UFR IM²AG

UNIVERSITÉ Grenoble Alpes

UE INF404 - Projet Logiciel

Projet “Interpréteur”

Instruction Conditionnelle

L2 Informatique

Année 2023 - 2024

Au menu

1. Point sur le projet
2. Rapppel et compléments sur l'arbre abstrait
3. Langage L2 : instruction conditionnelle

Au menu

1. Point sur le projet
2. Rapppel et compléments sur l'arbre abstrait
3. Langage L2 : instruction conditionnelle

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 !)

Interpréteur pour un “petit” langage de programmation

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 ;
```

Interpréteur pour un “petit” langage de programmation

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 ;
```

Interpréteur pour un “petit” langage de programmation

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 ;
```

Interpréteur pour un “petit” langage de programmation

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 ;
```

Interpréteur pour un "petit" langage de programmation

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

1. Point sur le projet
2. Rapppel et compléments sur l'arbre abstrait
3. Langage L2 : instruction conditionnelle

Langage source

- séquence d'instructions
- différentes instructions :
  - affectations
  - lectures/écritures
  - instructions conditionnelles
  - itérations
  - etc.

Exemple

```
lire (X) ;
Y := 1 ;
tanque X > 0 faire
     Y = Y * X ;
     X = X -1 ;
fait ;
ecrire (Y) ;
```

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[NB_MOTCLE][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

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

Construction d'un Arbre Abstrait (AsT)

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 !

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)`)

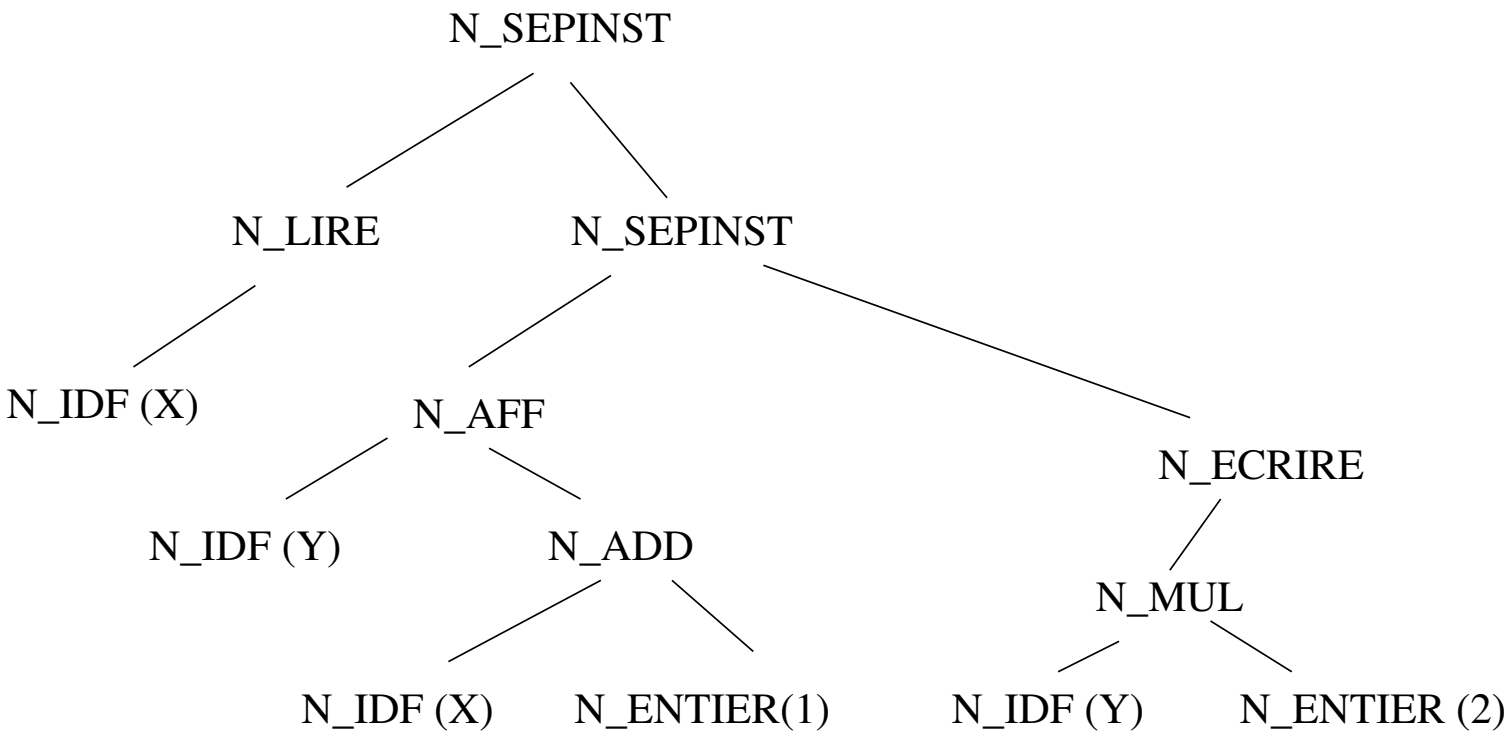
$\Rightarrow$ 4 (nouveaux) types de noeuds dans l'arbre abstrait :

- `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 SEPINST :
      avancer() ;  // on lit le 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, Ad : 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)

inst → IDF AFF *eag* SEPINST

inst → LIRE PARO IDF PARF

inst → ECRIRE PARO eag PARF

```
Rec_inst (A : resultat Ast)  =
  Ag, Ad : 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(A.fils_gauche) ;
              interpreter(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() ;

programme principal
    A : Ast ;
    analyser (fichier, A) ; // A contient l'Ast du programme lu
    interpreter(A) ;
    ecrire_TS() ; // on affiche me contenu de la table des symboles
```

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
    lire(v) ;  // scanf("%d", &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 a l'ecran
    ecrire (v) ; // printf ("%d\n", v)
```

Interprétation des instructions de lecture/écriture

```
interpreter_lire (A : Ast)
    v : entier ;
    // lecture d'un entier au clavier
    lire(v) ;  // scanf("%d", &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 a l'ecran
    ecrire (v) ; // printf ("%d\n", v)
```

Au menu

1. Point sur le projet
2. Rapppel et compléments sur l'arbre abstrait
3. Langage L2 : instruction conditionnelle

Définir le langage ?

Exemple

```
X := 12 * 3 ;
Y := X + 5 ;
Z := 42 ;
si (Y > X) alors
   Z := X-1 ;
sinon
   Y := X ;
   si (Y != Z) alors
        Z := X + 2 ;
   fsi
fsi
X := Y * 2 + Z
```

Que doit-on modifier par rapport à L1+ ?

Analyse Lexicale

Nouveaux opérateurs ?

- opérateurs de comparaison
  `OPCOMP` $= \{$ `'='`, `'>'`, `'<'`, `'!='`, `'<='`, ... $\}$
- opérateurs booléens
  `OPBOOL` $= \{$ et, ou , non $\}$

Nouveaux mots-clés

`si`, `alors`, `sinon`, `fsi`

Analyse Lexicale

Nouveaux opérateurs ?

- opérateurs de comparaison
  `OPCOMP` $= \{$ `'='`, `'>'`, `'<'`, `'!='`, `'<='`, ... $\}$
- opérateurs booléens
  `OPBOOL` $= \{$ et, ou , non $\}$

Nouveaux mots-clés

`si`, `alors`, `sinon`, `fsi`

Reconnaissance des mots-clés (détail)

Dans la procédure `Reconnaitre_Lexeme` :

1. ajouter les lexèmes `SI`, `ALORS`, `SINON`, `FSI` au type `Nature_Lexeme`
2. déclarer un tableau de 6 mot-clés (de 20 caractères max)

```
#define NB_MOTCLE 6
char motCle[NB_MOTCLE][20] = {"lire", "ecrire",
                              "si", "alors", "sinon", "fsi"}
```

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 ;
            case 2: lexeme_en_cours.nature = SI; break ;
                            ...
            default: break ;
        }
    }
```

Analyse Syntaxique (1)

On étend la grammaire : on ajoute l'instruction conditionnelle ...

Programme = séquence d'Instructions

$$
\begin{aligned}
\text{pgm} &\rightarrow \text{seq\_inst} \\
\text{seq\_inst} &\rightarrow \text{inst suite\_seq\_inst} \\
\text{suite\_seq\_inst} &\rightarrow \texttt{SEPINST seq\_inst} \\
\text{suite\_seq\_inst} &\rightarrow \varepsilon \\
\text{inst} &\rightarrow \texttt{IDF AFF eag} \\
\text{inst} &\rightarrow \texttt{SI condition ALORS seq\_inst SINON seq\_inst FSI} \\
\text{inst} &\rightarrow \text{autres formes d'instructions ...}
\end{aligned}
$$

Analyse Syntaxique (2)

Syntaxe d'une Instruction Conditionnelle

inst → `SI` condition `ALORS` seq_inst `SINON` seq_inst `FSI`

Syntaxe d'une Condition ?

Plusieurs choix possibles :

1. expression booléenne "générale"
   `(X < 2 + Z) et (Y >= 42) ou (X != Z)`
   ↪ grammaire complexe (eag avec opérateurs booléens) /
2. expression booléenne "simple"
   `X < 3, Y >= 42`
   comparaison entre 2 opérandes entiers (pas d'opérateurs booléens)
   `OPCOMP` = {`==, !=, >=, <=, etc.`}
   ↪ grammaire plus simple **(faire ce choix pour commencer !)**

condition → eag `OPCOMP` eag

Analyse Syntaxique (2)

Syntaxe d'une Instruction Conditionnelle

inst → `SI` condition `ALORS` seq_inst `SINON` seq_inst `FSI`

Syntaxe d'une Condition ?

Plusieurs choix possibles :

1. expression booléenne "générale"
   `(X < 2 + Z) et (Y >= 42) ou (X != Z)`
   ↪ grammaire complexe (eag avec opérateurs booléens) /
2. expression booléenne "simple"
   `X < 3, Y >= 42`
   comparaison entre 2 opérandes entiers (pas d'opérateurs booléens)
   `OPCOMP` = {`==, !=, >=, <=, etc.`}
   ↪ grammaire plus simple **(faire ce choix pour commencer !)**

condition → eag `OPCOMP` eag

Analyse Syntaxique (2)

Syntaxe d'une Instruction Conditionnelle

inst → `SI` condition `ALORS` seq_inst `SINON` seq_inst `FSI`

Syntaxe d'une Condition ?

Plusieurs choix possibles :

1. expression booléenne "générale"
   `(X < 2 + Z) et (Y >= 42) ou (X != Z)`
   ↪ grammaire complexe (eag avec opérateurs booléens) /
2. expression booléenne "simple"
   `X < 3, Y >= 42`
   comparaison entre 2 opérandes entiers (pas d'opérateurs booléens)
   `OPCOMP` = {`==, !=, >=, <=, etc.` }
   ↪ grammaire plus simple **(faire ce choix pour commencer !)**

condition → eag `OPCOMP` eag

Structure de l'Arbre Abstrait ?

```
r := 1 ;
si  x > 1 alors
   r := r*x ;
   x := x-1 ;
sinon
   x := 2 ;
fsi
```

Deux instructions sur cet exemple :

1. une instruction d'affectation (`r := 1`)
2. une instruction conditionnelle :
   - une condition (`x > 1`)
   - une branche "then" avec 2 affectations
   - une branche "else" avec 1 affectations

$\Rightarrow$ (encore) un nouveau type de noeud dans l'arbre abstrait :

- `N_IF`, instruction conditionnelle (avec 3 fils)

Structure de l'Arbre Abstrait ?

```
r := 1 ;
si  x > 1 alors
   r := r*x ;
   x := x-1 ;
sinon
   x := 2 ;
fsi
```

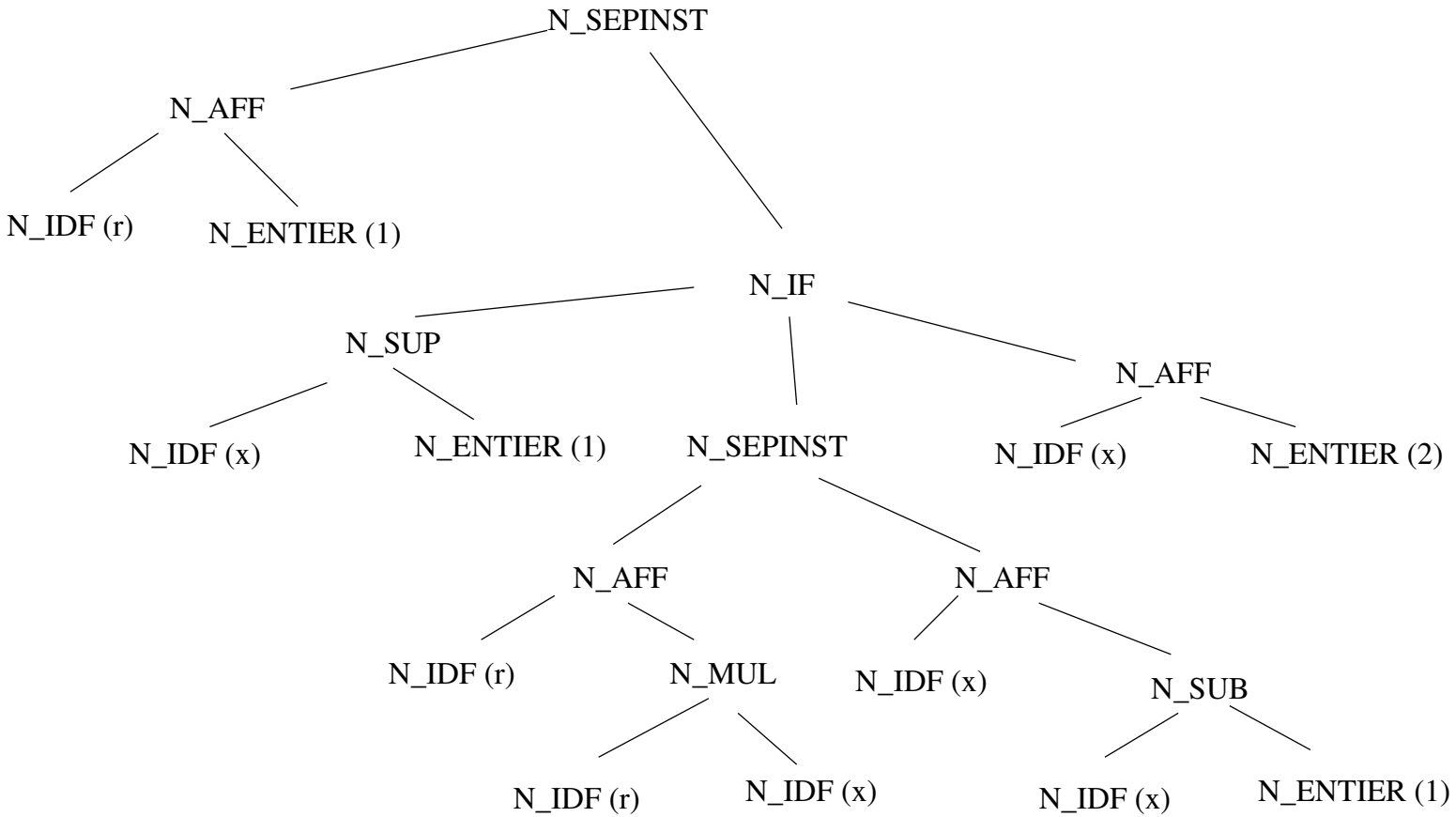
Deux instructions sur cet exemple :

1. une instruction d'affectation (`r := 1`)
2. une instruction conditionnelle :
   - une condition (`x > 1`)
   - une branche "then" avec 2 affectations
   - une branche "else" avec 1 affectations

$\Rightarrow$ (encore) un nouveau type de noeud dans l'arbre abstrait :

- `N_IF`, instruction conditionnelle (avec 3 fils)

Arbre Abstrait de l'exemple précédent

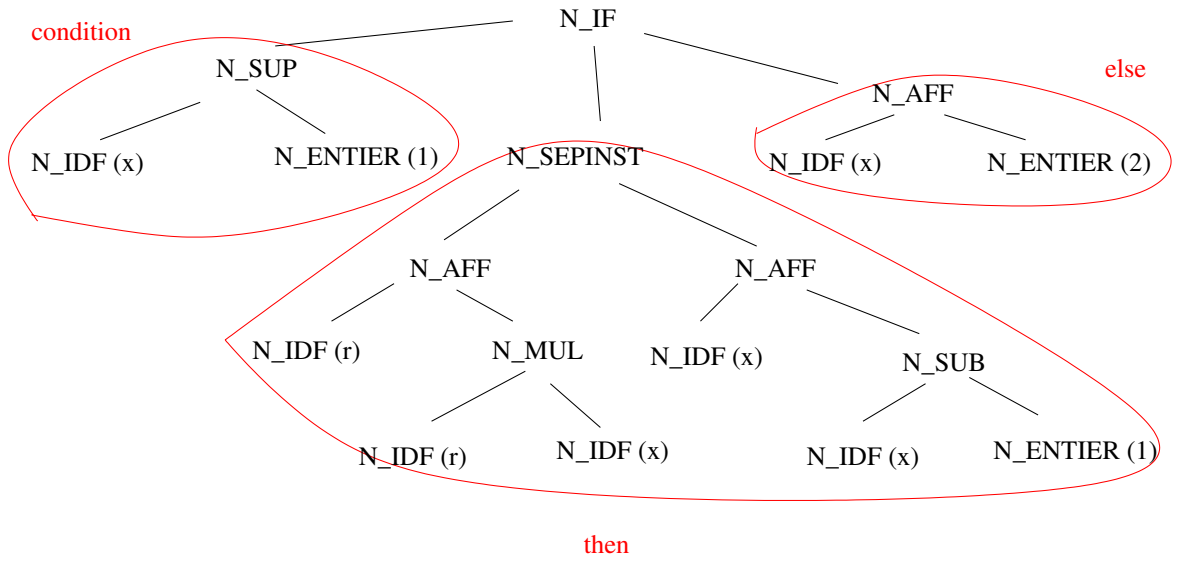


Construction de l'Arbre Abstrait

inst → SI condition ALORS seq_inst SINON seq_inst FSI

```
Rec_inst (A : resultat Ast)  =
   Ast Acond, Athen, Aelse ;
   selon (LC.nature)
        cas AFF : // cree et renvoie un AST pour une affectation
        cas SI :
            avancer
            Rec_condition(Acond)
            si (LC.nature = ALORS) alors avancer sinon Erreur
            Rec_seq_inst(Athen)
            si (LC.nature = SINON) alors avancer sinon Erreur
            Rec_seq_inst(Aelse)
            si (LC.nature = FSI) alors avancer sinon Erreur
            A = creer_if(Acond, Athen, Aelse)
               // cree le noeud N_IF a partir de ses 3 fils
        cas LIRE : // cree et renvoie un AST pour une lecture
        etc.
```

Interprétation d'une instruction conditionnelle (1)



Il suffit de parcourir les fils du noeud `N_IF` :

1. le fils gauche pour évaluer la condition
2. si elle vaut "vrai" on interprète le fils central (branche "then")
3. si elle vaut "faux" on interprète le fils droit (branche "else")

Interprétation d'une instruction conditionnelle (2)

```
interpreter_si_alors_sinon (A : Ast)
    condition : booléen ; // valeur de la condition
    // on évalue la condition
    condition  =  valeur_booleenne(A.fils_gauche)
    si condition alors
       interpreter (A.fils_central) ;
    sinon
       interpreter (A.fils_droit) ;


valeur_booleenne (A : Ast)
    // évalue l'arbre abstrait d'une condition
    valeurg, valeurd : valeurs de fils gauche et droit
    valeurg = evaluer(A.fils_gauche) ;
    valeurd = evaluer(A.fils_droit) ;
    selon A.nature
        cas N_EGAL : return (valeurg = valeur_d)
        cas N_SUP : return (valeurg > valeur_d)
        etc.
```

Interprétation d'une instruction conditionnelle (2)

```
interpreter_si_alors_sinon (A : Ast)
    condition : booléen ; // valeur de la condition
    // on évalue la condition
    condition  =  valeur_booleenne(A.fils_gauche)
    si condition alors
       interpreter (A.fils_central) ;
    sinon
       interpreter (A.fils_droit) ;


valeur_booleenne (A : Ast)
    // évalue l'arbre abstrait d'une condition
    valeurg, valeurd : valeurs de fils gauche et droit
    valeurg = evaluer(A.fils_gauche) ;
    valeurd = evaluer(A.fils_droit) ;
    selon A.nature
        cas N_EGAL : return (valeurg = valeur_d)
        cas N_SUP : return (valeurg > valeur_d)
        etc.
```

Dans la suite . . .

- commencer (ou continuer) les TP5 et TP6
  en vous aidant *si nécessaire* du corrigé de la calculette . . .
- poursuivre avec le TP7
- réfléchir à l'ajout d'une instruction `tantque` ?