TP PLY3

Fonctions

Prérequis : pour ce TP, vous devez avoir un calculateur :

- sans conflits
- avec un AST et 2 fonctions d'évaluation evalExpr et evalInst
- avec les if et les boucles while (et for eventuellement)

Principe général de gestion des fonctions :

- **définition des fonctions** : stockées sous la forme d'un tuple dans un dict *functions*

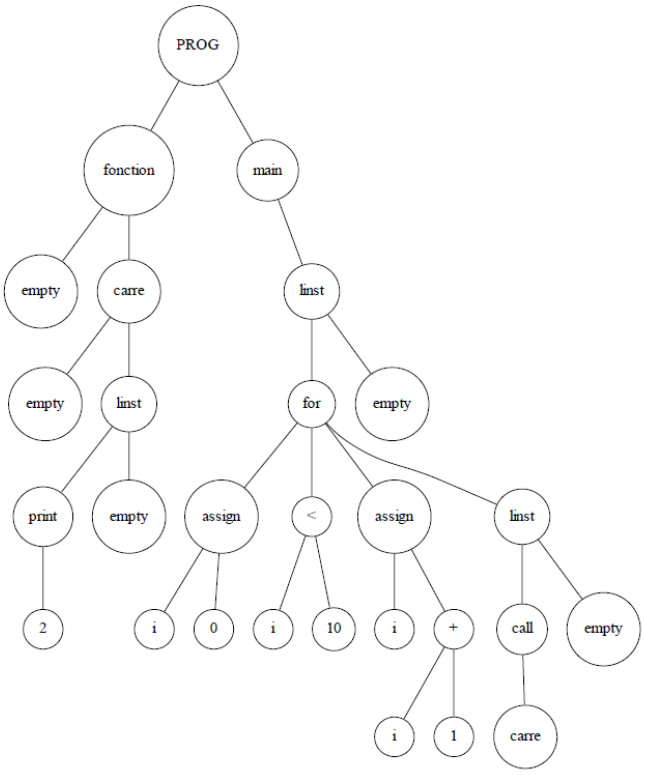
  Par exemple :

  {'carre': ('empty', ('bloc', ('print', 2), 'empty'))}

  {NOM : (paramètres, corps)}
- **appel des fonctions** : nouvelle instruction : CALL

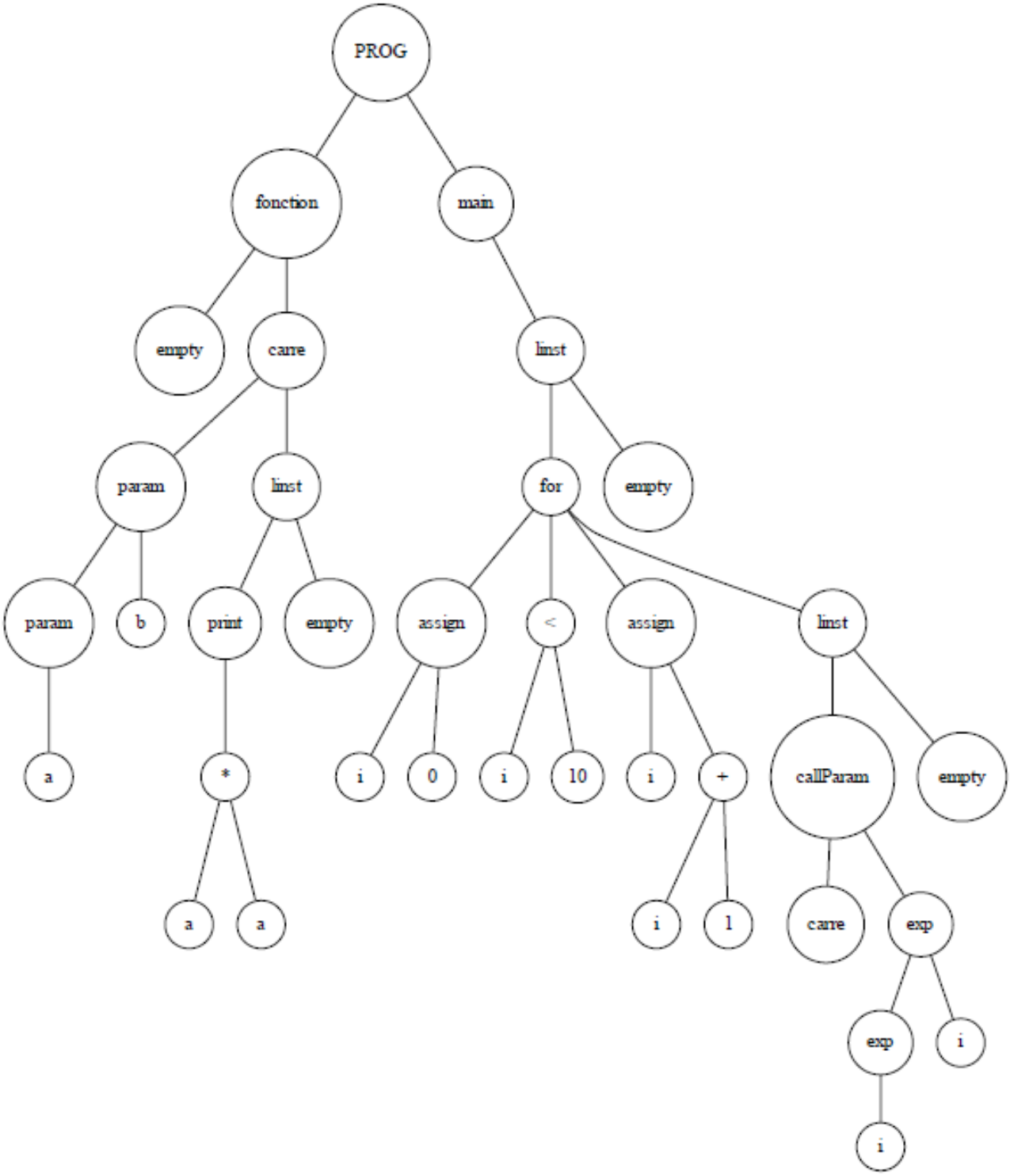
1. Fonctions void sans paramètre

s='fonction carre(){print(2);}for(i=0;i<10;i=i+1;){carre();}'



2. Fonctions void avec paramètre

s='fonction carre(a,b){print(a*a);}for(i=0;i<10;i=i+1;){carre(i, i);}'



3. Fonctions avec paramètres et return

Indications :

1. Stockage : Distinguer les fonctions *void* des fonctions *non void* : ajouter un attribut dans le dict functions ou faire 2 dict
2. AST et grammaire :

   Distinguer les :

   - FonctionValue et CallValue (evalExpr)
   - FonctionVoid et CallVoid (evalInst)
3. Cas d'une valeur à retourner : plusieurs choix possibles

Méthode	Avec return explicite et forcément en fin de bloc	Avec return explicite et coupe circuit	Avec return implicite : exemple la valeur se trouve dans une variable qui a le même nom que la fonction

Déclaration	FonctionValue toto(a, b){
	c=a+b;
	Return c;
	}
		FonctionValue toto(a, b){
		c=a+b;
		Return c;
		Print(1);
		}
			FonctionValue toto(a, b){
			toto=a+b;
			Print(1);
			}

Console suite à print(toto(1,2));	3	3 et pas de 1	1 et 3

4. Gestion du scope des variables

FonctionVoid f(a){print(a+1) ;return ;}

FonctionValue g(a){a=a+1 ;b=1 ;f(a*2) ;return a ;}

Main(){x=2 ;print(g(x)) ;}

Principe de la pile d'exécution

FonctionVoid f(a) {
print(a+1) ;
return ;}
FonctionValue g(a) {
a=a+1 ;b=1 ;
f(a*2) ;
return a ;}
Main(){
x=2 ;
print(g(x)) ;}
	FonctionVoid f(a) {
	print(a+1) ;
	return ;}
	FonctionValue g(a) {
	a=a+1 ; b=1 ;
	f(a*2) ;
	return a ;}
	Main(){
	x=2 ;
	print(g(x)) ;}
		FonctionVoid f(a) {
		print(a+1) ;
		return ;}
		FonctionValue g(a)
		{ a=a+1 ; b=1 ;
		f(a*2) ;
		return a ;}
		Main(){
		x=2 ;
		print(**g(x)**) ;}
			FonctionVoid f(a) {
			print(a+1) ;
			return ;}
			FonctionValue g(a) {
			a=a+1 ; b=1 ;
			f(a*2) ;
			return a ;}
			Main(){
			x=2 ;
			print(g(x)) ;}
				FonctionVoid f(a) {
				print(a+1) ;
				return ;}
				FonctionValue g(a) {
				a=a+1 ; b=1 ;
				f(a*2) ;
				return a ;}
				Main(){
				x=2 ;
				print(g(x)) ;}
					FonctionVoid f(a) {
					print(a+1) ;
					return ;}
					FonctionValue g(a) {
					a=a+1 ; b=1 ;
					f(a*2) ;
					return a ;}
					Main(){
					x=2 ;
					print(g(x)) ;}

Main, {}	Main, {x :2}	g, {a :2} Main, {x :2}	g, {a :3, b :1} Main, {x :2}	f, {a : 6} g, {a :3, b :1} Main, {x :2}	g, {a :3, b :1} Main, {x :2}

FonctionVoid f(a) {
print(a+1) ;
return ;}
FonctionValue g(a) {
a=a+1 ; b=1 ;
f(a*2) ;
return a ;}
Main(){
x=2 ;
print(g(x)) ;}

Main, {x :2}

cf : https://fr.wikipedia.org/wiki/Pile_d%27ex%C3%A9cution

https://fr.wikipedia.org/wiki/Trace_d%27appels

4. Récursivité terminale

Dans une fonction récursive, si le return ne comprend que l'appel récursif (et aucune opération supplémentaire), on dit que la récursion est terminale
https://fr.wikipedia.org/wiki/R%C3%A9cursion_terminale

Suite de Fibonacci : fonction récursive

```
def fiboRec(n):
    if n ==0 or n==1 : return n
    return fiboRec(n-1)+fiboRec(n-2)


print(fiboRec(9))
```

Suite de Fibonacci : fonction récursive terminale

```
def fibo(n, som, som2):
    if n != 1:
        return fibo(n-1, som+som2, som)
    return som

print(fibo(9, 1, 0))
```

5. Variables globales

Principe : si on ne trouve pas une variable (une key du dict), on va chercher plus loin dans la pile