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 possibles . . .

- simulation
  robot, système physique, “jeu”, . . .
- exécution/interprétation
  langage de programmation, langage graphique, . . .
- évaluation calculatrice étendue
- traduction
- vérification de type
- etc. . . .

Des exemples concrets

- simulateur assembleur ARM
- exécution programme “Tortue Logo”
- traduction langage L $\rightarrow$ C
- simulateur langage L
- langage “graphique”
  composition de figures élémentaires
- traduction langage L $\rightarrow$ HTML
- etc. . . .

Langage L1 : séquences d'affectations

Exemple

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

Interpréteur pour L1

$\Rightarrow$ afficher les valeurs finales des variables

1. analyse lexicale : nouveaux lexèmes
2. analyse syntaxique : étendre la grammaire
3. interpréteur : ajouter une **table des symboles** (mémorise les valeurs des variables) $\rightarrow$ interprétation **ligne par ligne**

Au menu

Objectifs

Actuellement (TP5)

un programme = une séquence d'affectation

```
X := 5 ;
Y := X * 2 ;
```

Etape suivante (TP6)

Ajouter une nouvelle instruction `if-then-else`

1. définir le nouveau langage
2. étendre l'analyse (lexicale et sytaxique)
3. étendre l'interpréteur ...

Définir le langage ?

Exemple

```
X := 12 * 3 ;
Y := X + 5 ;
Z := 42 ;
if (Y > X) then
   Z := X-1 ;
else
   Y := X ;
   if (Y != Z) then
        Z := X + 2 ;
   fi
fi
X := Y * 2 + Z ;
```

Que doit-on modifier par rapport à L1 ?

Analyse Lexicale

Nouveaux opérateurs ?

- opérateurs de comparaison
  `OPCOMP` $= \{ <, \leq, =, \neq,$ etc. $\}$
- opérateurs booléens
  `OPBOOL` $= \{$ et, ou , non $\}$

Nouveaux mots-clés

`if`, `then`, `else`, `fi`

Distinguer mots-clés et noms de variables ?

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 cherche d'abord dans une liste finie (!) de mot-clés
2. si on trouve, c'est un mot-clé, sinon c'est un IDF . . . !

(voir le transparent suivant)

Analyse Lexicale

Nouveaux opérateurs ?

- opérateurs de comparaison
  `OPCOMP` $= \{ <, \leq, =, \neq, \text{etc.} \}$
- opérateurs booléens
  `OPBOOL` $= \{$ et, ou , non $\}$

Nouveaux mots-clés

`if`, `then`, `else`, `fi`

Distinguer mots-clés et noms de variables ?

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 cherche d'abord dans une liste finie (!) de mot-clés
2. si on trouve, c'est un mot-clé, sinon c'est un IDF . . . !

(voir le transparent suivant)

Reconnaissance des mots-clés (détail)

Dans la procédure `Reconnaitre_Lexeme` :

1. ajouter les lexèmes `IF`, `THEN`, `ELSE`, `FI` au type `Nature_Lexeme`
2. déclarer un tableau de 4 mot-clés (de 20 caractères max)

```
#define NB_MOTCLE 4
char motCle[4][20] = {"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 = IF; break ;
            case 1: lexeme_en_cours.nature = THEN; break ;
                        ...
            default: break ;
        }
    }
```

Analyse Syntaxique (1)

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

Programme = séquence d'Instructions

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

Analyse Syntaxique (2)

Syntaxe d'une Instruction Conditionnelle

inst $\rightarrow$ `IF` condition `THEN` seq_inst `ELSE` seq_inst `FI`

Syntaxe d'une Condition ?

Plusieurs choix possibles :

1. expression boolénne "générale"
   `(X < 2 + Z) et (Y >= 42) ou (X != Z)`
   $\hookrightarrow$ grammaire complexe (eag avec opérateurs booléens)
2. expression boolénne "simple"
   `X < 3, Y >= 42`
   comparaison entre 2 opérandes entiers (pas d'opérateurs booléens)
   $\hookrightarrow$ grammaire plus simple **(faire ce choix pour commencer !)**

condition $\rightarrow$ eag `OPCOMP` eag

Analyse Syntaxique (2)

Syntaxe d'une Instruction Conditionnelle

inst $\rightarrow$ `IF` condition `THEN` seq_inst `ELSE` seq_inst `FI`

Syntaxe d'une Condition ?

Plusieurs choix possibles :

1. expression boolénne "générale"
   `(X < 2 + Z) et (Y >= 42) ou (X != Z)`
   $\hookrightarrow$ grammaire complexe (eag avec opérateurs booléens)
2. expression boolénne "simple"
   `X < 3, Y >= 42`
   comparaison entre 2 opérandes entiers (pas d'opérateurs booléens)
   $\hookrightarrow$ grammaire plus simple **(faire ce choix pour commencer !)**

condition $\rightarrow$ eag `OPCOMP` eag

Interprétation (1)

Solution 1 : 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 ...

Interprétation (2)

Solution 2 : ne pas construire d'AST "complet"

interprétation instruction par instruction lors de l'analyse syntaxique

Avantage : moins de programmation ...

Inconvénient :

- peu extensible (ajout de nouveaux traitements ?)
- on n'analyse **que** les instructions **exécutées** ... ?

```
X := 12 ;
if X > 3 then
   Y := X + 2 ;
else
   Y := X # 2 ;  /* erreur non detectee ! */
fi
```

⇒ La solution 1 est préférable !

Structure de l'Arbre Abstrait ?

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

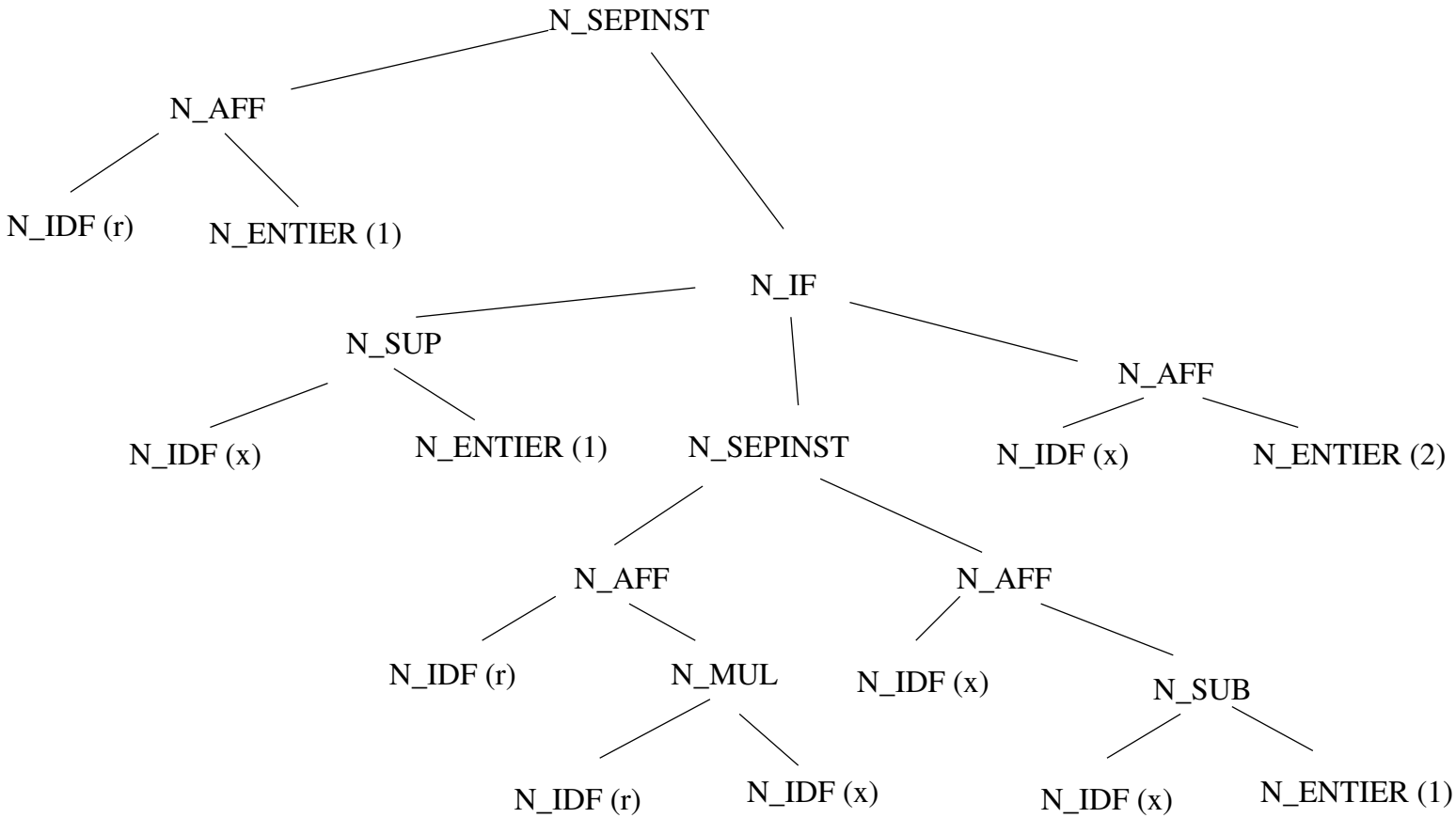
Deux instructions sur cet exemple :

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

⇒ Deux nouveaux types de noeuds dans l'arbre abstrait :

- `N_SEPINST`, séparateur d'instructions (avec 2 fils)
- `N_IF`, instruction conditionnelle (avec 3 fils)

Arbre Abstrait de l'exemple précédent



Construction de l'Arbre Abstrait (1)

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

```
Rec_seq_inst =
  A1 = Rec_inst
   // produit l'Ast A1 de l'instruction lue
  A2 = Rec_suite_seq_inst (A1)
   // produit l'Ast A2 de la sequence d'instructions lues
  return A2


Rec_suite_seq_inst (Ast A1) =
  Ast A2
  selon LC.nature
    cas SEPINST :
      A2 = Rec_seq_inst
      creer_seqinst(A1, A2)
   // cree un noeud N_SEPINST de fils gauche A1
   //        et de fils droit A2
```

Construction de l'Arbre Abstrait (2)

inst $\rightarrow$ IF condition THEN seq_inst ELSE seq_inst FI

```
Rec_inst =
    Ast Acond, Athen, Aelse
    selon (LC.nature)
         cas AFF : // cree et renvoie un AST pour une affectation
         cas IF :
             avancer
             Acond = Rec_condition()
             si (LC.nature = THEN) alors avancer sinon Erreur
             Athen = Rec_seq_inst
             si (LC.nature = ELSE) alors avancer sinon Erreur
             Aelse = Rec_seq_inst
             si (LC.nature = FI) alors avancer sinon Erreur
             return creer_if(Acond, Athen, Aelse)
                // cree le noeud N_IF a partir de ses 3 fils
```

Interprétation d'une instruction conditionnelle

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")

Autre extension : entrées-sorties

- lire des entrées au clavier $\rightarrow$ `lire`
- afficher des sorties à l'écran $\rightarrow$ `ecrire`

Exemple :

```
lire (X) ;
lire (Y) ;
if X > Y then
   ecrire (X) ;
else
   ecrire (Y) ;
fi
```

Grammaire :

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