Interface PM
IFT2905
Feuille de notes

Franz Girardin

23 mai 2024

Table des matières

Section 1

Extended Backus-Naur Form

Définition 1 EBNF

Une **grammaire** qui fournit un façon d'exprimer formellement la **structure** d'un langage.

- **Clarifie** et communique la struct.

$$\texttt{Lang. prog.} ::= \{\text{symboles}\} \hookrightarrow \{\text{phrase}\}$$
$$\texttt{Vocabulaire} ::= \sum \text{phrases}$$
$$\texttt{Grammaire} ::= \{\text{Règles} \mid \texttt{règle} = f(\texttt{symbole})\}$$

Définition 2 Grammaire

La **grammaire** est l'ensemble des **règles** de **syntaxe** qui spécifie l'usage adéquat du langage

En **BNF**, un langage de programmation $L(G)$ est un **ensemble** composé d'éléments p tels qu'il est possible d'utiliser un élément de départ `dp` et une série de **règle de production** pour obtenir un nouvel élément p.

$$L(G) = \{\langle p\rangle \mid \texttt{dp} \hookrightarrow \cdots \hookrightarrow p\rangle\}$$

Exemple 1 Définition d'une catégorie en BNF

// Une catégirie composée d'élem. x_i

$\langle \texttt{cat}\rangle ::= x_1 x_2 \cdots x_n$

$\langle \texttt{bin}\rangle ::= 0$

$\langle \texttt{bin}\rangle ::= 1$

// Règle production d'un nouveau binaire

$\langle \texttt{bin}\rangle ::= \langle \texttt{bin}\rangle\langle \texttt{bin}\rangle$

// Application d'une règle de production

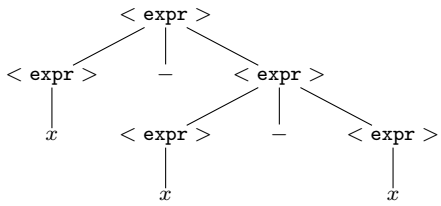
$\langle \texttt{bin}\rangle ::= \langle \texttt{bin}\rangle\langle \texttt{bin}\rangle \hookrightarrow \langle \texttt{bin}\rangle 0 \hookrightarrow 1\ 0$

Définition 3 Grammaire ambigue

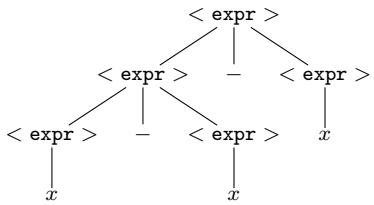
G est ambigue s'il $p \in L(G)$ tel que p a plus d'un *parse tree* ou **arbre de dérivation**.

Exemple 2 Expression ambigue $x - x - x$

Arbre de dérivation 1 :



Arbre de dérivation 2 :



Définition 4 Arbre de syntaxe abstraite (ASA)

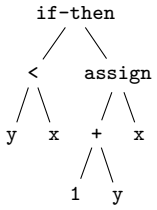
Structure *hiérarchique* qui représente la **structure syntaxique** abstraite d'une expression.

Exemple 3 ASA d'expr. conditionnelle

Soit l'expression suivante

```
if (x < y) then :
 x := y + 1
```

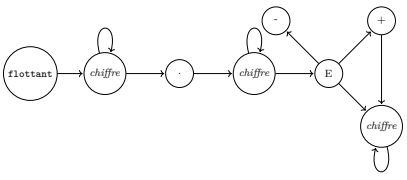
On a l'**ASA** :



Définition 5 Diagramme syntaxique

Il s'agit d'une représentation **graphique** d'une *règle de syntaxe* qui permet de visualiser l'**arrangement des symboles** terminaux et non terminaux.

Exemple 4 Diagramme syntaxique d'un flottant



Section 2

Base de la syntaxe Haskell

2.1 Fonctions standards Haskell

En plus des **opération de base** telles que `+ - * /` présentes dans les autres langages de programmation, Haskell possède plusieurs **fonctions standards**.

Exemple 5

```haskell
-- Retourne le premier nombre de la liste
head [1, 2, 3, 4, 5]
-- Retourne 1

-- Retire le premier element de la liste
tail [1, 2, 3, 4, 5]
-- Retourne [2, 3, 4, 5]

-- Selectionne le nth element d'une liste; index commence a 0
[1, 2, 3, 4, 5] !! 2
-- Retourne 3

-- Selectionne les nth premiers elements d'une liste
take 3 [1, 2, 3, 4, 5]
-- retourne [1, 2, 3]

-- Retire les nth premiers elements d'une liste
drop 3 [1, 2, 3, 4, 5]
-- retourne [4, 5]

-- Calcule la longueur d'une liste
length [1, 2, 3, 4, 5]
-- Retourne 5

-- Calcule la somme des nombres d'une liste
sum [1, 2, 3, 4, 5]
-- Retourne 15

-- Calcule le produit des elements d'une liste
product [1, 2, 3, 4, 5]
-- Retourne 120

-- Joint deux liste
[1, 2, 3] ++ [4, 5]
-- Retourne [1, 2, 3, 4, 5]

```

2.2 Application de fonctions

En mathématique, l'application de fonction est **mise en évidence** par l'usage de parenthèse et

la multiplication est **implicite** lorsque deux termes sont séparés d'un espace.

En Haskell, la **multiplication est est mise en évidence** par l'usage de $*$ et l'application de fonction est **implicite**, comme en lambda calcul.

Mathématique	Haskell
$f(a,b)+cd$	`f a b + c*d`
$f(a)+b$	`f a + b`
$f(x)$	`f x`
$f(x,y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x,g(y))$	`f x (g y)`
$f(x)g(y)$	`f x * g y`

2.3 Opérateur infixé

En Haskell, par défaut, les opérateurs de base `+ - * < > <= >= == /=`, etc. et tout ceux qui s'**utilisent normalement de manière infixé** peuvent être utilisé de façon préfixé en utilisant des parenthèses

Infixé	Préfixé
`a + b`	`(+) a b`
`a > b`	`(>) a b`

Les opérateurs qui sont des **fonctions qui s'utilisent de manière préfixé** sans parenthèse telles que `div` peuvent être utilisé de façon **infixé** en les encadrant des guillements arrière ``` `` ```.

Infixé	Préfixé
``a `div` b``	`div a b`
``a `mod` b``	`mod a b`
``a `quot` b``	`quot a b`
``a `rem` b``	`rem a b`

Exemple 6 Programmes simples

```haskell
-- Calcule le produit des nombre 1 a n
factorial n = product[1..n]

-- Calcule la moyenne d'une liste de nombres ns
Avg ns = sum ns  `div` length ns
```

2.4 Conventions d'écriture

► Les noms de **fonctions** et **variables** débutent par une *minuscule*.

► Les **argument de liste** ont habituellement un `s` à la fin de leur nom.

`xs` ::= liste de valeur de type `x`

► Les noms de **types** débutent avec une lettre majuscule

► Dans une séquence de définitions, chaque définition doit commencer précisément sur **la même colone**. Cela permet le **groupement implicite** de définitions.

✓	✗	✗
`a = 10`	`a = 10`	`a = 10`
`b = 20`	`b = 20`	`b = 20`
`c = 30`	`c = 30`	`c = 30`

Exemple 7 Groupement im-/ex-plicite déf.

```haskell
-- Implicite par l'usage d'espc.
a = b + c
  where
    b = 1
    c = 2
d = a * b


-- Explicite: usage de ';' et '{}'
a = b + c
  where
    {b = 1;
     c = 2}
d = a * b

```

2.5 Commandes GHCi utiles

Command	**Meaning**
`:load` *name*	Load script `name`
`:reload`	Reload current script
`:edit` *name*	Edit script `name`
`:edit`	Edit current script
`:type` *expr*	Show type of `expr`
`:?`	Show all commands
`:quit`	Quit GHCi

Arguments et retour

Java

```java
boolean SThan(int size, (@* String str) {
    return str.length < longueur;
}
```

Haskell

```haskell
{-
Definit le type de la fonction;
prend Int et String et renvoit bool
-}
lThan :: Int -> String -> Bool

lThan size str = l str < size
```