UFR IM²AG

UNIVERSITÉ Grenoble Alpes

# UE INF404 - Projet Logiciel

## Interpréteur

## Séquence d'affectations

L2 Informatique

Année 2023 - 2024

# Interpréteur pour un langage de programmation

**En entrée :** un programme en langage “source”, des options
**En sortie :**

- le résultat de l'exécution de ce programme, avec :
  - la valeur finale des variables
  - un mode “pas-à-pas” ?
  - etc.
- . . .ou un message d'erreur explicite . . .

**Exemples :**

- interpréteur Python
- interpréteur Caml
- interpéteur pour un langage de votre choix !

# Etape 1 : enrichir le langage des EAG

- notion de variable/identificateur
  + *affectation* ou liaison nom $\leftrightarrow$ valeur
- structures de contrôle (if, while, ...)
- types de données
- fonctions / procédures (+ récursivité ?)
- etc.

# Approche “incrémentale”

1. calculette (L0)

```
12 - 5 * (2+3)
```

2. affectations (L1)

```
X = 2 ;
Y = X + 2 ;
X = 3*Y - X ;
```

3. instructions conditionnelles et entrée-sorties (L2)

```
lire (X) ;
if X>0 then Y = X + 2 else Y = 3 ;
ecrire (Y) ;
```

4. instruction itérative (L3)
5. types, fonctions, etc.

# Approche “incrémentale”

1. calculette (L0)

```
12 - 5 * (2+3)
```

2. affectations (L1)

```
X = 2 ;
Y = X + 2 ;
X = 3*Y - X ;
```

3. instructions conditionnelles et entrée-sorties (L2)

```
lire (X) ;
if X>0 then Y = X + 2 else Y = 3 ;
ecrire (Y) ;
```

4. instruction itérative (L3)
5. types, fonctions, etc.

# Approche “incrémentale”

1. calculette (L0)

```
12 - 5 * (2+3)
```

2. affectations (L1)

```
X = 2 ;
Y = X + 2 ;
X = 3*Y - X ;
```

3. instructions conditionnelles et entrée-sorties (L2)

```
lire (X) ;
if X>0 then Y = X + 2 else Y = 3 ;
ecrire (Y) ;
```

4. instruction itérative (L3)
5. types, fonctions, etc.

# Approche “incrémentale”

1. calculette (L0)

```
12 - 5 * (2+3)
```

2. affectations (L1)

```
X = 2 ;
Y = X + 2 ;
X = 3*Y - X ;
```

3. instructions conditionnelles et entrée-sorties (L2)

```
lire (X) ;
if X>0 then Y = X + 2 else Y = 3 ;
ecrire (Y) ;
```

4. instruction itérative (L3)
5. types, fonctions, etc.

# Approche “incrémentale”

1. calculette (L0)

```
12 - 5 * (2+3)
```

2. affectations (L1)

```
X = 2 ;
Y = X + 2 ;
X = 3*Y - X ;
```

3. instructions conditionnelles et entrée-sorties (L2)

```
lire (X) ;
if X>0 then Y = X + 2 else Y = 3 ;
ecrire (Y) ;
```

4. instruction itérative (L3)
5. types, fonctions, etc.

# Langage L1 : séquences d'affectations

## Exemple

```
X := 12 * 3 ;
Y := X + 5 ;
X := X * 2 ;
```

## Lexique : 3 nouveaux lexèmes . . .

- IDF = seq. non vide de lettre/chiffre (débutant par lettre)
- AFF = opérateur d'affectation (:=)
- SEPINST = ';' (et fin_de_ligne devient un séparateur)

## Syntaxe : étendre la grammaire

$$\begin{aligned} \textit{seq\_aff} &\rightarrow \textit{aff}\ \textit{seq\_aff} \\ \textit{seq\_aff} &\rightarrow \varepsilon \\ \textit{aff} &\rightarrow \texttt{IDF AFF}\ \textit{eag}\ \texttt{SEPINST} \end{aligned}$$

et ajouter $\textit{facteur} \rightarrow \texttt{IDF}$

# Langage L1 : séquences d'affectations

## Exemple

```
X := 12 * 3 ;
Y := X + 5 ;
X := X * 2 ;
```

## Lexique : 3 nouveaux lexèmes . . .

- IDF = seq. non vide de lettre/chiffre (débutant par lettre)
- AFF = opérateur d'affectation (:=)
- SEPINST = ';' (et fin_de_ligne devient un séparateur)

## Syntaxe : étendre la grammaire

$$\begin{aligned} \textit{seq\_aff} &\rightarrow \textit{aff}\ \textit{seq\_aff} \\ \textit{seq\_aff} &\rightarrow \varepsilon \\ \textit{aff} &\rightarrow \texttt{IDF AFF}\ \textit{eag}\ \texttt{SEPINST} \end{aligned}$$

et ajouter $\textit{facteur} \rightarrow \texttt{IDF}$

# Langage L1 : séquences d'affectations

## Exemple

```
X := 12 * 3 ;
Y := X + 5 ;
X := X * 2 ;
```

## Lexique : 3 nouveaux lexèmes . . .

- IDF = seq. non vide de lettre/chiffre (débutant par lettre)
- AFF = opérateur d'affectation (:=)
- SEPINST = ';' (et fin_de_ligne devient un séparateur)

## Syntaxe : étendre la grammaire

$$\begin{aligned} \textit{seq\_aff} &\rightarrow \textit{aff}\ \textit{seq\_aff} \\ \textit{seq\_aff} &\rightarrow \varepsilon \\ \textit{aff} &\rightarrow \texttt{IDF AFF}\ \textit{eag}\ \texttt{SEPINST} \end{aligned}$$

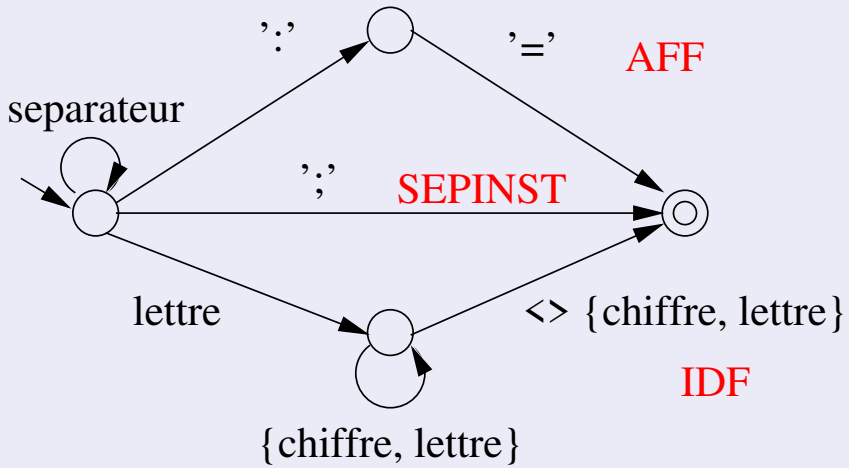
et ajouter $\textit{facteur} \rightarrow \texttt{IDF}$

# Analyse Lexicale

## Nouveaux lexèmes possibles :

```
typedef {ENTIER, PLUS, ..., AFF, SEPINST, IDF} Nature_Lexeme
```

## Nouvelles transitions de l'automate de `reconnaitre_lexeme()`




## Version plus simple :

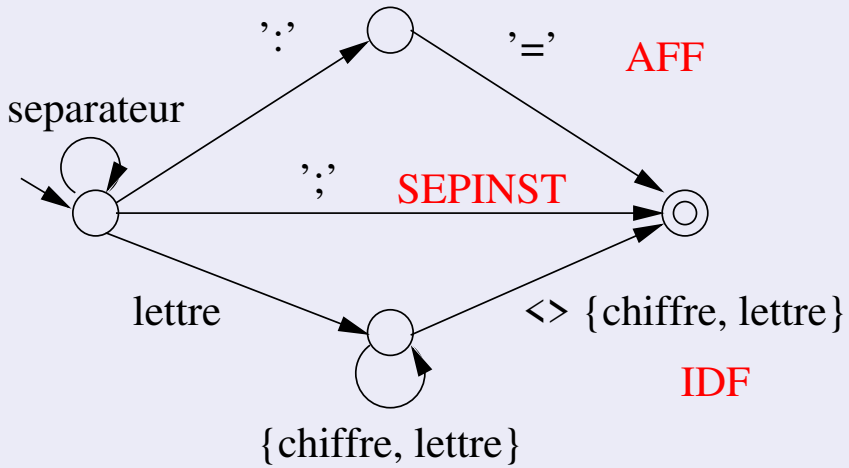
- affectation sur `un seul` caractère : '='
- IDF sur `une seule lettre` entre 'a' et 'z'

# Analyse Lexicale

## Nouveaux lexèmes possibles :

```
typedef {ENTIER, PLUS, ..., AFF, SEPINST, IDF} Nature_Lexeme
```

## Nouvelles transitions de l'automate de `reconnaitre_lexeme()`




## Version plus simple :

- affectation sur `un seul` caractère : '='
- IDF sur `une seule lettre` entre 'a' et 'z'

# Analyse Syntaxique : la grammaire complète

Grammaire des *eag* plus 3 nouvelles règles

| | | |
|---|---|---|
| *seq_aff* | $\rightarrow$ | *aff seq_aff* |
| *seq_aff* | $\rightarrow$ | $\varepsilon$ |
| *aff* | $\rightarrow$ | IDF AFF *eag* SEPINST |
| *eag* | $\rightarrow$ | *seq_terme* |
| ... | $\rightarrow$ | ... |
| *facteur* | $\rightarrow$ | *ENTIER* |
| *facteur* | $\rightarrow$ | PARO *eag* PARF |
| *facteur* | $\rightarrow$ | IDF |
| *op*1 | $\rightarrow$ | PLUS |
| *op*1 | $\rightarrow$ | MOINS |
| *op*2 | $\rightarrow$ | MUL |

$\Rightarrow$ Etendre l'analyse syntaxique pour prendre en compte ces nouvelles règles ?
Deux nouvelles procédures : `rec_seq_aff` et `rec_aff`

Et modifier `rec_facteur` ...

# Analyse Syntaxique : les 2 nouvelles procédures

$$\begin{aligned} \textit{seq\_aff} &\rightarrow \textit{aff}\ \ \textit{seq\_aff} \\ \textit{seq\_aff} &\rightarrow \varepsilon \end{aligned}$$

```
rec_seq_aff =
  selon LC().nature
     cas IDF: rec_aff ; rec_seq_aff
     sinon : // rien, epsilon
```

---

$$\textit{aff} \rightarrow \texttt{IDF AFF}\ \textit{eag}\ \texttt{SEPINST}$$

```
rec_aff =
  A : A Ast ;
  si LC().nature = IDF alors avancer sinon Erreur() ;
  si LC().nature = AFF alors avancer sinon Erreur() ;
  rec_eag(A) ;  // A contient l'AST de l'expression
  si LC().nature = SEPINST alors avancer sinon Erreur() ;
```

# Analyse Syntaxique : les 2 nouvelles procédures

$$\textit{seq\_aff} \rightarrow \textit{aff}\ \textit{seq\_aff}$$
$$\textit{seq\_aff} \rightarrow \varepsilon$$

```
rec_seq_aff =
  selon LC().nature
     cas IDF: rec_aff ; rec_seq_aff
     sinon : // rien, epsilon
```

---

$$\textit{aff} \rightarrow \texttt{IDF AFF}\ \textit{eag}\ \texttt{SEPINST}$$

```
rec_aff =
  A : A Ast ;
  si LC().nature = IDF alors avancer sinon Erreur() ;
  si LC().nature = AFF alors avancer sinon Erreur() ;
  rec_eag(A) ;  // A contient l'AST de l'expression
  si LC().nature = SEPINST alors avancer sinon Erreur() ;
```

# Analyse Syntaxique : `rec_facteur` modifié

$$
\begin{aligned}
\textit{facteur} &\rightarrow \textit{ENTIER} \\
\textit{facteur} &\rightarrow \texttt{PARO}\ \textit{eag}\ \texttt{PARF} \\
\textit{facteur} &\rightarrow \texttt{IDF}
\end{aligned}
$$

```
Rec_facteur() =
   selon LC().nature  // LC est le lexeme_courant()
        cas ENTIER : Avancer()
        cas PARO : Avancer() ; Rec_eag() ;
                   si LC.nature = PARF alors Avancer sinon Erreur
        cas IDF : Avancer()
        autre : Erreur
fin
```

# Evaluation ?

```
X := 12 * 3 ;
Y := X + 5 ;
X := X + Y ;
```

1. calculer `12 * 3` (c'est une EAG !)
2. mémoriser le résultat (36) dans la variable `X`
3. calculer `X + 5` (presque une EAG ?)
   $\rightarrow$ utiliser la valeur 36 pour `X`
4. mémoriser le résultat (41) dans la variable `Y`
5. calculer `X + Y`
   (en utilisant les valeurs 36 et 41 pour `X` et `Y`)
6. mémoriser le résultat (77) dans la variable `X`

# Evaluation ?

```
X := 12 * 3 ;
Y := X + 5 ;
X := X + Y ;
```

1. calculer `12 * 3` (c'est une EAG !)
2. mémoriser le résultat (36) dans la variable `X`
3. calculer `X + 5` (presque une EAG ?)
   $\rightarrow$ utiliser la valeur 36 pour `X`
4. mémoriser le résultat (41) dans la variable `Y`
5. calculer `X + Y`
   (en utilisant les valeurs 36 et 41 pour `X` et `Y`)
6. mémoriser le résultat (77) dans la variable `X`

# Evaluation ?

```
X := 12 * 3 ;
Y := X + 5 ;
X := X + Y ;
```

1. calculer `12 * 3` (c'est une EAG !)
2. mémoriser le résultat (36) dans la variable `X`
3. calculer `X + 5` (presque une EAG ?)
   $\rightarrow$ utiliser la valeur 36 pour `X`
4. mémoriser le résultat (41) dans la variable `Y`
5. calculer `X + Y`
   (en utilisant les valeurs 36 et 41 pour `X` et `Y`)
6. mémoriser le résultat (77) dans la variable `X`

# Evaluation ?

```
X := 12 * 3 ;
Y := X + 5 ;
X := X + Y ;
```

1. calculer `12 * 3` (c'est une EAG !)
2. mémoriser le résultat (36) dans la variable `X`
3. calculer `X + 5` (presque une EAG ?)
   $\rightarrow$ utiliser la valeur 36 pour `X`
4. mémoriser le résultat (41) dans la variable `Y`
5. calculer `X + Y`
   (en utilisant les valeurs 36 et 41 pour `X` et `Y`)
6. mémoriser le résultat (77) dans la variable `X`

# Evaluation ?

```
X := 12 * 3 ;
Y := X + 5 ;
X := X + Y ;
```

1. calculer `12 * 3` (c'est une EAG !)
2. mémoriser le résultat (36) dans la variable `X`
3. calculer `X + 5` (presque une EAG ?)
   $\rightarrow$ utiliser la valeur 36 pour `X`
4. mémoriser le résultat (41) dans la variable `Y`
5. calculer `X + Y`
   (en utilisant les valeurs 36 et 41 pour `X` et `Y`)
6. mémoriser le résultat (77) dans la variable `X`

# Evaluation ?

```
X := 12 * 3 ;
Y := X + 5 ;
X := X + Y ;
```

1. calculer `12 * 3` (c'est une EAG !)
2. mémoriser le résultat (36) dans la variable `X`
3. calculer `X + 5` (presque une EAG ?)
   $\rightarrow$ utiliser la valeur 36 pour `X`
4. mémoriser le résultat (41) dans la variable `Y`
5. calculer `X + Y`
   (en utilisant les valeurs 36 et 41 pour `X` et `Y`)
6. mémoriser le résultat (77) dans la variable `X`

# Evaluation ?

```
X := 12 * 3 ;
Y := X + 5 ;
X := X + Y ;
```

1. calculer `12 * 3` (c'est une EAG !)
2. mémoriser le résultat (36) dans la variable `X`
3. calculer `X + 5` (presque une EAG ?)
   $\rightarrow$ utiliser la valeur 36 pour `X`
4. mémoriser le résultat (41) dans la variable `Y`
5. calculer `X + Y`
   (en utilisant les valeurs 36 et 41 pour `X` et `Y`)
6. mémoriser le résultat (77) dans la variable `X`

# Memoriser `IDF` $\leftrightarrow$ Valeur

Structure de données de "Table des Symboles" (TS)

## Interface (`table_symbole.h`)

- initialiser une table vide
- insérer/remplacer un couple (IDF, valeur)
- rechercher la valeur de IDF
- afficher la table

## Implémentation (`table_symbole.c`)

tableau, liste chainée, etc.

# Memoriser `IDF` $\leftrightarrow$ Valeur

Structure de données de "Table des Symboles" (TS)

## Interface (`table_symbole.h`)

- initialiser une table vide
- insérer/remplacer un couple (IDF, valeur)
- rechercher la valeur de IDF
- afficher la table

## Implémentation (`table_symbole.c`)

tableau, liste chainée, etc.

# Interpréter une séquence d'affectation ?

Lors de l'analyse syntaxique :

- fonction `rec_AFF()` :

$$Aff \rightarrow \texttt{IDF AFF } Eag \texttt{ SEPINST}$$

  - `Rec_Eag` calcule la valeur $v$ de la partie droite
  - insérer/remplacer (IDF,$v$) dans la table des symboles

- fonction `Rec_Facteur` :

$$Facteur \rightarrow \texttt{IDF}$$

  rechercher la valeur de IDF dans la table des symboles

# Nouvelle version de rec_aff

$$aff \rightarrow \texttt{IDF AFF } eag \texttt{ SEPINST}$$

```
rec_aff =
      A : A Ast ;
      v : entier ;  // valeur de l'IDF
      idf : chaine de caractères  // nom de l'IDF
  si LC().nature = IDF alors
         idf = LC().chaine ;
         avancer() ;
  sinon Erreur() ;
  si LC().nature = AFF alors avancer sinon Erreur() ;
      rec_eag(A) ;  // A contient l'AST de l'expression
      v = evaluer(A) ; // v contient la valeur de l'expression
      inserer(idf,v) ; // insere/remplace dans la TS
  si LC().nature = SEPINST alors avancer sinon Erreur() ;
```

# Nouvelle version de rec_facteur

$$\begin{aligned} \textit{facteur} &\rightarrow \textit{ENTIER} \\ \textit{facteur} &\rightarrow \texttt{PARO}\ \textit{eag}\ \texttt{PARF} \\ \textit{facteur} &\rightarrow \texttt{IDF} \end{aligned}$$

```
Rec_facteur(resultat : Ast A) =
   v : entier ;
   trouvé : booleen
   selon LC().nature  // LC est le lexeme_courant()
        cas ENTIER : A = creer_feuille(LC().valeur) ; Avancer()
        cas PARO : avancer() ; Rec_eag(A) ;
                   si LC.nature = PARF alors Avancer sinon Erreur
        cas IDF :
                  // recherche de la valeur v dans la TS
                  trouvé = chercher(LC().chaine, v) ;
                  si trouvé alors
                         A = creer_feuille(v) ;
                  sinon Erreur() ;
                  avancer() ;
        autre : Erreur()
fin
```

# Important pour la suite . . . !

## la semaine prochaine en TP

**fin calculette ! ! !**

ou **début du projet ...** (TP5)

## Sans oublier :

Chaque **binôme** dépose sur Moodle :

- les sources de la calculette (achevée ou non)
- servira d'**inscription** pour le projet !