INF404 - Travaux pratiques - Séances 7 et/ou 8

Un petit langage de programmation (suite et fin)

Ce TP est à adapter en fonction du langage que vous avez choisi de traiter dans le projet ...

Avant de commencer cette séance :

1. Créez un répertoire ***TP7*** dans votre répertoire ***INF404***
2. Placez-vous dans ***INF404/TP7*** et recopiez les fichiers utilisés pendant le TP6 : `cp ../TP6/* .`

L'objectif de ce TP est de compléter l'**interpreteur** commencé lors du précédent TP. On rappelle que cet interpréteur doit pouvoir au minimum :

— analyser (analyse lexicale et syntaxique) une séquence d'affectation ;
— afficher la valeur des variables, soit après chaque affectation, soit à la fin du programme.

La définition du langage (lexique et syntaxe) reste libre, on donne ci-dessous un exemple :

```
lire (x) ;
r := 1 ;
while x > 0 do
  r := r * x ;
  x : x - 1
od ;
ecrire (r)
```

Lexique et Syntaxe du langage

Dans ce langage un programme est une suite (non vide) d'instructions. Une instruction peut être soit une affectation, soit une instruction conditionnelle (`if`), soit une instruction itérative (`while`), soit une instruction d'entrée-sortie (`lire`, `ecrire`), etc.

La grammaire de ce langage pourra donc être de la forme :

pgm	→	seq_inst
seq_inst	→	inst suite_seq_inst
suite_seq_inst	→	`SEPINST` seq_inst
suite_seq_inst	→	ε
inst	→	`IDF AFF` eag
inst	→	autres formes d'instructions ...

On pourra choisir de représenter le lexeme `SEPINST` par un `';'`, et on complètera le lexique au fur et à mesure de l'ajout de nouvelles instructions ....

Instruction itérative

On donne ci-dessous un exemple de règle pour l'instruction itérative :

$$\begin{aligned} \text{inst} &\rightarrow \texttt{WHILE}\ \text{condition}\ \texttt{DO}\ \text{seq\_inst}\ \texttt{OD} \\ \text{condition} &\rightarrow \text{eag}\ \texttt{OPCOMP}\ \text{eag} \end{aligned}$$

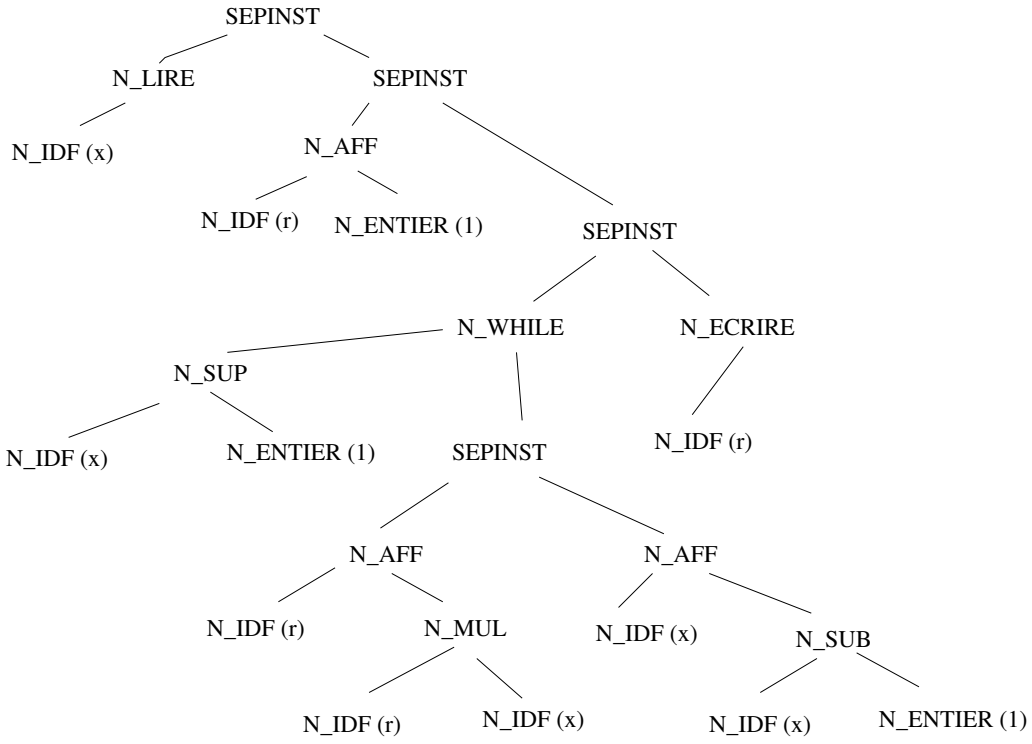
Ici "condition" désigne une expression booléenne avec une syntaxe simple dans laquelle `OPCOMP` désigne un opérateur de comparaison ($=$, $\neq$, $\leq$, $\geq$, $<$, $>$, etc.) entre deux expressions arithmétiques, comme dans le TP6.

Analyse

Compléter les analyses lexicales et syntaxiques de votre interpréteur pour prendre en compte cette instruction itérative (sans construction de l'AsT dans un premier temps).

Interprétation

Pour cette instruction il est préfèrable de construire lors de l'analyse syntaxique une représentation intermédiaire complète du code du programme, par exemple sous forme d'un arbre abstrait (AST). On donne ci-dessous un AST représentant le programme exemple donné en introduction :



construction de l'AST

Il s'agit ici de modifier à nouveau l'analyse syntaxique pour construire un arbre abstrait complet de l'ensemble du programme lu. Il faut donc définir cet arbre abstrait (quelles infos mémoriser pour chaque instruction, comment représenter une séquence d'instruction ?). Il faudra ensuite compléter la définition du type Âst en fonction de ces choix. Typiquement :

- Pour une affectation (IDF AFF eag) le noeud `N_AFF` aura un IDF pour fils gauche et une eag pour fils droit ;
- Pour une instruction conditionnelle (IF condition THEN seq_inst ELSE seq_inst FI) le noeud `N_IF` aura trois fils : une condition pour fils gauche, la partie "then" pour fils central et la partie "else" pour fils droit ;
- Pour une instruction itérative (WHILE condition DO seq_inst OD) le noeud `N_WHILE` aura une condition pour fils gauche et le corps de la boucle pour fils droit.

Pour construire l'AST d'une séquence d'instruction (seq_inst) on pourra s'inspirer de la solution vue en cours.

parcours de l'AST pour l'interprétation

Il s'agit ici d'écrire une fonction de parcours de l'arbre abstrait (AST) construit lors de l'analyse syntaxique. Le rôle de cette focntion est de simuler l'exécution du programme représenté par cet AST. Pour un noeud `N_WHILE` le code de cette fonction pourra être de la forme :

```
void Interpretation (Ast A) {
// A est l'AST du programme a interpreter
   ....
   switch (Ast->nature) {
        ....
       case N_WHILE:
       while (Evaluer(A->fgauche)) {
       // la fonction Evaluer evalue l'AST d'une condition booleene
   (renvoie 0 ssi cette condition est fausse)
      Interpretation(A->fdroit) ;
   } ;
       break ;
        ....
   } ;
}
```

Variantes et extensions ...

Quelques pistes pour prolonger cette version "minimale" de l'interpréteur, et anticiper sur la suite :

- imposer que les variables soit déclarées en début de programme (et donc détecter les doubles déclarations, les variables non déclarées) ;
- paramètrer l'interpréteur pour que l'utilisateur puisse choisir quelles variables il veut afficher, à quelles instructions ? Prévoir un mode "exécution en pas à pas" ;
- etc. !