UFR IM²AG

UNIVERSITÉ Grenoble Alpes

UE INF404 - Projet Logiciel

Calculatrice : étape 1

Analyse et évaluation d'une expression arithmétique simple

L2 Informatique

Année 2022 - 2023

Rappel des précédents épisodes

Ecrire un interpréteur d'expressions arithmétiques

($\sim$ "calculatrice en ligne", comme la commande `bc` sous Linux)

Version initiale = "Expressions Arithmétiques Simples" (EAS)

- les opérandes sont des entiers
- opérateurs arithmétiques usuels (+, -, *, /)
- pas de priorités (évaluation de gauche à droite)

Exemples :

$$
\begin{aligned}
25 + 2 &\rightsquigarrow 27 \\
25 - 4 * 2 &\rightsquigarrow 42 \\
25 &\rightsquigarrow 25 \\
25 + *2 &\rightsquigarrow \text{erreur !} \\
-25 &\rightsquigarrow \text{erreur !} \\
25 \# 2 &\rightsquigarrow \text{erreur !} \\
25/0 &\rightsquigarrow \text{erreur !}
\end{aligned}
$$

Rappel des précédents épisodes

Ecrire un interpréteur d'expressions arithmétiques

($\sim$ "calculatrice en ligne", comme la commande `bc` sous Linux)

Version initiale = "Expressions Arithmétiques Simples" (EAS)

- les opérandes sont des entiers
- opérateurs arithmétiques usuels (+, -, *, /)
- pas de priorités (évaluation de gauche à droite)

Exemples :

$$
\begin{aligned}
25 + 2 &\rightsquigarrow 27 \\
25 - 4 * 2 &\rightsquigarrow 42 \\
25 &\rightsquigarrow 25 \\
25 + *2 &\rightsquigarrow \text{erreur !} \\
-25 &\rightsquigarrow \text{erreur !} \\
25 \# 2 &\rightsquigarrow \text{erreur !} \\
25/0 &\rightsquigarrow \text{erreur !}
\end{aligned}
$$

Rappel des précédents épisodes

Ecrire un interpréteur d'expressions arithmétiques

($\sim$ "calculatrice en ligne", comme la commande `bc` sous Linux)

Version initiale = "Expressions Arithmétiques Simples" (EAS)

- les opérandes sont des entiers
- opérateurs arithmétiques usuels (+, -, *, /)
- pas de priorités (évaluation de gauche à droite)

Exemples :

$$
\begin{array}{rcl}
25 + 2 & \rightsquigarrow & 27 \\
25 - 4 * 2 & \rightsquigarrow & 42 \\
25 & \rightsquigarrow & 25 \\
25 + *2 & \rightsquigarrow & \text{erreur !} \\
-25 & \rightsquigarrow & \text{erreur !} \\
25 \# 2 & \rightsquigarrow & \text{erreur !} \\
25/0 & \rightsquigarrow & \text{erreur !}
\end{array}
$$

Spécifier le langage d'entrée (1)

Alphabet = ensemble des caractères autorisés

V = {0, 1, 2, ..., 9, +, -, *, /, espace, tabulation, fin-de-ligne}

On pourra ajouter :

- des lettres (pour écrire des opérateurs `plus`, `moins`, `exp`, `log`, etc.)
- le caractère '.' (pour érire des "nombres à virgules")
- les parenthèses ouvrantes et fermantes
- etc.

Les caractères espace, tabulation, fin-de-ligne sont des **séparateurs**

Lexique = ensemble des "mots" du langage

Deux classes de lexèmes :

- entiers : séquence non vide de chiffres
- opérateurs : PLUS ('+'), MOINS ('-'), MULT ('*'), DIV ('/')

→ peuvent être définis par une **expression régulière**

Spécifier le langage d'entrée (1)

Alphabet = ensemble des caractères autorisés

V = {0, 1, 2, ..., 9, +, -, *, /, espace, tabulation, fin-de-ligne}

On pourra ajouter :

- des lettres (pour écrire des opérateurs `plus`, `moins`, `exp`, `log`, etc.)
- le caractère '.' (pour érire des "nombres à virgules")
- les parenthèses ouvrantes et fermantes
- etc.

Les caractères espace, tabulation, fin-de-ligne sont des **séparateurs**

Lexique = ensemble des "mots" du langage

Deux classes de lexèmes :

- entiers : séquence non vide de chiffres
- opérateurs : PLUS ('+'), MOINS ('-'), MULT ('*'), DIV ('/')

→ peuvent être définis par une **expression régulière**

Spécifier le langage d'entrée (1)

Alphabet = ensemble des caractères autorisés

V = {0, 1, 2, ..., 9, +, -, *, /, espace, tabulation, fin-de-ligne}

On pourra ajouter :

- des lettres (pour écrire des opérateurs `plus`, `moins`, `exp`, `log`, etc.)
- le caractère '.' (pour érire des "nombres à virgules")
- les parenthèses ouvrantes et fermantes
- etc.

Les caractères espace, tabulation, fin-de-ligne sont des **séparateurs**

Lexique = ensemble des "mots" du langage

Deux classes de lexèmes :

- entiers : séquence non vide de chiffres
- opérateurs : PLUS ('+'), MOINS ('-'), MULT ('*'), DIV ('/')

→ peuvent être définis par une **expression régulière**

Spécifier le langage d'entrée (2)

Syntaxe = ensemble des "phrases bien formées"

Expression régulière (ou automate) sur les lexèmes :

$$\text{entier}.(\text{operateur}.\text{entier})^*$$

Exo : ajouter le "moins unaire" ?

Sémantique = ensemble des phrases "qui ont un sens"

règles de l'arithmétique (pas de division par 0 !)

Spécifier le langage d'entrée (2)

Syntaxe = ensemble des "phrases bien formées"

Expression régulière (ou automate) sur les lexèmes :

$$\text{entier}.(\text{operateur}.\text{entier})^*$$

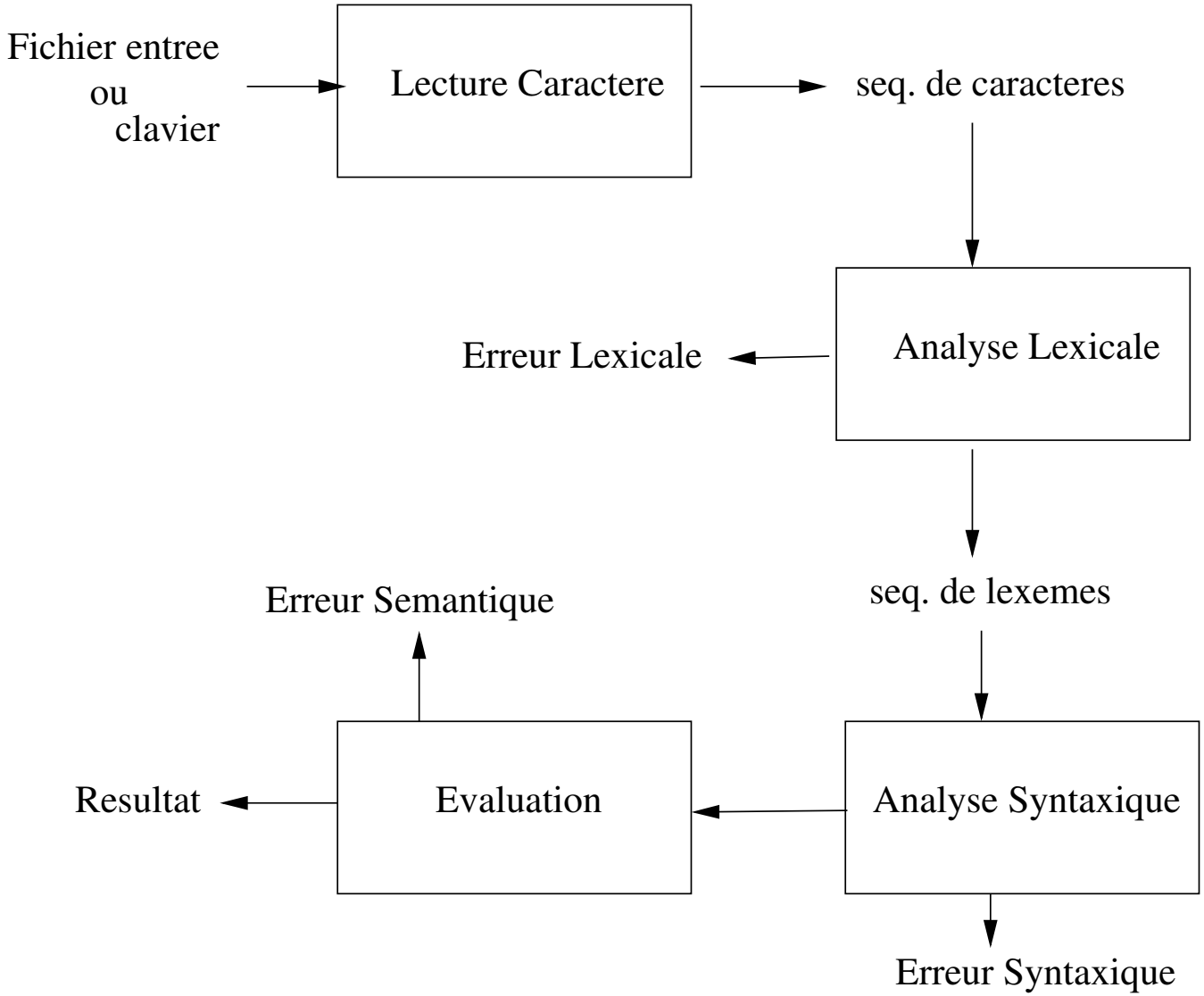
Exo : ajouter le "moins unaire" ?

Sémantique = ensemble des phrases "qui ont un sens"

règles de l'arithmétique (pas de division par 0 !)

Structure de la calculatrice

Quatre composants/modules principaux . . .



Lecture des caractères

Accès à une *séquence de caractères*

En entrée : un nom de fichier (ou la chaîne vide si lecture au clavier)
En sortie : accès séquentiel aux caractères du fichier / msg d'erreur ...

Primitives :

`demarrer_car`, `avancer_car`, `caractere_courant`, `fin_de_sequence_car`, `arreter_car`

Implémentation :

- utilisation des primitives de `stdio.h` ...
- module fourni, à compléter/modifier le cas échéant ...

Lecture des caractères

Accès à une *séquence de caractères*

En entrée : un nom de fichier (ou la chaîne vide si lecture au clavier)
En sortie : accès séquentiel aux caractères du fichier / msg d'erreur ...

Primitives :

`demarrer_car`, `avancer_car`, `caractere_courant`, `fin_de_sequence_car`, `arreter_car`

Implémentation :

- utilisation des primitives de `stdio.h` ...
- module fourni, à compléter/modifier le cas échéant ...

Lecture des caractères

Accès à une *séquence de caractères*

En entrée : un nom de fichier (ou la chaîne vide si lecture au clavier)
En sortie : accès séquentiel aux caractères du fichier / msg d'erreur ...

Primitives :

`demarrer_car`, `avancer_car`, `caractere_courant`, `fin_de_sequence_car`, `arreter_car`

Implémentation :

- utilisation des primitives de `stdio.h` ...
- module fourni, à compléter/modifier le cas échéant ...

Analyse lexicale

Accès à une *séquence de lexèmes*

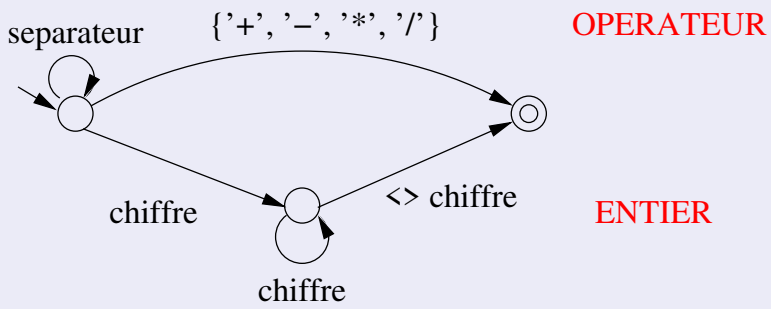
En entrée : une séquence de caractères
En sortie : accès séquentiel aux lexèmes / msg d'erreur ...
Primitives : définition d'un type `Lexeme` (publique) `demarrer`, `avancer`, `lexeme_courant`, `fin_de_sequence`, `arreter`

Implémentation :

- reconnaissance des lexèmes :

  entier = chiffre.(chiffre)* et operateur = '+' + '-' + '*' + '/'

  $\rightsquigarrow$ automate :



  Exo : ajouter des "nombres décimaux" ?

- module fourni, à compléter/modifier le cas échéant ...

Analyse syntaxique . . .

En entrée : une séquence de lexèmes

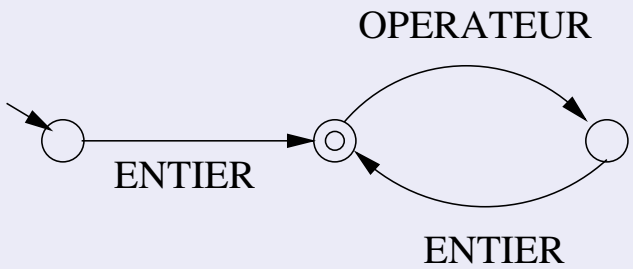
En sortie : valeur de l'expression arithmétique / msg d'erreur

Primitives :

```
int analyser (char *f, int *resulat) ;
// etat initial : f est un nom d'un fichier
// etat final :
    renvoie vrai si f contient une expression correcte syntaxiquement
    dans ce cas resultat est la valeur de l'expression
```

Implémentation :

entier.(operateur.entier)* $\rightsquigarrow$ automate !



Exo : ajouter le "moins unaire" ?

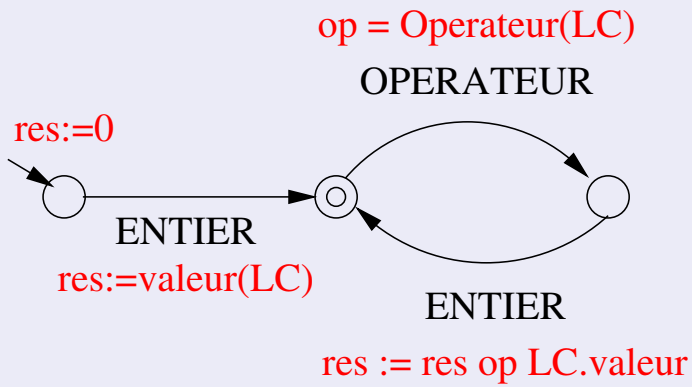
. . .et évaluation !

Ni parenthèses, ni priorité d'opérateurs $\rightsquigarrow$ évaluation de gauche à droite

$\Rightarrow$ évaluation possible pendant l'**analyse syntaxique**

Implémentation

Etendre l'automate avec des actions



Variables et fonctions auxilliaires

- `res` : valeur de l'expression (un entier)
- `LC` : lexeme courant (fourni par l'analyse lexicale)
- `Valeur(LC)` : valeur d'un lexeme ENTIER
- `Operateur(LC)` : type d'opérateur (PLUS, MOINS, etc.)

La suite ?

Etendre cette version

- nombres à virgules ($25.2 - 7.36$)
- nouveaux opérateurs (`exp`, `modulo`, `plus`, etc.)
- "moins unaire" ($-25 + 12$, $- - -25 + - - 12$)

Généralisation : priorités, donc parenthèses . . .

ex : $5 + 3 * 4 = 17 \quad (5 + 3) * 4 = 32$

- nouveaux lexèmes : `PARO` et `PARF`
  $\rightarrow$ on peut étendre l'analyse lexicale . . .

Mais :

- la syntaxe ne se décrit plus par un automate
  pas un **langage régulier** (imbrication de parenthèses)
- algo d'analyse et d'évaluation ? ? ?

$\Rightarrow$ **définir un nouveau formalisme ?**

La suite ?

Etendre cette version

- nombres à virgules ($25.2 - 7.36$)
- nouveaux opérateurs (`exp`, `modulo`, `plus`, etc.)
- "moins unaire" ($-25 + 12$, $- - -25 + - - 12$)

Généralisation : priorités, donc parenthèses ...

ex : $5 + 3 * 4 = 17 \quad (5 + 3) * 4 = 32$

- nouveaux lexèmes : `PARO` et `PARF`
  $\rightarrow$ on peut étendre l'analyse lexicale ...

Mais :

- la syntaxe ne se décrit plus par un automate
  pas un **langage régulier** (imbrication de parenthèses)
- algo d'analyse et d'évaluation ? ? ?

$\Rightarrow$ **définir un nouveau formalisme ?**