Filière Licence Sciences pour l'Ingénieur en Informatique - Université de Corse

Rapport Final sur la Réalisation d'un compilateur

Ce projet a pour objectif la creation progressive d'un interpreteur dedie au langage d'assemblage. Chaque etape du processus de developpement a apporte des ameliorations significatives, dotant l'interpreteur de nouvelles capacites d'analyse, d'interpretation, et d'execution des programmes. Le resultat final est un interpreteur fonctionnel capable de traiter des programmes ecrits en langage d'assemblage, avec des perspectives d'optimisation et d'extension pour les etapes a venir.

Encadré par le professeur : SANTUCCI, JEAN-FRANÇOIS.

Réalisé par : Mohamed Amine Boussoualef

Année scolaire 2023-2024

Introduction

Le developpement de l'interpreteur en langage d'assemblage a suivi plusieurs etapes cruciales, visant a construire un interpreteur robuste et fonctionnel. Chaque etape a apporte des ameliorations specifiques, renforçant la capacite de l'interpreteur a analyser, interpreter et executer des programmes ecrits dans ce langage.

Plan :

Étape 1 : Analyse Lexicale

L'analyse lexicale a ete la premiere etape du processus, consistant a decouper le code source en unites lexicales. Les tokens tels que "program," "var," et "const" ont ete identifies, marquant le debut de la construction de la structure syntaxique du programme.

Étape 2 : Analyse Syntaxique

L'etape suivante a introduit l'analyse syntaxique, visant a definir la structure grammaticale du langage d'assemblage. Les regles syntaxiques ont ete etablies, permettant la reconnaissance de la hierarchie des elements du programme.

Étape 3 : Analyse Sémantique et Gestion des Erreurs

L'analyse semantique a ete integree pour gerer la signification des identificateurs dans le programme. La creation d'une table des symboles a facilite le suivi des informations associees a chaque identificateur. La gestion des erreurs a ete renforcee, fournissant des messages d'erreur informatifs en cas de declarations incorrectes.

Étape 4 : Génération de Code P-Code

L'etape 4 a marque une avancee significative avec la generation de code P-Code. Cette transformation a permis de traduire les structures syntaxiques en instructions executables. Les procedures ont ete etendues pour inclure la generation de code pour les expressions, les conditions, les affectations, les entrees/sorties, et les structures de controle.

Tests

Conclusion

Étape 1 : Analyse Lexicale

Introduction :

Ce programme represente un interpreteur conçu pour executer des programmes ecrits dans un langage d'assemblage simple. Chaque etape du programme, telle que la premiere, contribue a l'implementation complete de cette machine virtuelle d'assemblage.

Allocation de Mémoire et Interaction Utilisateur

La premiere etape du programme joue un ro^ le crucial dans la preparation de la memoire et l'interaction avec l'utilisateur. L'utilisation des mnemoniques d'instructions, les codes operationnels definis par le langage d'assemblage, guide le comportement de l'interpreteur. Cette premiere sequence d'instructions demontre comment le programme alloue de l'espace memoire, sollicite l'entree utilisateur, effectue des operations arithmetiques simples, et prend des decisions conditionnelles.

Plus specifiquement :

L'instruction "INT" alloue deux emplacements memoire supplementaires pour le stockage des donnees.

Les instructions "LDA" et "LDV" manipulent les adresses et les valeurs dans la memoire.

L'instruction "INN" permet a l'utilisateur d'entrer une valeur, laquelle est stockee a une adresse specifiee dans la memoire.

Des operations arithmetiques telles que "ADD" sont effectuees, resultant en des modifications de la memoire.

L'instruction "BZE" introduit une decision conditionnelle, sautant a une adresse specifiee si une condition est remplie.

Explication détaillé du code :

MNEMONIQUES : Cette liste regroupe les mnemoniques des instructions, definissant ainsi les codes operationnels que le programme peut executer. Chaque mnemonique correspond a une instruction specifique, comme l'addition (ADD), la soustraction (SUB), la multiplication (MUL), la division (DIV), etc.

interpreteur(PCODE) : La fonction principale, "interpreteur," accepte en parametre un programme en langage d'assemblage, represente sous forme d'une liste de tuples. Chaque tuple contient une instruction et son operande associe. L'interpreteur parcourt ces instructions pour executer le programme.

MEM (Mémoire) : Utilisee comme une pile, la memoire (MEM) stocke les donnees et les resultats intermediaires du programme. Les operations s'effectuent en manipulant les elements de cette pile.

SP (Stack Pointer - Pointeur de pile) : SP indique le sommet de la pile dans la memoire, facilitant l'ajout et la suppression d'elements.

PC (Program Counter - Compteur d'instructions) : PC pointe vers l'instruction en cours d'execution dans le programme. Son ro^ le est de suivre l'avancement du programme.

PS (Program Status - État du programme) : Initialement defini sur "EXECUTION," PS represente l'etat actuel du programme. Lorsque PS atteint "END," l'execution prend fin.

INST (Instruction) : INST stocke le mnemonique de l'instruction en cours d'execution, guidant le programme sur la nature de l'operation a effectuer.

OPERANDE : Chaque instruction est associee a un operande qui determine son comportement specifique. Par exemple, l'instruction "INT" alloue de la memoire en fonction de cet operande.

Le cœur du programme reside dans une boucle principale (while PS != "END") ou chaque iteration traite une instruction. Les instructions sont implementees a l'interieur de la structure conditionnelle (if-elif-else) en fonction de leur mnemonique.

Par exemple, pour l'instruction "ADD," le programme realise une addition en manipulant les elements du sommet de la pile. Des operations similaires sont effectuees pour les mnemoniques "SUB," "MUL," et "DIV."

L'instruction "INN" sollicite l'interaction de l'utilisateur pour fournir une valeur, laquelle est ensuite stockee a une adresse specifiee dans la memoire. A l'inverse, l'instruction "PRN" imprime la valeur du sommet de la pile.

Enfin, l'instruction "HLT" met un terme a l'execution du programme."

Cette premiere sequence cree un aperçu detaille des operations fondamentales et de l'interaction avec l'utilisateur au sein de la machine virtuelle d'assemblage. Ces operations elementaires seront ensuite exploitees et etendues dans les etapes ulterieures du programme pour accomplir des ta^ ches plus complexes.

Conclusion de l'Étape 1 : Fondations de l'Interpréteur

Cette premiere sequence, axee sur l'allocation de memoire et l'interaction utilisateur, constitue les fondations essentielles de notre interpreteur de langage d'assemblage. En demontrant comment le programme alloue et manipule la memoire, sollicite l'entree utilisateur, et effectue des operations de base, cette etape cree une infrastructure robuste pour les etapes ulterieures. Ces operations elementaires, combinees aux mnemoniques et a la logique de contro^ le du programme, posent les bases sur lesquelles des fonctionnalites plus complexes seront edifiees. L'analyse detaillee de cette etape permet de mieux comprendre le fonctionnement interne de notre machine virtuelle d'assemblage, pavant la voie a l'accomplissement de ta^ ches plus avancees dans les prochaines phases du programme.

Étape 2 : Analyse Syntaxique

Introduction

l'analyse syntaxique, visant a definir la structure grammaticale du langage d'assemblage. Les regles syntaxiques ont ete etablies, permettant la reconnaissance de la hierarchie des elements du programme

Traitement des Instructions

Le traitement des instructions a ete optimise pour garantir une separation adequate entre les differents elements du code source. Les caracteres speciaux tels que ";", ",", et "\n" sont correctement geres, ce qui simplifie la structure du code avant l'analyse. Cela ameliore la lisibilite du code et facilite l'identification des elements cles.

```
if lettre == ";":
    code += " ; "
elif lettre == ",":
    code += " , "
elif lettre == "\n":
    code += " "
else:
    code += lettre
```

Gestion des Blocs

L'analyseur lexical identifie maintenant de maniere robuste les blocs "begin" et "end" ainsi que les declarations de constantes et de variables a l'interieur de ces blocs. Cette gestion precise des blocs contribue a une meilleure comprehension de la structure du code, facilitant ainsi les etapes ulterieures de l'interpretation.

```
assert ['begin'] in instructions, "SyntaxError: BEGIN expected"
```

Traitement des Erreurs

```
assert re.match("[a-zA-Z_][a-zA-Z_0-9]*", nom_const), f"SyntaxError: \"{nom_const}\"
```

```
is not a legal const name"
```

Des assertions ont ete integrees pour detecter les erreurs potentielles de syntaxe des l'analyse lexicale. Par exemple, des verifications sont effectuees pour s'assurer que la declaration de constante est correcte, que les noms de variables sont valides, et que le bloc principal se termine par "end". Ces assertions renforcent la fiabilite du processus d'analyse.

Intégration avec le Code Principal

Encapsulation dans une Fonction

Le code de l'analyseur lexical a ete encapsule dans une fonction appelee analyseur_lexical, prenant un fichier en parametre. Cette modularite facilite l'utilisation de l'analyseur lexical dans le code principal et favorise une structure plus propre et comprehensible.

```
def analyseur_lexical(fichier:str):
    # ...
    return instructions
```

Interaction entre les Deux Analyseurs Lexicaux

Une premiere version d'integration a ete realisee entre les analyseurs lexicaux du code principal et de l'aide. L'analyseur lexical de l'aide genere des listes de declarations de constantes et de variables, et l'objectif est d'harmoniser ces listes avec celles generees par le code principal. Cette interaction pose les bases d'une approche unifiee pour traiter le langage d'assemblage.

Conclusion de l'Étape 2

L'Etape 2 represente une etape cruciale dans l'amelioration de la robustesse de l'analyseur lexical. Les ameliorations apportees permettent une gestion plus precise du code source, tandis que l'integration entre les deux analyseurs lexicaux renforce la coherence du projet. Ces ajustements jetteront les bases d'une implementation solide pour les etapes suivantes du developpement. L'attention portee aux details et la gestion proactive des erreurs sont des elements cles de cette avancee.

Étape 3 :

Analyse Sémantique et Gestion des Erreurs

Introduction

L'Etape 3 du developpement de l'interpreteur de langage d'assemblage marque une avancee majeure dans l'amelioration de la robustesse de l'interpreteur. Les principales fonctionnalites ajoutees comprennent l'analyse semantique, la gestion des erreurs, et la creation d'une table des symboles. Ces ameliorations sont cruciales pour garantir une interpretation correcte des programmes.

Analyse Sémantique

L'analyse semantique constitue une etape cle dans le traitement des programmes. Elle se concentre sur la gestion de la table des symboles (TABLESYM), qui agit comme une base de donnees centralisee pour stocker des informations sur chaque identificateur utilise.

```
# Initialisation de l'offset et de la table des symboles
offset = 0
TABLESYM = []
```

Table des Symboles

La table des symboles est une liste de triplets, stockant le nom, la classe (programme, constante, variable), et l'adresse memoire de chaque identificateur.

```
# Fonction pour entrer un symbole dans la table des symboles
def entrerSym(classe, value):
    global TABLESYM, offset
    if classe == 'constant':
        value = PROGRAM[i + 1]
    TABLESYM += [(PROGRAM[i - 1], classe, value)]
    offset += 1
```

Gestion des Constantes et Variables

Les fonctions consts et Vars ont ete etendues pour integrer la creation des symboles correspondants dans la table des symboles. Cela garantit une declaration correcte de chaque identificateur utilise dans le programme.

Allocation de Mémoire et Gestion des Erreurs

L'allocation de memoire repose sur un offset, qui s'incremente a mesure que de nouveaux symboles sont ajoutes. Cette approche assure une utilisation efficace de la memoire.

```
# Fonction pour chercher un symbole dans la table des symboles
def chercherSym(sym):
    global TABLESYM
    res = False
    for s in TABLESYM:
        if s[0] == sym:
            res = s
    if res:
        return res
    else:
        erreur_dec(sym)
```

Gestion des Erreurs

La gestion des erreurs a ete renforcee pour fournir des messages d'erreur detailles. Par exemple, si un identificateur n'est pas declare, un message specifique est affiche.

Perspectives Futures

L'etape 3 offre une base solide pour l'analyse de programmes plus complexes en langage d'assemblage. Les developpements futurs pourraient inclure l'extension des fonctionnalites, l'ajout de structures de contro^ le avancees, et des ameliorations continues de la gestion des erreurs.

Conclusion

L'Etape 3 represente un progres significatif dans le renforcement de l'interpreteur. Les nouvelles fonctionnalites, en particulier la gestion avancee des identificateurs via la table des symboles, augmentent la qualite de l'analyse semantique. Ces ajustements preparent le terrain pour des developpements futurs plus avances, consolidant ainsi la capacite de l'interpreteur a traiter des programmes complexes en langage d'assemblage.

Étape 4 :

Génération de Code P-Code

Introduction

L'etape 4 represente une avancee majeure dans le developpement de l'interpreteur en se focalisant sur la generation de code P-Code. Cela permet de traduire les structures syntaxiques en instructions executables, marquant ainsi une etape cruciale vers l'execution effective des programmes analyses.

Génération de Code lors de l'Allocation des Données

La premiere etape consiste a allouer l'espace necessaire dans la pile P-Code, realisee par la procedure BLOCK. L'instruction P-Code INC est generee pour reserver cet espace.

```
def block():
    global offset
    if token == "const":
        consts()
    if token == "var":
        Vars()
    generer2('INC', offset)
    insts()
```

MonCode.code

```
program abc ;
const C = 10 ; var A,B ; (* Ceci est un commentaire *)
begin
A := 0 ;
B := 0 ;
while A <> 0 do
begin
read ( A ) ; (* entrer la valeur de A tant que A different 0 *)
B := A + B ;
end ;
write ( B ) ;  (* affiche le resultat final de B *)
end .
```

Terminaison du Programme et Instruction HLT

La procedure PROGRAM genere l'instruction P-Code HLT pour indiquer la fin du programme.

```
def program():
    teste("program")
    test_et_entre(ID, 'program')
    teste(";")
    block()
    generer1('HLT')
    if token != ".":
        erreur(".", token)
```

Génération de Code pour les Expressions

La generation de code pour les expressions est detaillee dans les procedures TERM, FACT, et EXPR. Chaque terme laisse une valeur sur la pile P-Code, et les operations de multiplication et de division sont prises en compte.

```
def term():
    global token
    fact()
    while token in ["*", "/"]:
        op = token
        next_token()
        fact()
        if op == "*":
            generer1('MUL')
        else:
            generer1('DIV')
```

Génération de Code pour les Conditions

La generation de code pour les conditions suit une approche similaire a celle des expressions, en prenant en compte les operations relationnelles.

```
def cond():
    expr()
    if token in ["==", "<>", "<", ">", "<=", ">="]:
        next_token()
        expr()
```

Génération de Code pour les Instructions d'Affectation

La procedure AFFEC genere le code P-Code pour une instruction d'affectation A := expression.

```
def affec():
    global token, PLACESYM
    ADDR = getAdresseFromTableSym(token)
    test_et_cherche(ID)
    generer2('LDA', ADDR)
    teste(":=")
    expr()
    generer1('STO')
```

Génération de Code pour les Instructions d'Entrée/Sortie

Les instructions d'entree/sortie, ECRIRE et LIRE, sont egalement gerees dans la generation de code.

```
def ecrire():
    teste("write")
    teste("(")
    expr()
    generer1('PRN')
    while token == ",":
        next_token()
        expr()
        generer1('PRN')
    teste(")")
```

Procédures de Génération de P-Code pour les Structures de Contrôle

Les procedures de generation de P-Code sont etendues pour traiter les structures de contro^ le **F** et WHILE.

```
def si():
    teste("if")
    cond()
    teste("then")
    generer2('BZE', 0)
    inst()
```

Conclusion

L'etape 4 marque une transition essentielle vers la concretisation des programmes analyses. La generation de code P-Code permet de traduire les structures syntaxiques en instructions executables, ouvrant la voie a une interpretation complete des programmes en langage d'assemblage. Les perspectives futures pourraient inclure l'optimisation du code genere et l'ajout de fonctionnalites avancees.

Test

Elever le mot program

MonCode.code

```
1   abc ;
2   const C = 10 ; var A,B ; (* Ceci est un commentaire *)
3   begin
4   A := 0 ;
5   B := 0 ;
6   while A <> 0 do
7   begin
8   read ( A ) ; (* entrer la valeur de A tant que A different 0 *)
9   B := A + B ;
10  end ;
11  write ( B ) ;  (* affiche le resultat final de B *)
12  end .
13
```

```
File "c:\Users\pc\Desktop\DOSSIER\compilation\projet\mon projet\aide.py", line 117, in <module>
  analyseur_lexical("MonCode.code")
File "c:\Users\pc\Desktop\DOSSIER\compilation\projet\mon projet\aide.py", line 51, in analyseur_lexical
  assert instruction[0] == "program", "SyntaxError: Missing program declaration in header"
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: SyntaxError: Missing program declaration in header
```

Elever le mot Const avant C

```
1   program abc ;
2   C = 10 ; var A,B ; (* Ceci est un commentaire *)
3   begin
4   A := 0 ;
5   B := 0 ;
6   while A <> 0 do
7   begin
8   read ( A ) ; (* entrer la valeur de A tant que A different 0 *)
9   B := A + B ;
10  end ;
11  write ( B ) ;  (* affiche le resultat final de B *)
12  end .
```

```
on/projet/mon projet/etape4.py"
ERREUR  expected:  begin  given:  C
```

Elever le mot var

```
1   program abc ;
2   const C = 10 ;
3   A,B ; (* Ceci est un commentaire *)
4   begin
5   A := 0 ;
6   B := 0 ;
7   while A <> 0 do
8   begin
9   read ( A ) ; (* entrer la valeur de A tant que A different 0 *)
10  B := A + B ;
11  end ;
12  write ( B ) ;  (* affiche le resultat final de B *)
13  end .
```

```
File "c:\Users\pc\Desktop\DOSSIER\compilation\projet\mon projet\aide.py", line 108, in analyseur_lexical
  assert element in listVar or element in listConst, f"SyntaxError: '{element}' variable not declared"
         ^^^^^^^
UnboundLocalError: cannot access local variable 'listVar' where it is not associated with a value
```

Elever le ;

```
program abc ;
const C = 10 ;
var A,B  (* Ceci est un commentaire *)
begin
A := 0 ;
B := 0 ;
while A <> 0 do
begin
read ( A ) ; (* entrer la valeur de A tant que A different 0 *)
B := A + B ;
end ;
write ( B ) ;  (* affiche le resultat final de B *)
end .
```

```
File "c:\Users\pc\Desktop\DOSSIER\compilation\projet\mon projet\aide.py", line 117, in <module>
  analyseur_lexical("MonCode.code")
File "c:\Users\pc\Desktop\DOSSIER\compilation\projet\mon projet\aide.py", line 83, in analyseur_lexical
  while instruction[i] != ";":
        ~~~~~~~~~~~~~~^^^
```

Elever le mot begin

```
1   program abc ;
2   const C = 10 ;
3   var A,B ; (* Ceci est un commentaire *)
4
5   A := 0 ;
6   B := 0 ;
7   while A <> 0 do
8   begin
9   read ( A ) ; (* entrer la valeur de A tant que A different 0 *)
10  B := A + B ;
11  end ;
12  write ( B ) ;  (* affiche le resultat final de B *)
13  end .
```

```
PS C:\Users\pc\Desktop\DOSSIER\compilation\projet\mon projet> & C:/Users/pc/AppData
on/projet/mon projet/etape4.py"
ERREUR  expected:  begin  given:  A
```

Elever le mot end a la fin

```
program abc ;
const C = 10 ; var A,B ; (* Ceci est un commentaire *)
begin
A := 0 ;
B := 0 ;
while A <> 0 do
begin
read ( A ) ; (* entrer la valeur de A tant que A differen
B := A + B ;
end ;
write ( B ) ;  (* affiche le resultat final de B *)
.
```

```
PS C:\Users\pc\Desktop\DOSSIER\compilation\projet\mon projet> & C:/
on/projet/mon projet/etape4.py"
ERREUR  expected:  end  given:  .
```

Utiliser variable non déclarer

```
1   program abc ;
2   const C = 10 ; var B ; (* Ceci est un commentaire *)
3   begin
4   A := 0 ;
5   B := 0 ;
6   while A <> 0 do
7   begin
8   read ( A ) ; (* entrer la valeur de A tant que A different 0 *)
9   B := A + B ;
10  end ;
11  write ( B ) ;  (* affiche le resultat final de B *)
12  end .
```

```
  File "c:\Users\pc\Desktop\DOSSIER\compilation\projet\mon projet\aide.py", line 108, in analyseur_lexical
    assert element in listVar or element in listConst, f"SyntaxError: '{element}' variable not declared"
                               ^^^^^^^^^^^^^^^^^^^^
AssertionError: SyntaxError: 'A' variable not declared
```

Test réussi

```
program abc ;
const C = 10 ; var A,B ; (* Ceci est un commentaire *)
begin
A := 0 ;
B := 0 ;
while A <> 0 do
begin
read ( A ) ; (* entrer la valeur de A tant que A different 0 *)
B := A + B ;
end ;
write ( B ) ;  (* affiche le resultat final de B *)
end .
```

```
on/projet/mon projet/etape4.py"
1
2
3
0
B = 6
```

Conclusion Final

Le developpement progressif de l'interpreteur a abouti a la creation d'un outil capable d'analyser et d'executer des programmes en langage d'assemblage. Chaque etape a contribue a renforcer la capacite de l'interpreteur, ouvrant la voie a des ameliorations continues et a des developpements futurs. Des perspectives comme l'optimisation du code genere et l'ajout de fonctionnalites avancees pourraient enrichir davantage cet interpreteur.