UFR IM²AG

UNIVERSITÉ Grenoble Alpes

UE INF404 - Projet Logiciel

Projet

Séquence d'instructions et entrées/sorties

L2 Informatique

Année 2023 - 2024

Au menu

Au menu

Objectifs du projet

Ecrire un interpréteur

1. choisir ce que l'on veut interpréter ...
2. définir le langage d'entrée
   alphabet, lexique, syntaxe, sémantique
3. écrire les fonctions d'analyse (lexicale et syntaxique)
4. définir et produire l'Ast
5. écrire le "traitement" de l'Ast

$\Rightarrow$ même démarche que pour la calculette

(et réutilisation partielle possible de certains modules !)

Quelques pistes (réalistes !) possibles ...

- calculatrice étendue ...
- exécution/interprétation
  langage de programmation (*), assembleur ARM, ...
- traduction
  langage L $\rightarrow$ Python, langage L $\rightarrow$ HTML, ...
- etc. ...

Attention : soutenances/démo la semaine du 26 avril

- ne pas être trop ambitieux ...
- progresser de manière incrémentale
- privilégier des solutions simples

(*) détaillé dans les sujets des TPs

Mise en oeuvre en TP

1. calculette (L0)

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

2. affectations (L1) [TP5]

```
X = 2 ;
Y = X + 2 ;
```

3. entrées-sorties (L1+) [TP6]

```
lire(X) ;
ecrire (X+12) ;
```

4. instructions conditionnelles (L2) [TP7]

```
if X>0 then Y = X + 2 else Y = 3 ; fi
```

5. instruction itérative (L3) [TP8]

```
while X<10 do X = X * 2 ; od ;
```

Mise en oeuvre en TP

1. calculette (L0)

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

2. affectations (L1) [TP5]

```
X = 2 ;
Y = X + 2 ;
```

3. entrées-sorties (L1+) [TP6]

```
lire(X) ;
ecrire (X+12) ;
```

4. instructions conditionnelles (L2) [TP7]

```
if X>0 then Y = X + 2 else Y = 3 ; fi
```

5. instruction itérative (L3) [TP8]

```
while X<10 do X = X * 2 ; od ;
```

Mise en oeuvre en TP

1. calculette (L0)

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

2. affectations (L1) [TP5]

```
X = 2 ;
Y = X + 2 ;
```

3. entrées-sorties (L1+) [TP6]

```
lire(X) ;
ecrire (X+12) ;
```

4. instructions conditionnelles (L2) [TP7]

```
if X>0 then Y = X + 2 else Y = 3 ; fi
```

5. instruction itérative (L3) [TP8]

```
while X<10 do X = X * 2 ; od ;
```

Mise en oeuvre en TP

1. calculette (L0)

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

2. affectations (L1) [TP5]

```
X = 2 ;
Y = X + 2 ;
```

3. entrées-sorties (L1+) [TP6]

```
lire(X) ;
ecrire (X+12) ;
```

4. instructions conditionnelles (L2) [TP7]

```
if X>0 then Y = X + 2 else Y = 3 ; fi
```

5. instruction itérative (L3) [TP8]

```
while X<10 do X = X * 2 ; od ;
```

Mise en oeuvre en TP

1. calculette (L0)

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

2. affectations (L1) [TP5]

```
X = 2 ;
Y = X + 2 ;
```

3. entrées-sorties (L1+) [TP6]

```
lire(X) ;
ecrire (X+12) ;
```

4. instructions conditionnelles (L2) [TP7]

```
if X>0 then Y = X + 2 else Y = 3 ; fi
```

5. instruction itérative (L3) [TP8]

```
while X<10 do X = X * 2 ; od ;
```

Au menu

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

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

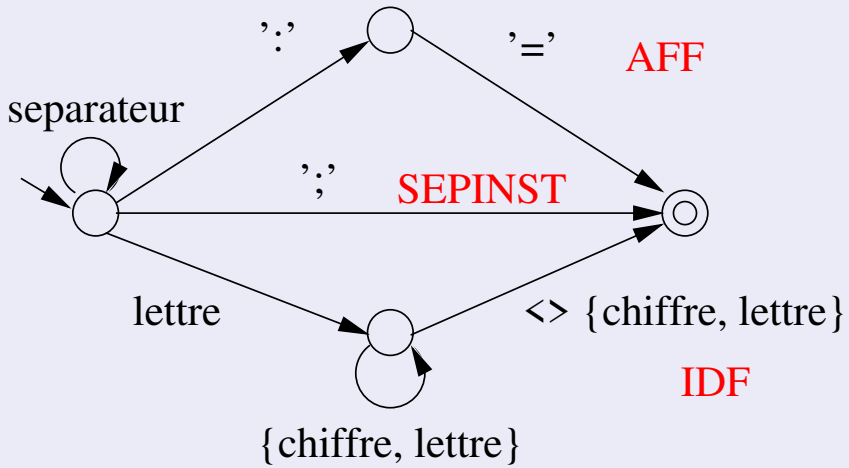
et ajouter $\textit{facteur} \rightarrow \texttt{IDF}$

Analyse Lexicale

Nouveaux lexèmes possibles :

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

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



Version plus simple :

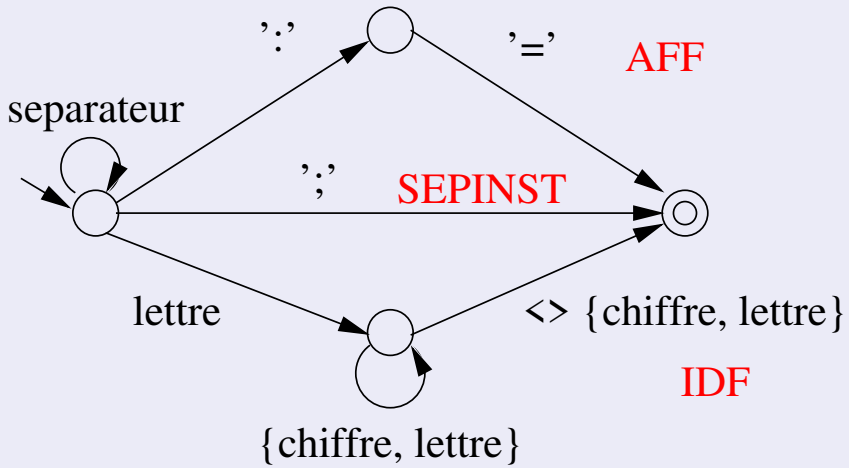
- affectation sur `un seul` caractère : '='
- IDF sur `une seule lettre` entre 'a' et 'z'

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	→	*aff seq_aff*
seq_aff	→	ε
aff	→	IDF AFF *eag* SEPINST
eag	→	*seq_terme*
...	→	...
facteur	→	*ENTIER*
facteur	→	PARO *eag* PARF
facteur	→	IDF
*op*1	→	PLUS
*op*1	→	MOINS
*op*2	→	MUL

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

Et modifier `rec_facteur` ...

Interprétation ?

```
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`

Interprétation ?

```
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`

Interprétation ?

```
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`

Interprétation ?

```
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`

Interprétation ?

```
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`

Interprétation ?

```
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`

Interprétation ?

```
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()` :

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

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

- fonction `Rec_Facteur` :

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

  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
```

Au menu

Instructions d'entrées-sorties

- lire des entrées au clavier → `lire`
- afficher des sorties à l'écran → `ecrire`

Exemple

```
lire(X) ;
Y := X+1 ;
ecrire(Y*(X+3)) ;
```

Interprétation

`lire (X)` :

1. appel à la fonction de lecture clavier (`scanf()`)
2. insertion de la valeur lue dans la table des symboles . . .

`ecrire (Y*(X+3))` :

1. évaluer l'expression `Y * (X+3)`
2. affichage du résultat à l'écran (`printf()`)

Analyse Lexicale

Introdution de mots-clés

`lire`, `ecrire`, `...si`, `alors`, `sinon`, `...`

Distinguer mots-clés et noms de variables (idf) ?

solution 1 : mots-clés en minuscules et IDF en majuscules ...
solution 2 : IDF = toute suite de lettres-chiffres qui n'est pas un mot-clé

1. on reconnait un lexéme de la forme suite de lettres-chiffres
2. on cherche s'il appartient à une liste finie (!) de mot-clés
3. si on le trouve, c'est un mot-clé, sinon c'est un IDF . . . !

Reconnaissance des mots-clés (détail)

Dans la procédure `Reconnaitre_Lexeme` :

1. ajouter les lexèmes `LIRE`, `ECRIRE` au type `Nature_Lexeme`
2. déclarer un tableau de 2 mot-clés (de 20 caractères max)

```
#define NB_MOTCLE 2
char motCle[2][20] = {"lire", "ecrire"}
```

3. une suite de lettres est considérée (a priori) comme un IDF ...
4. vérifier alors si cet IDF est ou non un mot-clé

```
for (i=0 ; i<NB_MOTCLE ; i++)
    if (strcmp(lexeme_en_cours.chaine, motCle[i]) == 0) {
        switch(i) {
            case 0: lexeme_en_cours.nature = LIRE; break ;
            case 1: lexeme_en_cours.nature = ECRIRE; break ;
                        ...
            default: break ;
        }
    }
```

Cette solution est facile à étendre par ajout de nouveaux mots-clés ...

Syntaxe d'un **Programme**

On étend la grammaire : un **programme** est une séquence d'**instructions** (`seq_inst`), où chaque instruction est soit une affectation soit une autre instruction (`lire`, etc.).

Programme = séquence d'Instructions

$$
\begin{aligned}
\text{pgm} &\rightarrow \text{seq\_inst} \\
\text{seq\_inst} &\rightarrow \text{inst suite\_seq\_inst} \\
\text{suite\_seq\_inst} &\rightarrow \texttt{SEPINST seq\_inst} \\
\text{seq\_inst} &\rightarrow \varepsilon \\
\text{inst} &\rightarrow \texttt{IDF AFF eag} \\
\text{inst} &\rightarrow \texttt{LIRE PARO IDF PARF} \\
\text{inst} &\rightarrow \texttt{ECRIRE PARO eag PARF} \\
\text{inst} &\rightarrow \text{autres formes d'instructions } \ldots
\end{aligned}
$$

Construction d'un Arbre Abstrait (AsT)

L'ajout des instructions conditionnelles `if` et itératives `while` nécessitera de construire un AST "complet" du programme ...

Intérêt :
une seule lecture du fichier $\Rightarrow$ plusieurs traitements possibles

- analyse lexicale et syntaxique complète du fichier
- interprétation = parcours de l'AST
- autres applications possibles :
  - verification des types
  - génération de code assembleur
  - etc.

  $\rightarrow$ par parcours de l'AST ...

Structure de l'Arbre Abstrait ?

```
lire (X) ;
Y := X+1 ;
ecrire (Y * 2) ;
```

3 types d'instructions sur cet exemple :

1. instruction d'affectation (`X := 1`)
2. instruction de lecture (`lire (X)`
3. instruction d'écriture (`ecrire (Y * 2)`

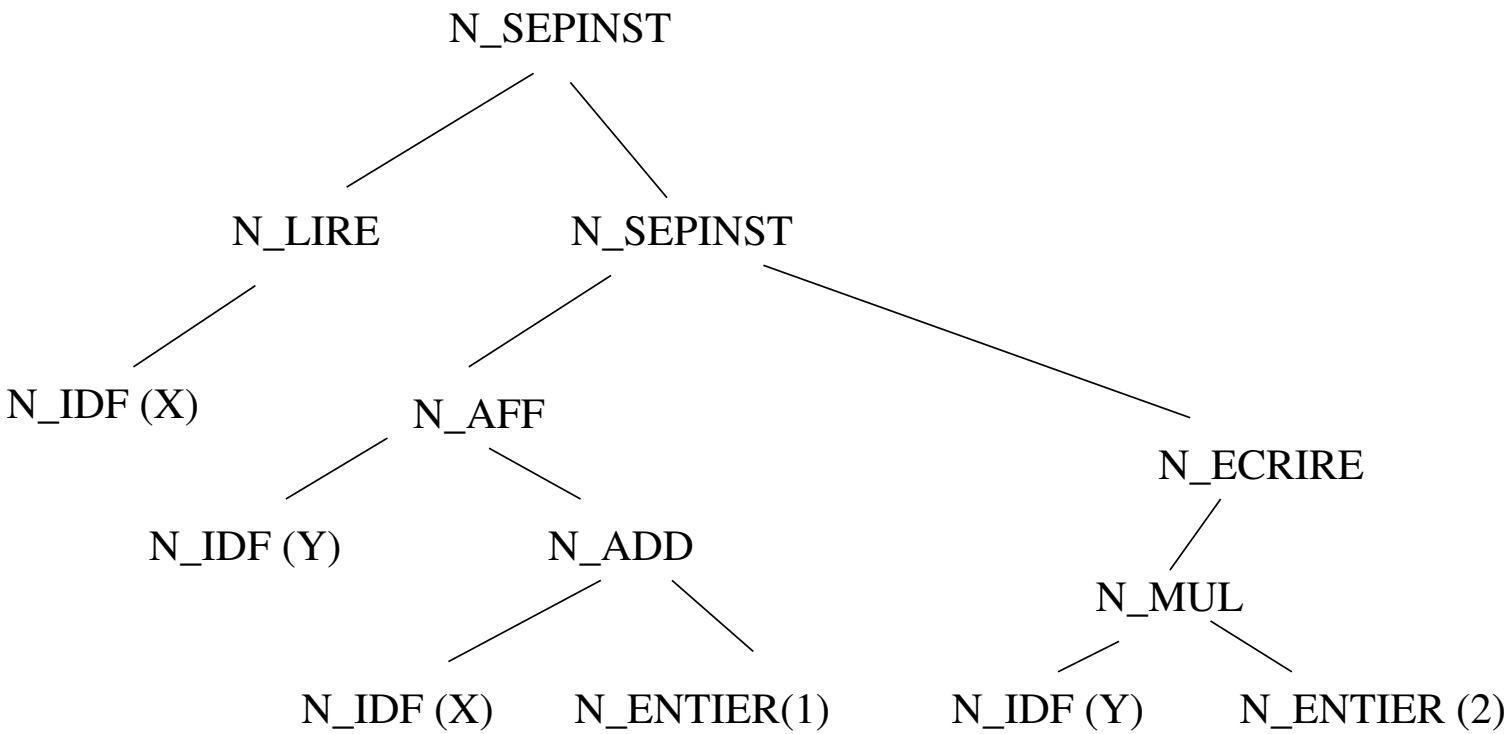
$\Rightarrow$ 4 (nouveaux) types de noeuds dans l'arbre abstrait :

- `N_SEPINST`, séparateur d'instructions (avec 2 fils)
- `N_AFF`, instruction d'affectation (avec 2 fils)
- `N_LIRE`, instruction de lecture (avec 1 seul fils)
- `N_ECRIRE`, instruction d'écriture (avec 1 seul fils)

. . .et 4 nouvelles fonctions de construction !

Arbre Abstrait de l'exemple précédent

```
lire (X) ;
Y := X+1 ;
ecrire (Y * 2) ;
```



Construction de l'Arbre Abstrait (1)

Etendre l'analyse syntaxique et les modules Ast en ajouter les nouveaux types de noeuds et les procédures de construction associées.

```
Rec_seq_inst (A : resultat Ast) =
  A1 : Ast ;
  Rec_inst (A1)
   // produit l'Ast A1 de l'instruction lue
  Rec_suite_seq_inst (A1, A)
   // produit l'Ast A de la sequence d'instructions lues

Rec_suite_seq_inst (A1 : donné Ast; A : resultat Ast) =
  Ast A2 ;
  selon LC.nature
    cas N_SEPINST :
          Rec_seq_inst (A2) ;
          A := creer_seqinst(A1, A2)
   // cree un noeud N_SEPINST de fils gauche A1
   //        et de fils droit A2
   sinon :  // epsilon
         A := A1
```

Construction de l'Arbre Abstrait (2)

$$\begin{aligned} \text{inst} &\rightarrow \texttt{IDF AFF } \textit{eag} \\ \text{inst} &\rightarrow \texttt{LIRE PARO IDF PARF} \\ \text{inst} &\rightarrow \texttt{ECRIRE PARO eag PARF} \end{aligned}$$

```
Rec_inst (A : resultat Ast)  =
  Ag, Ad : Ast ;
  selon LC().nature
     cas IDF : // affectation
                // cree un arbre gauche qui contient l'IDF
                Ag = creer_idf (LC().chaine)
        avancer()
        si LC().nature = AFF alors avancer() sinon Erreur() ;
        rec_eag(Ad) ;  // Ad contient l'AST de l'expression
                // cree un noeud N_AFF de fils Ag et Ad
                A = creer_aff (Ag, Ad)
     cas LIRE : // transparent suivant
     cas ECRIRE : // transparent suivant
     sinon : Erreur()
```

Construction de l'Arbre Abstrait (3)

inst → IDF AFF *eag*

inst → LIRE PARO IDF PARF

inst → ECRIRE PARO eag PARF

```
Rec_inst (A : resultat Ast)  =
  Ag : Ast ;
  selon LC().nature
     cas IDF : // transparent précédent
     cas LIRE :
          avancer() ;
          si LC().nature = PARO alors avancer() sinon Erreur() ;
          si LC().nature = IDF alors
              Ag = creer_idf (LC().chaine) ;  avancer()
          sinon Erreur() ;
                // cree un noeud N_LIRE de fils gauche Ag
                  A = creer_lire (Ag)
          si LC().nature = PARF alors avancer() sinon Erreur() ;
     cas ECRIRE : // transparent suivant
     sinon : Erreur()
```

Construction de l'Arbre Abstrait (4)

$$\begin{aligned} \text{inst} &\rightarrow \texttt{IDF AFF } \textit{eag} \\ \text{inst} &\rightarrow \texttt{LIRE PARO IDF PARF} \\ \text{inst} &\rightarrow \texttt{ECRIRE PARO eag PARF} \end{aligned}$$

```
Rec_inst (A : resultat Ast)  =
  Ag : Ast ;
  selon LC().nature
     cas IDF : // transparent précédent
     cas LIRE : // transparent précédent
     cas ECRIRE :
          avancer() ;
          si LC().nature = PARO alors avancer() sinon Erreur() ;
          rec_eag(Ag)
                // cree un noeud N_ECRIRE de fils gauche Ag
                  A = creer_ecrire (Ag)
          si LC().nature = PARF alors avancer() sinon Erreur() ;
     sinon : Erreur()
```

Interprétation du programme complet

Parcours (récursif) de l'Ast du programme et traitement par cas :

```
interpreter (A : Ast)
  selon A.nature
     cas N_SEPINST :
                  interpreter_aff(A.fils_gauche) ;
                  interpreter_aff(A.fils_droit) ;
     cas N_AFF : interpreter_aff(A) ;
                  // transparent suivant
     cas N_LIRE : interpreter_lire(A) ;
                  // transparent suivant
     cas N_ECRIRE : interpreter_ecrire (A) ;
                  // transparent suivant
     etc.
     sinon : Erreur() ;
```

Interprétation des instructions d'affectation

```
interpreter_aff (A : Ast)
    idf : chaine de caractères  // nom de l'IDF
    v : entier ;  // valeur de l'IDF
          // on récupère le nom de l'IDF à affecter
    idf = A.fils_gauche.chaine
          // on récupère la valeur de l'expression
    v = evaluer(A.fils_droit) ;
          // on insere/remplace ce couple dans la TS
    inserer(idf, v) ;
```

Interprétation des instructions de lecture/écriture

```
interpreter_lire (A : Ast)
    v : entier ;
    // lecture d'un entier au clavier (scanf)
    lire(v) ;
    // insere/remplace dans la TS
    inserer (A.fils_gauche.chaine, v) ;

interpreter_ecrire (A : Ast)
    v : entier ;
    // calcul de l'eag à afficher
    v = evaluer(A.fils_gauche) ;
    // affichage de v à l'écran (printf)
    ecrire (v) ;
```

Dans la suite . . .

- commencer (ou continuer) le TP5
  en vous aidant *si nécessaire* du corrigé de la calculette . . .
- poursuivre avec le TP6 . . .
- réfléchir à l'ajout d'une instruction `if-then-else` ?