Théorie des langages et outils pour la compilation

ESGI 3A AL

Projet mini langage

Informations pratiques :

- Les dates de la pré soutenance, et de la soutenance sont visibles sur myges.

REMARQUE : Les discussions entre groupes de projet sont encouragées mais vous devez impérativement écrire vous-même le code. Le partage de code entrainera automatiquement un 0 pour les deux parties.

Sujet :

L'objectif du projet est de concevoir un interpréteur pour un mini langage. Il est obligatoire de baser l'interprétation sur un arbre de syntaxe abstrait construit au cours de l'analyse syntaxique du « programme – input » donné en entrée.

Libertés

- syntaxe de l'input : libre
- arbre de syntaxe abstrait : nommage des sommets libre et structure libre (ordre des fils d'un sommet, ordre pour la structure de bloc

Spécifications de la version minimale (8/20) :

1. Votre interpréteur devra gérer les noms de variables à plusieurs caractères.
2. Gérer les instructions suivantes :
    a. affectation
    b. affichage d'expressions numériques (pouvant contenir des variables numériques)
    c. instructions conditionnelles : implémenter le si-alors-sinon/si-alors
    d. structures itératives : implémenter le while et le *for*
    e. Affichage de l'arbre de syntaxe (sur la console ou avec graphViz)

Gros Bonus (entre autre):

1. Fonctions void : 2 pts
2. Fonctions avec return et scope des variables et pile d'execution (5 pts)
3. Les tableaux (suivant degré d'aboutissemnt)
4. la POO (suivant degré d'aboutissemnt)
5. Gérer le passage des paramètres par référence (cf id() en python) (suivant degré d'aboutissemnt)

Petits bonus (entre autre): (1.5 pt max pour l'ensemble des petits bonus)

1. Gestion des erreurs (variable non initialisée, …)
2. Gérer la déclaration explicite des variables
3. Gestion du type chaine de caractères (et extension d'autant de l'instruction d'affichage)
4. Gestion des variables globales
5. affectations multiples à la python : a, b = 2, 3

6. comparaison multiples à la python : 1<2<3
7. print multiples : print(x+2, « toto ») ;
8. incrémentation et affectation élargie : x++, x+=1
9. possibilités de mettre des commentaires dans le code (et génération automatique d'une docString)
10. printString

Rendu :

- votre **code** (1 ou plusieurs fichiers)
- un fichier **readme** qui détaille
    - les fonctionnalités implémentées
    - les différents inputs associés aux fonctionnalités ci dessus

Opérations sur la pile (stack)

Pour return coupe circuit + scope des variables :

Vous devrez surement supprimer le mécanisme suivant dans evalInst(t)

```
if t[0]=='bloc' :
    evalInst(t[1])
    evalInst(t[2])
```

Exemple :

stack = []

input =
func toto(x) if(x>0) print(x); end return 1; print(2) ; end
main c=2; print(toto(6));

stack = [main{} / print(toto(6)) / assign]

pop de assign (x=2)

stack = [main{‘x’ :2} / print(toto(6))

pop de print(toto(6))

push du contexte d’exécution toto
stack = [main{‘x’ :2} / toto{}]

push du corps de la fonction toto avec la paramètre x = 6

stack = [main{‘x’ :2} / toto{‘x’:6} / print(2) / return 1 / if(x>0)…]
pop de if(x>0)print(x)
stack = [main{‘x’ :2} / toto{‘x’:6} / print(2) / return 1

push de print(x)

stack = [main{‘x’ :2} / toto{‘x’:6} / print(2) / return 1 / print(x)]
pop de print(x)
CALC> 6

stack = [main{‘x’ :2} / toto{‘x’:6} / print(2) / return 1]
pop du return 1

stack = [main{‘x’ :2} / toto{‘x’:6}, returnValue = 1 / print(2)]
flush de la fin de la fonction toto (return coupe circuit)

stack = [main{‘x’ :2} / toto{‘x’:6}, returnValue = 1]

pop du contexte d’exécution toto

stack = [main{‘x’ :2}]

… on peut désormais finir le print(toto(6)) qui était en suspend
CALC>1

on pop le main
stack = []

Remarque

Dans cet exemple, la pile contient des instructions (à stocker sous forme de tuple)
et des contextes d’exécution.
Rien ne vous empêche de gérer 2 piles distinctes.
,