MIV 2019/2020

Rapport Projet

Compilation

BANDOUI Nazim 161631055599
MEKBAL Mohamed Amine 161631057438

Partie I :

Définition de la compilation :

La compilation informatique désigne le procédé de traduction d'un programme, écrit et lisible par un humain, en un programme exécutable par un ordinateur. De façon plus globale, il s'agit de la transformation d'un programme écrit en code source, en un programme transcrit en code cible, ou binaire. Habituellement, le code source est rédigé dans un langage de programmation (langage source), il est de haut niveau de conception et facilement accessible à un utilisateur. Le code cible, quant à lui, est transcrit en langage de plus bas niveau (langage cible), afin de générer un programme exécutable par une machine.

Etapes de la compilation :

Quelles sont les étapes de compilation d'un programme ? Un programme se compile selon une série d'étapes :

- l'analyse lexicale : elle divise le code source en petits morceaux : les tokens (les jetons). Chaque token a une unité lexicale unique de la langue (ou lexème), tel qu'un identifiant, un symbole ou un mot-clé par exemple.
- l'analyse syntaxique : elle analyse la séquence des tokens pour reconnaître la structure syntaxique du programme. On modifie la syntaxe linéaire des tokens par une structure en arborescence, générée selon la grammaire formelle qui détermine la syntaxe du langage.
- l'analyse sémantique : le compilateur, durant cette étape, complète l'arborescence par des informations sémantiques et érige la table des symboles. Cette phase teste les erreurs de type, ou une tâche définie (telles que les variables locales), ou l'objet de liaison, et peut même dispenser des avertissements ou rejeter des programmes inexacts.
- la modification du code source en code intermédiaire.
- l'allocation de registre avec la génération de codes et la traduction du code intermédiaire en code cible.
- l'édition des liens, pour finir.

Définition R :

Le langage R, un projet de GNU similaire à S, est un langage de programmation et un environnement mathématique utilisés pour le traitement de données et l'analyse statistique. Depuis plusieurs années, deux nouvelles versions apparaissent au printemps et à l'automne.

1 - Logo du langage R

R dispose de nombreuses fonctions graphiques. R est fondé sur le langage S qui a été développé par John Chambers des laboratoires Bell. R est considéré par ses créateurs comme étant une exécution de S, avec la sémantique dérivée du langage Scheme. R est librement disponible sous la GPL et est disponible pour Microsoft Windows, Macintosh et de nombreux systèmes de type Unix.

But du projet :

Le but de ce projet est de réaliser un mini-compilateur pour le langage R en passant par les différentes phases de la compilation à savoir l'analyse lexicale en utilisant l'outil FLEX et l'analyse syntaxico-sémantique en utilisant l'outil BISON, la génération du code intermédiaire, l'optimisation ainsi que la génération du code machine.

Les traitements parallèles concernant la gestion de la table des symboles ainsi que le traitement des différentes erreurs doivent être également réalisés lors des phases d'analyse du processus de compilation.

Langage et outils utilisés :

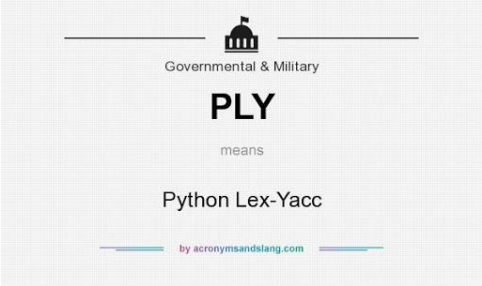
Python est un langage de programmation interprété, orienté objet, de haut niveau avec une sémantique dynamique. Ses structures de données intégrées de haut niveau, combinées au typage dynamique et à la liaison dynamique, le rendent très attrayant pour le développement d'applications rapides, ainsi que pour une utilisation comme langage de script ou de collage pour connecter les composants existants. Python supporte les modules et les paquets, ce qui encourage la modularité des programmes et la réutilisation du code.

2 - Logo du langage Python

Visual Studio Code est (selon Wikipédia) un éditeur de code extensible développé par Microsoft pour Windows, Linux et MacOs. Les fonctionnalités incluent la prise en charge du débogage, la mise en évidence de la syntaxe, la complétion intelligente du code, les snippets, la refactorisation du code et Git intégré

3 - Logo de Visual Studio Code (VS Code)

Python Lex-Yacc ou en abrégé PLY est une librairie de Python qui permet d'implémenter les outils d'analyse Lex et Yacc, elle dispose de toutes fonction nécessaire à la réalisation de ce projet.

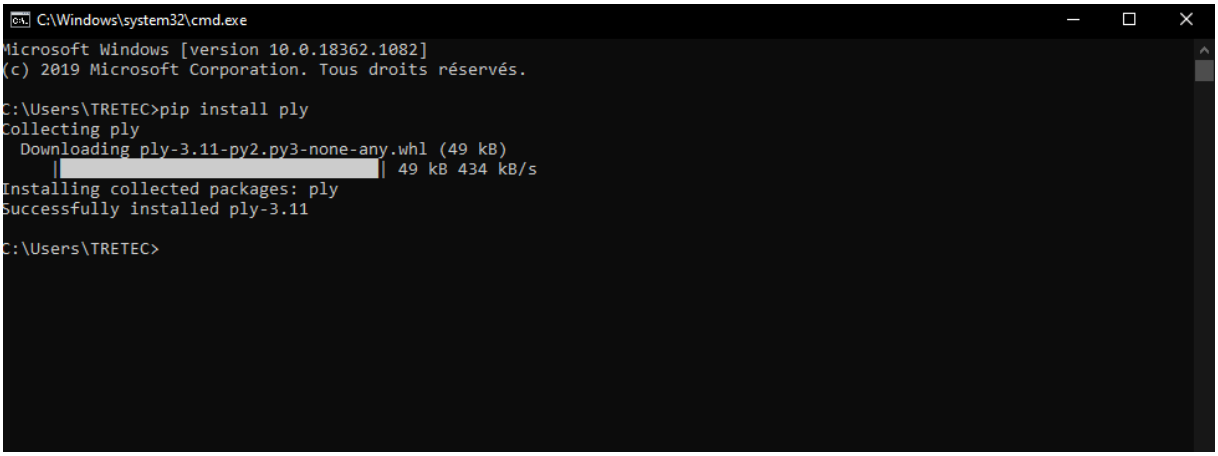


Pour installer PLY sur votre machine, ouvrez votre invite de commandes et tapez « pip install ply » sous Windows ou bien « pip3 install ply » sous linux.

N'oubliez pas d'importer la librairie et ses modules avant de commencer :

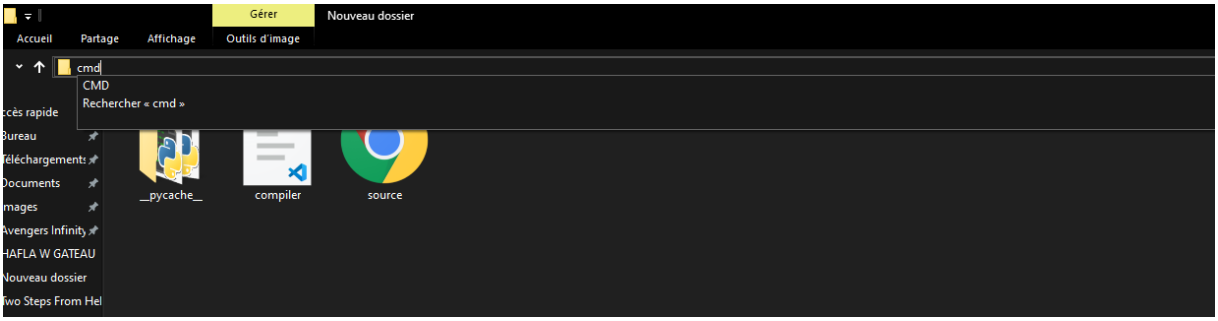
```
import ply
import ply.lex as lex
import ply.yacc as yacc
```

4 - Import des librairies utilisées



5 - Installation de la libraire Python Lex-Yacc

Pour exécuter le fichier, décompresser le fichier envoyée et ouvrez votre invite de commande dans le dossier destination en tapant cmd dans la barre de recherche en haut comme montrée ci-dessous :



6 - Ouverture de l'invite de commande dans un dossier donné

Votre invite de commande s'ouvrira dans le dossier

Pour lancer le script exécutez la commande suivante

python runThis.py source.r

Le code qui est dans le fichier source.R se compilera :

```
C:\Users\TRETEC\Desktop\Projet Compil>python runThis.py source.r
Syntax error at token COMMENT
Syntax error at token NEWLINE
===QUADRUPLETS :===
['=', 1, None, 'A']
['=', 1, None, 'B']
['=', 6, None, 'C']
['>', 'C', 5, 'TEMPORARY_1']
['BZ', 7, 'TEMPORARY_1', None]
['=', 9, None, 'X']
['BR', 8, None, None]
['=', 10, None, 'X']
['END', '', '', '']
================================
================================
```

PARTIE II : Analyse lexicale

Comme dit précédemment, le but de cette analyse est de créer un Lexer qui permettra de diviser un code source en petites partie qu'on appelle Token, chaque Token représente une entité lexicale telle qu'un entier, l'identifiant d'une variable, le symbole d'une opération arithmétique ou bien un mot-clé réservé du langage tel que le WHILE ou FOR.

Afin d'implémenter cette analyse, nous allons utiliser principalement le module Lex de la librairie PLY.

On commencer avec l'initialisation et la construction du Lexer

```
#Init. Lexer
lexer = lex.lex()
```

7 - Construction du Lexer

Ensuite, nous entamons la définition et la création d'une liste contenant des chaines de caractères qui représentent tous les noms des token possible de notre langage pouvant être engendré par notre Lexer sauf pour les mots-clés qu'on verra plus tard.

```
tokens = [
    "LPAREN","RPAREN",
    "LACCO","RACCO",
    "LBRACKET","RBRACKET",
    "COMMA",
    "COLON",
    "EQUAL",
    "ADD","SUB","MUL","DIV","MOD",
    "SUP","INF","SUPEQUAL","INFEQUAL","EEQUAL","NOTEQUAL",
    "numeric_value","integer_value","character_value",
    "IF","ELSE",
    "FOR","IN",
    "IDF",
    "NEWLINE",
    "OR","AND",
    "WHILE",
    "NUMERIC","INTEGER","CHARACTER","LOGICAL",
    "TRUE","FALSE",
    "INC","DEC",
    "COMMENT",
]
```

8 - Création des tokens

Pour les mots-clés, nous avons opté pour l'utilisation d'une nouvelle structure : le dictionnaire à la place d'une simple liste, cette structure est parfaitement adéquate au besoin ou la clé est le mot écrit dans le code et la valeur est le token générée

```
#Dictionnaire pour les MR
#Mot1 : Token_Mot1
reserved = {
    "if": "IF",
    "else": "ELSE",
    "for": "FOR",
    "in": "IN",
    "or": "OR",
    "and": "AND",
    "while": "WHILE",
    "NUMERIC": "NUMERIC",
    "INTEGER": "INTEGER",
    "CHARACTER": "CHARACTER",
    "LOGICAL": "LOGICAL",
}
```

9 - Création des mots réservés

Toutes les valeurs de la liste et les valeurs du dictionnaire seront mis dans une liste qu'on appellera tout simplement tokens :

```
#MR + Tokens
tokens = list(reserved.values())+tokens
```

10 - Création de toute la liste des tokes

Nous utiliserons la libraire «re» (Regular Expression) pour l'écriture des expression régulière de chaque token avec un préfix t_ (t pour Token) de la forme t_NOMDUTOKEN imposée par le module PLY.lex, on définira l'expression régulière par une chaine de caractère précédée par un « r »

```
t_LPAREN = r"\("
t_RPAREN = r"\)"
t_LACCO = r"\{"
t_RACCO = r"\}"
t_ADD = r"\+"
t_SUB = r"\-"
t_MUL = r"\*"
t_DIV = r"/"
t_MOD = r"%"
t_AFFECT = r"="
t_INF = r"<"
t_SUP = r">"
t_RBRACKET=r"]"
t_LBRACKET=r"\["
t_COMMA=r","
t_COLON = r":"
```

11 - Définition des tokens

Nous ferons la même chose pour les mots-réservés sauf qu'au lieu de créer une variable, nous définiront une fonction afin de gérer l'attribue « Value » du token.

Elle prend en paramètre un token « t »

```
def t_INCR(t):
    r"\+\="
    t.value = "+="
    return t

def t_DECR(t):
    r"\-\="
    t.value = "-="
    return t

def t_NOTEQUAL(t):
    r"!="
    t.value = "!="
    return t
```

12 - Définition des mots réservés

La fonction «t_error » permet de retourner la ligne et la colonne d'une erreur lexicale lors de la détection d'un caractère ne possédant pas de token.

```
#Retourne le position de l'erreur
def t_error(t):
    print("Lexical Error : Line: {}  Column:{}".format(t.lineno, col(t)))
    t.lexer.skip(1)


#Init  Lexer
```

13 - Définition de la fonction d'erreur du Lexer

Pour la gestion des commentaires et des espaces, nous utiliserons «t_ignore_ » et leurs expressions régulières.

```
t_ignore_BLANK   = " \t"
t_ignore_COMMENT   = r"\#.*"
```

14 - Définition des caractères à ignorer

Vu qu'on a un Lexer déjà construit, nous pouvons commencer à transformer le « code » en Tokens, deux fonction du module Lex sont appeler, la première est lexer.input (code) ou code est le code source que nous voulons analyser, cette fonction permet de remettre le Lexer a zéro et stocker une nouvelle chaine de caractères en entrée. La seconde est la fonction lexer.token() qui permet de vérifier le token suivant , elle retourne vrai en cas de succès et faux dans le cas d'un caractère illégal.

Nous allons vérifier toutes les entités lexical du code, lexer.token() retourne vrai alors nous allons ajouter ce token dans une liste sinon nous allons générer une erreur.

```
def lexer(source_code):
    global line_start
    line_start = -1
    lexer_.input(source_code)

    output_tokens = []
    while True:
        token = lexer_.token()
        if not token:
            break
        output_tokens.append({"Type": token.type, "Value": token.value, "Line": token.lineno, "Col":  col(token)})

    return output_tokens
```

15 - Création du Lexer

PARTIE III : Analyse syntaxico-sémantique

Afin d'implémenter ce Parser, nous allons utiliser principalement le module Yacc de la librairie PLY.

On commencer avec l'initialisation et la construction du Parser

```
parser=yacc.yacc()
parser.parse(s)
```

16 - Création du Parser

L'analyse syntaxique ou le parsing est le processus d'analyse d'une chaîne de symboles et pour implémenter l'analyseur syntaxico-sémantique, il va falloir écrire la grammaire qui génère le langage défini ci-dessus.

La grammaire associée doit être LALR. En effet l'outil BISON est un analyseur ascendant qui opère sur des grammaires LALR.

Yacc utilise une technique d'analyse connue sous le nom de LR-parsing. L'analyse LR est une technique ascendante qui tente de reconnaître le côté droit de diverses règles de grammaire. Chaque fois qu'un côté droit valide est trouvé dans l'entrée, le code d'action approprié est déclenché et les symboles grammaticaux sont remplacés par le symbole grammatical du côté gauche.

```
def p_starter(p):
    '''
    STARTER : BLOC_INST
    '''

def p_bloc_inst(p):
    '''
    BLOC_INST : BLOC_INST INST NEWLINE
              | INST NEWLINE
              | IF_CLAUSE
              | WHILE_BLOCK
              | FOR_BLOCK
              | BLOC_INST IF_CLAUSE
              | BLOC_INST WHILE_BLOCK
              | BLOC_INST FOR_BLOCK
    '''

def p_inst(p):
    '''
    INST : DECLARATION
         | AFFECTATION
         | AFFECTATION_COND
    '''
```

17 - Implémentation de la grammaire LALR

Chaque nom de règle est une fonction avec le préfix p_NOM_DE_LA_REGLE, on définit la règle dans le docstring de la fonction.

Chaque fonction prend un paramètre p qui est un tuple tel que dans la règle suivante :

$$A \rightarrow B\ C\ D$$

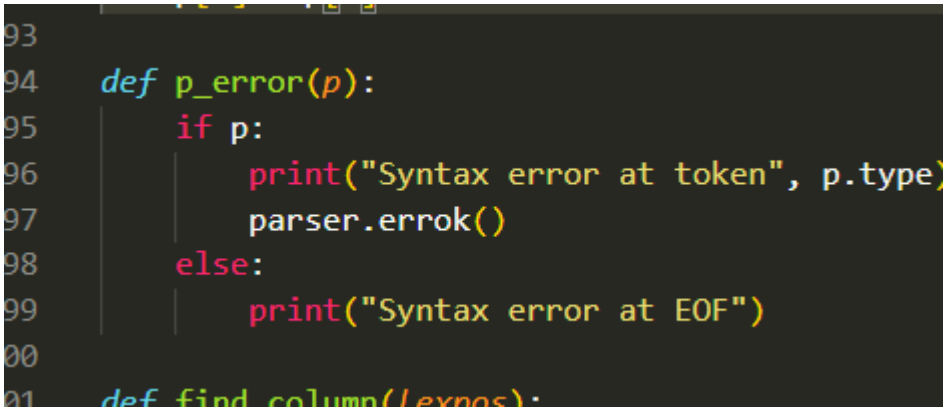
$$p[0] = A$$

$$p[1] = B$$

$$p[2] = C$$

$$p[3] = D$$

La première règle définie dans la spécification Yacc détermine le symbole de grammaire de départ. Chaque fois que la règle de départ est réduite par l'analyseur et qu'il n'y a plus d'entrée disponible, l'analyse s'arrête et la valeur finale est renvoyée (cette valeur sera celle de la règle la plus haute placée dans p [0]).



18 - Définition de la fonction d'erreur du Parser

La fonction «p_error » permet de retourner la ligne et la colonne d'une erreur syntaxique

PARTIE IV : Table des symboles et quadruplets

Table des symboles :

Elle centralise les informations sémantiques des identificateurs d'un programme :

- pour une variable : nom, type, taille, etc.
- pour un type : nom, types de base, etc.
- pour un tableau : nom, dimension, etc.
- pour une fonction : type, nb de paramètres, etc.
- en fonction de la nature des identifiants : table des types, etc.
- en fonction des blocs du programme (portée des déclarations).

Un symbole a été implémenté en une classe « Symb » avec les attributs suivants :

- name : nom du symbole
- type : INTEGER, NUMERIC etc.
- category : Variable ou non variable
- size : 1 si c'est une variable simple, taille du tableau si c'est un tableau
- initialized : Vrai ou faux

```
class Symb(object):
    def __init__(self, name, type=None,category=None,size=1,initialized=False):
        self.name = name
        self.type = type
        self.category = category
        self.size=size
        self.initialized = initialized
```

19 - Définition de la classe Symbole

La table des symboles a été implémentée en une classe « TableSymb » avec les méthodes suivantes :

- __init__ : Constructeur
- Insert : insérer un symbole dans la table
- lookup : rechercher un symbole dans la table par son nom

```python
class TableSymb(object):
    def __init__(self):
        self._symbols = {}

    def insert(self, symbol):
        self._symbols[symbol.name] = symbol

    def lookup(self, name):
        symbol = self._symbols.get(name)
        return symbol

table_symbols = TableSymb()
```

20 - Définition de la table des symboles

En conclusion :

- Une table de symbole est une centralisation des informations rattachées aux
- identificateurs d'un programme.
- Elle doit être bien conçue pour des accès rapides.
- Elle peut être très compliquée et devient très vite une machine à gaz.
- Rien n'interdit d'avoir plusieurs tables de symboles, il faut simplement savoir les gérer.

Quadruplets :

Nous avons programmé la génération des quadruplets et du code intermédiaire nous-même due à l'absence d'outils dans la libraire PLY.

La génération des quadruplets c'est fait dans les p_ fonctions, le tout est stocker dans une liste.

```
def p_for_condition(p):
    '''
    FOR_CONDITION : FOR_START integer_value RPAREN NEWLINE
    '''
    pileAdresses.append([len(quadruple), "FORbge"])
    quadruple.append(['BGE', None, p[1], p[2]])
    p[0] = p[1]
```

21 - Création et définition d'un quadruplet

PARTIE V : Génération du code Assembleur

Une fois les quadruplets générés on se sert de ceux-ci, pour la traduction en assembleur, On utilisera la syntaxe ASMx86, la traduction se fait quadruplet par quadruplet en vérifiant le premier opérateur du quadruplet pour la traduire en l'expression correspondante, évidement le tout a été programme manuellement du a l'absence d'outils et de fonctions dans la libraire PLY.

```
def generateAssembleyCode(quadruple):
    '''
    for each quadruple : transalte a quadruple into an assembley line code
    '''
    toPrint = []
    toPrint.append("Section .DATA\n")
    for name, value in SymbolsTable._symbols.items():
        toPrint.append(name+" "*declarationTypeVar(value.size))
    toPrint.append("\nSection .START\n")

    sectionStart = []
    labels = []
    i = 0
    label_counter = 1
    for quad in quadruple:
        if quad[0] == "=":
            if not variableType(quad[1]) and type(quad[1]) is str:
                sectionStart.append([i, "MOV {}, '{}'".format(variableType(quad[3]), quad[1])])
            else:
                sectionStart.append([i, "MOV {}, {}".format(variableType(quad[3]), variableType(quad[1]))])
        elif quad[0] in ["+", "-", "/", "*"]:
            if quad[1] == quad[3]:
                sectionStart.append([i, "MOV RAX, {}".format(variableType(quad[3]))])
                if quad[2] == 1:
                    if quad[0] == "-":
                        sectionStart.append([i, "DEC RAX"])
                    elif quad[0] == "+":
                        sectionStart.append([i, "INC RAX"])
                else:
                    sectionStart.append([i, "{} RAX, {}".format(operationToToken(quad[0]), variableType(quad[2]))])
                sectionStart.append([i, "MOV {}, RAX".format(variableType(quad[3]))])
            else:
                sectionStart.append([i, "MOV {}, {}".format(variableType(quad[3]), quad[1])])
```

```
            sectionStart.append([i, "MOV RAX, {}".format(variableType(quad[3]))])
            if quad[2] == 1:
                if quad[0] == "-":
                    sectionStart.append([i, "DEC RAX"])
                elif quad[0] == "+":
                    sectionStart.append([i, "INC RAX"])
            else:
                sectionStart.append([i, "{} RAX, {}".format(operationToToken(quad[0]), variableType(quad[
            sectionStart.append([i, "MOV {}, RAX".format(variableType(quad[3]))])
        else:
            sectionStart.append([i, "MOV {}, {}".format(variableType(quad[3]), quad[1])])
            sectionStart.append([i, "{} {}, {}".format(operationToToken(quad[0]), variableType(quad[3]),
    elif quad[0] in ["!=", "==", ">=", "<=", "<", ">"]:
        sectionStart.append([i, "MOV {}, {}".format(variableType(quad[3]), variableType(quad[1]))])
        sectionStart.append([i, "SUB {}, {}".format(variableType(quad[3]), variableType(quad[2]))])
        sectionStart.append([i, "CMP {}, 0".format(variableType(quad[3]))])
        sectionStart.append([i, "{} .labelLocation_{}".format(comparToJump(quad[0]), label_counter)])
        sectionStart.append([i, "MOV {}, 0".format(variableType(quad[3]))])
        sectionStart.append([i, "JMP .labelLocation_{}".format(label_counter+1)])
        sectionStart.append([i, ".labelLocation_{}".format(label_counter)])
        sectionStart.append([i, "MOV {}, 1".format(variableType(quad[3]))])
        sectionStart.append([i, ".labelLocation_{}".format(label_counter+1)])
        label_counter += 2
    elif quad[0] in ["AND", "OR"]:
        sectionStart.append([i, "MOV {}, {}".format(variableType(quad[3]), variableType(quad[1]))])
        sectionStart.append([i, "{} {}, {}".format(quad[0].lower(), variableType(quad[3]), variableType(
    elif quad[0] == "BZ":
        sectionStart.append([i, "CMP {}, 0".format(variableType(quad[2]))])
        sectionStart.append([i, "{} .labelLocation_{}".format(jumpQuad(quad[0]), label_counter)])
        labels.append([quad[1], label_counter])
        label_counter += 1
    elif quad[0] == "BGE":
        sectionStart.append([i, "CMP {}, {}".format(variableType(quad[2]), variableType(quad[3]))])
        sectionStart.append([i, "{} .labelLocation_{}".format(jumpQuad(quad[0]), label_counter)])
        labels.append([quad[1], label_counter])
        label_counter += 1
    elif quad[0] == "BR":
        sectionStart.append([i, "JMP .labelLocation_{}".format(label_counter)])
        labels.append([quad[1], label_counter])
```

```
    elif quad[0] == "BGE":
        sectionStart.append([i, "CMP {}, {}".format(variableType(quad[2]), variableType(quad[3]))])
        sectionStart.append([i, "{} .labelLocation_{}".format(jumpQuad(quad[0]), label_counter)])
        labels.append([quad[1], label_counter])
        label_counter += 1
    elif quad[0] == "BR":
        sectionStart.append([i, "JMP .labelLocation_{}".format(label_counter)])
        labels.append([quad[1], label_counter])
        label_counter += 1
    i += 1
sectionStart.append([sectionStart[-1][0]+1, "END"])

for inst in sectionStart:
    to_delete = []
    for k, label in enumerate(labels):
        if label[0] == inst[0]:
            toPrint.append(".labelLocation_{}".format(label[1]))
            to_delete.append(k)
    labels = [j for i, j in enumerate(labels) if i not in to_delete]

    if inst[1] != "END":
        toPrint.append(inst[1])

return toPrint
```

Dans le cas de l'exemple suivant, voici le code assembleur et les quadruplets générées

```
INTEGER A = 1
INTEGER B = 5
INTEGER X=0

X = (A>5,9,10)

IF(X==10)
{
    A=2
}
ELSE IF (X==5)
{
    A =3
}
ELSE
{
    A=A+B
    A=A*2
    B=5
}
```

```
===QUADRUPLETS :===
['=', 1, None, 'A']
['=', 5, None, 'B']
['=', 0, None, 'X']
['>', 'A', 5, 'TEMPORARY_1']
['BZ', 7, 'TEMPORARY_1', None]
['=', 9, None, 'X']
['BR', 8, None, None]
['=', 10, None, 'X']
['==', 'X', 10, 'TEMPORARY_2']
['BZ', 12, 'TEMPORARY_2', None]
['=', 2, None, 'A']
['BR', 21, None, None]
['==', 'X', 5, 'TEMPORARY_3']
['BZ', 16, 'TEMPORARY_3', None]
['=', 3, None, 'A']
['BR', 21, None, None]
['+', 'A', 'B', 'TEMPORARY_4']
['=', 'TEMPORARY_4', None, 'A']
['*', 'A', 2, 'TEMPORARY_5']
['=', 'TEMPORARY_5', None, 'A']
['=', 5, None, 'B']
['END', '', '', '']
================================
================================
```

```
Section .DATA

A DB
B DB
X DB

Section .START

MOV [A], 1
MOV [B], 5
MOV [X], 0
MOV r1, [A]
SUB r1, 5
CMP r1, 0
JG .labelLocation_1
MOV r1, 0
JMP .labelLocation_2
.labelLocation_1
MOV r1, 1
.labelLocation_2
CMP r1, 0
JZ .labelLocation_3
MOV [X], 9
JMP .labelLocation_4
.labelLocation_3
MOV [X], 10
.labelLocation_4
MOV r2, [X]
SUB r2, 10
CMP r2, 0
JZ .labelLocation_5
MOV r2, 0
JMP .labelLocation_6
.labelLocation_5
MOV r2, 1
.labelLocation_6
CMP r2, 0
JZ .labelLocation_7
MOV [A], 2
JMP .labelLocation_8
```

```
.labelLocation_7
MOV r3, [X]
SUB r3, 5
CMP r3, 0
JZ .labelLocation_9
MOV r3, 0
JMP .labelLocation_
.labelLocation_9
MOV r3, 1
.labelLocation_10
CMP r3, 0
JZ .labelLocation_1
MOV [A], 3
JMP .labelLocation_
.labelLocation_11
MOV r4, A
ADD r4, [B]
MOV [A], r4
MOV r5, A
MUL r5, 2
MOV [A], r5
MOV [B], 5
.labelLocation_8
.labelLocation_12
```