Projet Architecture des ordinateurs

Analyseur Syntaxique de parenthésage
Assembleur x86-64

GHODBANE Rachid

Licence 2 Informatique – Université Jean Monnet 2024/2025

matricule n°19000721u

Table des matières

1 Présentation générale

Ce document fait l'objet d'une proposition de solution simple, en langage assembleur x86-64, permettant l'analyse syntaxique du parenthésage d'une chaîne de caractères (avec paires de parenthèses personnalisables).

1.1 Entrées et sorties

— **Entrée** : n paires de parenthèses et la chaîne de caractères à analyser
— **Sortie** : `OUI` si les parenthèses sont valides, `NON` sinon

1.2 Cas d'erreur

Le programme affiche `NON` dans les cas suivants :
— Trop de parenthèses fermantes
— Trop de parenthèses ouvrantes
— Parenthèses mal appariées
— Nombre d'arguments insuffisant (moins de 2)

1.3 Algorithme

Algorithme 1 : Vérification de parenthèses personnalisées

```
Données : n paires de parenthèses + chaîne à analyser
Résultat : OUI si valide, NON sinon
Initialiser pile vide, compteur ← 0;
foreach caractère c dans la chaîne do
    foreach paire p dans les parenthèses do
        if c = parenthèse ouvrante de p then
            Empiler c, compteur ← compteur + 1;
        if c = parenthèse fermante de p then
            if c correspond au sommet de pile then
                Dépiler, compteur ← compteur − 1;
            else
                Afficher NON et terminer;
if compteur = 0 then
    Afficher OUI;
else
    Afficher NON;
```

2 Programme x86-64

```
.data
    msg_usage: .string "2 args\n"
    msg_non:   .string "\nNON\n"
    msg_oui:   .string "\nOUI\n"

.text
.global _start

# Affiche chaine
affiche_chaine:
    xor %rdx, %rdx
    mov %rsi, %rbx
boucle_affichage:
    movb (%rbx), %al
    test %al, %al
    jz fin_affichage
    inc %rdx
    inc %rbx
    jmp boucle_affichage
fin_affichage:
    mov $1, %rax
    mov $1, %rdi
    syscall
    jmp fin_programme

# Usage
usage:
    mov $msg_usage, %rsi
    call affiche_chaine
    jmp fin_programme

# Verification parentheses
initialisation_des_registres:
    xor %rax, %rax
    xor %rbp, %rbp
    xor %r12, %r12
    push %rbp

charger_texte:
    xor %r8, %r8
    mov (%r15, %r14, 8), %r8

charger_parenthese:
    xor %r11, %r11
    mov (%r15, %r13, 8), %r11
    jmp parcourt_chaine

parcourt_chaine:
    movb (%r8), %al
    test %al, %al
    jz fin_verif
    jmp parcourt_parenthese

empiler_parenthese:
    push (%r11)
    inc %r12
    jmp caractere_suivant

depiler_parenthese:
    dec %r11
    movb (%rsp), %al
    cmpb (%r11), %al
    jne erreur
    pop %rbp
    dec %r12
    jmp caractere_suivant

parcourt_parenthese:
    cmpb (%r11), %al
    je empiler_parenthese
    inc %r11
    cmpb (%r11), %al
    je depiler_parenthese
    dec %r11
    jmp parenthese_suivant

parenthese_suivant:
    inc %r13
    cmp %r14, %r13
    jne charger_parenthese
    jmp caractere_suivant

caractere_suivant:
    xor %r13, %r13
    mov $1, %r13
    inc %r8
    jmp parcourt_chaine

fin_verif:
    cmp $0, %r12
    je fin_valide
    jmp erreur

fin_valide:
    mov $msg_oui, %rsi
    call affiche_chaine
    jmp fin_programme

erreur:
    mov $msg_non, %rsi
    call affiche_chaine

# Main
_start:
    mov %rsp, %r15
    movq (%r15), %rbx
    cmp $3, %rbx
    jl usage
    movq %rbx, %r14
    xor %r9, %r9
    xor %r10, %r10
    mov $1, %r13
    call initialisation_des_registres

fin_programme:
    mov $60, %rax
    xor %rdi, %rdi
    syscall
```

3 Explication détaillée du code

3.1 Point d'entrée principal (_start)

Le programme commence par vérifier le nombre d'arguments puis initialise les registres pour l'analyse.

```
_start:
    mov %rsp, %r15
    movq (%r15), %rbx
    cmp $3, %rbx
    jl usage
```

Fonctionnement : Sauvegarde le pointeur de pile dans r15, charge le nombre d'arguments dans rbx, et vérifie qu'il y a au moins 2 arguments (+ le nom du programme = 3 total).

3.2 Chargement de la chaîne à analyser

La chaîne à analyser est le dernier argument passé au programme.

```
charger_texte:
    xor %r8, %r8
    mov (%r15, %r14, 8), %r8
```

Fonctionnement : Calcule l'adresse `r15 + (r14 × 8)` pour récupérer l'adresse de la chaîne (dernier argument). r8 contiendra un pointeur vers la chaîne pour la parcourir caractère par caractère.

3.3 Parcours de la chaîne

Chaque caractère de la chaîne est lu et analysé jusqu'à la fin (caractère nul '0').

```
parcourt_chaine:
    movb (%r8), %al
    test %al, %al
    jz fin_verif
    jmp parcourt_parenthese
```

Fonctionnement : Charge le caractère courant dans al. Si c'est '
0', termine l'analyse. Sinon, vérifie si ce caractère est une parenthèse. Plus tard, `inc %r8` avancera d'un caractère.

3.4 Test des paires de parenthèses

Pour chaque caractère, le programme teste toutes les paires de parenthèses définies.

```
parcourt_parenthese:
    cmpb (%r11), %al
    je empiler_parenthese
    inc %r11
    cmpb (%r11), %al
    je depiler_parenthese
    dec %r11
    jmp parenthese_suivant
```

Fonctionnement : Compare le caractère avec la parenthèse ouvrante de la paire. Si correspondance, empile. Sinon compare avec la fermante (+1 octet). Si correspondance, dépile. Sinon essaie la paire suivante.

3.5 Empilage et dépilage

Empiler une parenthèse ouvrante :

```
empiler_parenthese:
    push (%r11)
    inc %r12
    jmp caractere_suivant
```

Empile la parenthèse ouvrante et incrémente le compteur.

Dépiler une parenthèse fermante :

```
depiler_parenthese:
    dec %r11
    movb (%rsp), %al
    cmpb (%r11), %al
    jne erreur
    pop %rbp
    dec %r12
    jmp caractere_suivant
```

Vérifie que la parenthèse au sommet de la pile correspond à l'ouvrante de la paire. Si oui, dépile et décrémente le compteur. Sinon, affiche une erreur.

3.6 Vérification finale

À la fin du parcours, le compteur doit être à 0 (toutes les parenthèses fermées).

```
fin_verif:
    cmp $0, %r12
    je fin_valide
    jmp erreur
```

Fonctionnement : Si le compteur r12 est à 0, toutes les parenthèses ouvrantes ont été correctement fermées → affiche `OUI`. Sinon, il reste des parenthèses non fermées → affiche `NON`.

4 Avantages du programme

- **Flexibilité** : Permet de définir des paires de parenthèses personnalisées (`()`, `[]`, `{}`, etc.).
- **Robustesse** : Vérifie le nombre d'arguments et gère les cas d'erreur appropriés.
- **Efficacité** : Utilise une pile pour un algorithme en temps linéaire O(n).
- **Modularité** : Code organisé en sous-programmes réutilisables.

5 Exemple d'utilisation

```
./analyseur "()" "(((blablabla)))"
Analyse : OUI

./analyseur "()" "((()))"
Analyse : NON

./analyseur "()" "{}"({ blabla )}"
Analyse : NON

./analyseur "[]" "ab" "aa[[a[ meow meow ]b]]bb"
Analyse : OUI
```