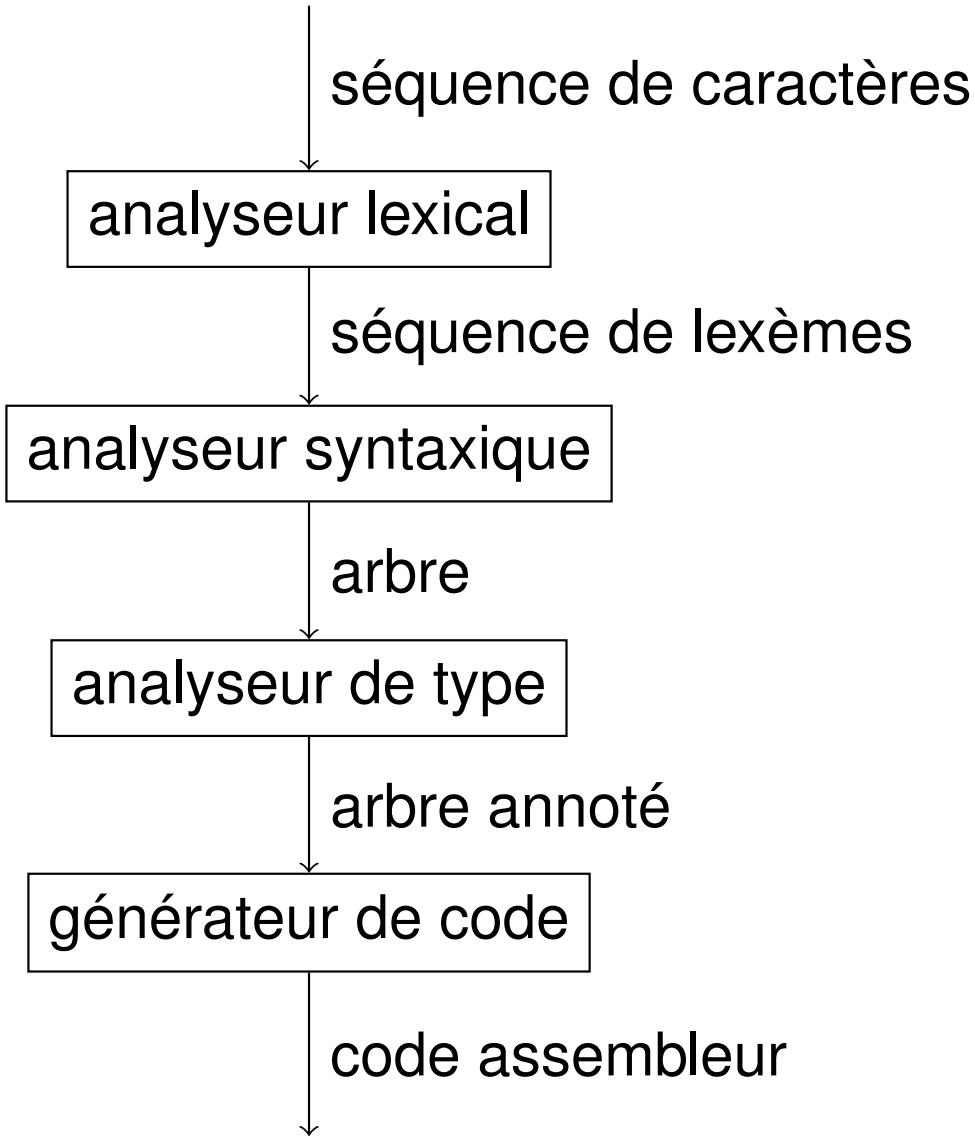
Il faut aussi se rappeler de la position dans le texte source pour pouvoir donner des messages d'erreur explicites.

```
type position = int * int

type binop = Add | Sub | Mul | Div

type expr =
  | BinOp of position * binop * expr * expr
  | IntLit of position * int
```

Structure du compilateur



Sucre syntaxique

On va faire quelques simplications pour faciliter les phases suivantes :

expression sucrée	expression désucrée
`e1 || e2`	`!(!e1 && !e2)`
`false`	`0`
`true`	`1`
`"abc"`	`new Cons(A, new Cons(B, new Cons(C ,new Nil())))`
	où `X` est le code ascii du caractère `x`
`""`	`new Nil()`
`if (Expr) Statement`	`if (Expr) Statement else {}`

Sémantique d'un langage

- Par exemple, en C, que fait :

```
int x = 0;
x = x++;
```

- Ordre d'évaluation des arguments.
  En ocaml, non spécifié :

```
let f x y = x + y in
  f (print_int 1;1) (print_int 2;2);;
```

- Règles de portée.
- On va décrire le fonctionnement des programmes `drei` en donnant une sémantique opérationnelle de `drei`.

Analyse des noms

La propriété “Chaque nom a besoin d’être déclaré” dépend du contexte.
On rejette les programmes bien formés syntaxiquement qui ne respectent pas cette règle.

```
class A {
  val a : Int;
  def f(a:Int) : Int = {
    return a + this.a
  };
}

printInt(new A(1).f(2));
```

Portées

Chaque nom a une *portée* : zone de visibilité à l'intérieur du programme.

- La portée d'un identificateur s'étend de l'endroit de sa déclaration jusqu'à la fin du bloc englobant
- Il est illégal de se référer à un identificateur en dehors de sa portée.
- Il est illégal de déclarer deux identificateurs avec le même nom si l'un est dans la portée de l'autre.

Un bloc :

- n'importe quoi entre accolades.
- la zone incluant la liste des paramètres et le corps de la méthode.

Représentation des contextes

- Par une table de symboles.
- Un symbole identifie de manière *unique* chacun des noms déclarés dans le programme. Il contient toutes les informations nécessaires au compilateur le concernant.
- Représentation des contextes : impérative ou fonctionnelle.

Analyse des types

- Aider le programmeur en éliminant les programmes *absurdes*.

```
printInt(1 + "");
```

- Imparfait.

```
printInt(1 && "");
```

- Parfois intrinsèquement liée à l'analyse de noms.

Propriétés de typage

fortement/faiblement typé

- Un langage est *fortement typé* (ou sûr) si la violation d'une règle de typage entraine une erreur.
- Il est dit *faiblement typé* (ou non typé) dans les autres cas. En particulier, si le comportement n'est plus spécifié en cas de typage incorrect.

statiquement/dynamiquement typé

- Un langage est dit *statiquement typé* s'il existe un système de typage qui peut détecter des programmes incorrects avant leur exécution.
- Il est dit *dynamiquement typé* dans les autres cas.

Exemple : Quelques règles de typage de Drei

- Les opérandes de + doivent être des entiers.
- Les opérandes de == doivent être compatibles (Int avec Int est compatible de même que List avec List, mais pas Int avec List).
- Le nombre d'arguments passés à une méthode doit être égal au nombre de paramètres formels de cette méthode.
- etc.

Typage, sous-typage, ensembles

- B est sous-type de A ($B <: A$) si partout on l'attendait des objets de type A, on peut passer des objets de type B.
- Typage nominal vs typage structurel.
  Typage nominal : les notions d'héritage et de sous-typage sont liées.
  Typage structurel : héritage et sous-typage sont deux notions orthogonales.
- Interprétation ensembliste du typage :

x est de type A	$x \in A$
B est sous-type de A	$B \subseteq A$