Travaux Dirigés Compilation: Feuille 2

Informatique 2ème année. ENSEIRB 2018/2019

—David Janin & Myriam Desainte-Catherine—

1 Expressions arithmétiques

On s'interesse ici à la conception d'une fonction récursive de production de code pour les expressions arithmétiques qui, pour une grammaire donné, s'appuie sur les arbres de dérivations de ces expressions.

La fonction souhaitée effectuera un parcours en profondeur à gauche d'abord de ces arbres de dérivation et, pour chaque règle (non terminal) et chaque terminal, produira (ou pas) un morceaux de code 3 adresses.

Contrainte : cette fonction n'aura pour seul argument que l'arbre (ou le sous-arbre) de dérivation dont on doit produire le code. Une fonction `vallex()` pourra être appelé pour avoir la "valeur lexical", c'est à dire la chaine de caractère, correspondant au dernier terminal lue. Le code sera produit, séquentiellement sur la sortie standard (`stdout`) via un printf ou équivalent. On pourra utiliser quelques variables globales pour permettre une "communication par effet de bord" entre les appels récursifs.

▸**Exercice 1.** *On considère la grammaire* `ETF` *définie par :*

```
E -> T | T + E
T -> F | F * T
F -> id | cst | (E)
```

avec les non-terminaux `E`, `T` *et* `F` *et les terminaux* `id`, `cst`, `+`, `*`, `(` *et* `)`.

(1) Dessiner les arbres de dérivations de $id + id * id$, de $id + (id * id)$ et de $(id + id) * id$. Qu'en conclure ?

(2) Proposer une fonction récursive de production de code 3 adresses pour les arbres de dérivations produit par cette grammaire.

▸**Exercice 2.** *On considère la grammaire* `E'TE'` *définie par :*

```
E  -> TE'
E' -> vide | +TE'
T -> FT'
T' -> vide | *TE'
F -> id | cst | (E)
```

avec les non-terminaux `E`, `E'`, `T`, `T'` *et* `F` *et les terminaux* `id`, `cst`, `+`, `*`, `(` *et* `)`.

(1) Dessiner les arbres de dérivations de $id + id * id$, de $id + (id * id)$ et de $(id + id) * id$.

(2) Proposer une fonction récursive de production de code 3 adresses pour les arbres de dérivations produit par cette grammaire.

`vide` *désignant ici le* mot vide *!*

2 Conditionelles

On considère la grammaire `COND` définie par :

```
I -> nop | C
C -> if B then I | if B then I else I
B -> id | true | false
```

avec les non-terminaux `I` (instructions), `C` (instructions conditionelles), `B` (booléens) et les terminaux `id`, `true`, `false`, `nop`, `if`, `then` et `else`.

▸**Exercice 3.** *Quel est le langage définit par cette grammaire ? Construire un arbre de dérivation pour*

```
if id then if id then nop else nop
```

En existe-t-il un autre ? Qu'en déduire ?

▸**Exercice 4.** *Proposer une grammaire non ambigüe permettant de définir le même langage. Indication : définir de façon plus détaillé les instructions conditionelles, en remplaçant le non terminal* `C` *par plusieurs non terminaux, afin de "forcer" l'association du "else" avec, soit le "if-then" qui précède le plus récent (lecture de gauche à droite) qui n'est pas déjà "fermé" par un "else", comme si "if-then" et "else" formaient des parenthèses ouvrantes et fermantes.*

▸**Exercice 5.** *Même question en cherchant à"forcer" l'association du "else" avec, soit le "if-then" qui précède le plus ancient (lecture de gauche à droite) qui n'est pas déjà "fermé" par un "else".*

3 Déclarations et portées des variables

On considère la grammaire `BLOCK` définie par :

```
B -> DL UL
DL -> D | D DL
UL -> U | U UL
D -> declare id
U -> use id
U -> begin B end
```

avec les non-terminaux `B` (bloc), `DL` (liste de déclarations), `D` (déclaration), `UL` (liste d'utilisations) et `U` (utilisation) et les terminaux `id`, `declare`, `use`, `begin` et `end`.

On supposera à nouveau qu'une fonction `vallex()` nous permet d'acceder à la valeur lexical du dernier terminal lue.

▸**Exercice 6.** *Proposer une arbre de dérivation de*

```
declare x
declare y
use x
begin
  declare y
  use y
  use x
end
use y
```

les valeurs lexicales x et y, ici explicitées, étant lu par la grammaire comme des terminaux `id`.

▸**Exercice 7.** *On veut résoudre les masquages tels qu'ils apparaissent ci-dessus. Proposer une fonction récursive de parcours des abres de dérivations de cette grammaire qui reproduit le code d'entrée mais en suffixant les noms des variables du numéro de leur bloc de définition.*

A partir de l'exemple précédent on souhaite produire :

```
declare x1
declare y1
use x1
begin
  declare y2
  use y2
  use x1
end
use y1
```

En numérotant les blocs dans l'ordre où on y entre pour la première fois. Attention, c'est bien `y2` *et pas* `y1` *qu'on utilise dans le sous-bloc.*

Indication : on pourra utilisé une pile globale (accessible comme une variable globale) qui contiendra des paires (nom-de-variable-déclarée, numéro-bloc-de-déclaration), qu'on pourra appeler environment *ou* table des symboles *qui sera mise à jour à chaque lecture de déclaration et/ou a chaque sortie de bloc. Une variable globale contenant le dernier numéro de bloc utilisé pourra aussi être utile.*