Réalisation d'un compilateur
L3 INFO 2020/2021

Le but est de réaliser un compilateur d'un langage de type pascal (appelé dans la suite L3Lang)
Le projet se décomposera en 4 étapes :
1 : Définition du projet global et réalisation d'un interpréteur de code exécutable.
2 : Réalisation de l'interface d'entrée et de l'analyseur lexicale associé au langage L3Lang.
3 : Réalisation de l'analyseur syntaxique et l'analyse sémantique associé au langage L3Lang.
4 : Réalisation de l'étape permettant la génération du code exécutable

Etape N°1 : Interpréteur de code exécutable

Le but de l'étape N°1 est d'écrire un interpréteur de code exécutable à partir du jeu d'instructions vues en cours (appelé dans la suite MACH):

ADD	additionne le sous-sommet de pile et le sommet, laisse le résultat au sommet (idem pour SUB, MUL, DIV)
EQL	laisse 1 au sommet de pile si sous-sommet = sommet, 0 sinon (idem pour NEQ, GTR, LSS, GEQ, LEQ)
PRN	imprime le sommet, dépile
INN	lit un entier, le stocke à l'adresse trouvée au sommet de pile, dépile
INT c	incrémente de la constante c le pointeur de pile (la constante c peut être négative)
LDI v	empile la valeur v
LDA a	empile l'adresse a
LDV	remplace le sommet par la valeur trouvée à l'adresse indiquée par le sommet (déréférence)
STO	stocke la valeur au sommet à l'adresse indiquée par le sous-sommet, dépile 2 fois
BRN i	branchement inconditionnel à l'instruction i
BZE i	branchement à l'instruction i si le sommet = 0, dépile
HLT	halte

L'interpréteur consiste en un logiciel permettant d'exécuter les instructions générées par le compilateur que vous devez écrire.

Ces instructions sont stockées dans un tableau que nous appellerons P-CODE.

L'exécution est réalisée à l'aide d'une pile appelé MEM ; le pointeur de pile s'appellera SP.

La structure de la mémoire en pile facilite la compilation de langages de haut niveau autorisant la récursivité. Les éléments de la pile sont des entiers : les adresses sont des adresses absolues dans la pile.

L'étape N°1 se décomposera en 3 parties :

1. Réfléchir aux structures de données permettant de représenter la pile de la machine et le pointeur de pile.
2. Réfléchir à la structure de données permettant de stocker instructions

3. Ecrire le programme qui permet d'exécuter les instructions contenues dans le tableau

P-CODE à l'aide de la pile mémoire. On utilisera une variable PC qui représente le pointeur d'instruction (adresse de l'instruction du tableau P-CODE devant être traitée).
On utilisera aussi la variable INST pour désigner l'instruction en cours de traitement.

Etaope N°2 : Réalisation de l'interface d'entrée et de l'analyseur lexicale associé au langage L3Lang

L3Lang - *Syntaxte du langage à implémenter :*

```
PROGRAM ::= program ID ; BLOCK .
BLOCK ::= CONSTS VARS INSTS
CONSTS ::= const ID = NUM ; { ID = NUM ; } | ε
VARS ::= var ID { , ID } ; | ε
INSTS ::= begin INST { ; INST } end
INST ::= INSTS | AFFEC | SI | TANTQUE | ECRIRE | LIRE | ε
AFFEC ::= ID := EXPR
SI ::= if COND then INST
TANTQUE ::= while COND do INST
ECRIRE ::= write ( EXPR { , EXPR } )
LIRE ::= read ( ID { , ID } )
COND ::= EXPR RELOP EXPR
RELOP ::= = | <> | < | > | <= | >=
EXPR ::= TERM { ADDOP TERM }
ADDOP ::= + | -
TERM ::= FACT { MULOP FACT }
MULOP ::= * | /
FACT ::= ID | NUM | ( EXPR )
```

Certains non-terminaux ne sont pas décrits par cette grammaire. Il s'agit des non-terminaux pris en charge par l'analyse lexicale :
-ID représente les identificateurs, c'est-à-dire toute suite de lettres ou chiffres commençant par une lettre et qui ne représente pas un mot clé (qui sont les terminaux présents dans la grammaire) ;
-NUM représente les constantes numériques, c'est-à-dire toute suite de chiffres.

L'étape N°2 consiste à écrire la procédure analyse-lexicale qui teste si la syntaxe du langage est bien respectée (voir les algorithmes définis en cours).

Etape N°3
Analyse sémantique et gestion des erreurs

Le but de l'étape N°3 est de transformer le programme déjà écrit lors de l'étape N°2 qui réalise l'analyse lexicale et syntaxique afin de : (i) prendre en compte l'analyse sémantique et (ii) réaliser la gestion des erreurs.
Pour améliorer l'analyse lexicale et syntaxique, il faut vérifier que les identificateurs utilisés sont bien déclarés. Pour cela, il est nécessaire de mémoriser les identificateurs déclarés pour tester la déclaration ou non des identificateurs utilisés. Cette mémorisation se fait dans une *table des symboles*.
L'analyse sémantique consiste à gérer la table de symbole qui représente l'ensemble des identificateurs utilisés dans un programme. Cette table est désignée par la suite par la variable TABLESYM.
Dans cette table, on associe à chaque entrée (chaque identificateur) toutes les informations connues sur lui, pour le moment :

- sa forme textuelle (son nom) ;
- sa classe, à savoir s'il désigne un programme, une constante, ou une variable.
- L'adresse de l'identificateur en mémoire

C'est le rôle du compilateur d'*allouer* ces symboles en mémoire. à chaque symbole, le compilateur doit associer un emplacement mémoire dont la taille dépend du type du symbole.

Une manière simple et naturelle de faire est de choisir les adresses au fur et à mesure de l'analyse des déclarations en incrémentant un *offset* qui indique la place occupée par les déclarations précédentes (variable OFFSET).
En programmation algorithmique la table serait déclarée ainsi :

```
array [TABLEINDEX] of record
                        NOM : Chaine;
                        CLASSE : CLASSES ;
                        ADRESSE : integer
                          end ;
```

Le type CLASSES représente les différentes classes d'identificateurs rencontrés : programme, constante, variable.
Le champ ADRESSE représente l'adresse mémoire de la variable ou la constante dans la pile d'exécution MEM qui a été utilisée dans le TP N°1.
La table des symboles est manipulée par deux fonctions :

ENTRERSYM

Ajoute le token TOKEN dans la table des symboles avec la classe passée en paramètre ; Elle gére de plus le champ adresse ; le champ ADRESSE n'est utilisé que pour les variables ; néanmoins, on l'utilisera pour stocker la valeur des constantes ;

CHERCHERSYM

Recherche l'identificateur TOKEN dans la table des symboles parmi les classes permises passées en paramètre et retourne l'index de l'entrée correspondante dans la table des symboles, ou s'arrête sur ERREUR ;

Il faut ensuite modifier les procédures écrites pour réaliser la gestion des erreurs selon l'approche vue en cours.

Les modifications de l'étape N°3 doivent permettre de gérer la table des symboles (allocation des données) et de gérer les erreurs.

Etape N°4
Génération de code

Une fois l'allocation des données réalisée, il est nécessaire de réserver l'emplacement suffisant dans la pile P-Code. Cette réservation est faite lors de l'analyse d'un BLOCK par la génération d'une instruction P-Code INST :

```
procedure BLOCK ;
begin
   OFFSET := 0 ;
   if TOKEN = CONST_TOKEN then CONSTS ;
   if TOKEN = VAR_TOKEN then VARS ;
   GENERER2 (INC, OFFSET) ;
   INSTS
end ;
```

Lors de la terminaison de l'analyse d'un programme, il est nécessaire de générer une instruction P-Code d'arrêt du programme, HLT :

```
procedure PROGRAM ;
begin
   TESTE (PROGRAM_TOKEN) ;
   TESTE_ET_ENTRE (ID_TOKEN, PROGRAMME) ;
   TEST (PT_VIRG_TOKEN) ;
   BLOCK ;
   GENERER1 (HLT) ;
   if TOKEN <> POINT_TOKEN then ERREUR
end ;
```

L'analyse de l'expression a + b, EXPR va appeler TERM deux fois, une fois pour analyser a et une fois pour analyser b ; l'analyse du + se fait au niveau de EXPR. Si les deux appels réussissent (c'est le cas dans notre exemple) et que le + est bien reconnu, EXPR réussit. Lorsque la génération est dirigée par la syntaxe, on adopte le même raisonnement : lorsqu'une procédure se termine, on considère que la génération des phrases analysées par les procédures appelées est terminée. Il ne reste plus qu'à générer le code pour l'addition, c'est-à-dire simplement l'instruction P-Code ADD. Il faut ensuite modifier les programmes qui gèrent les éléments de la grammaire. On commence donc par le génération des facteurs pour lesquels on laisse sur la pile P-Code une valeur : **Génération des facteurs (**FACT**)**

Génération des termes (TERM)

Un terme laisse sur la pile P-Code la valeur du terme. Il est nécessaire de mémoriser le token correspondant à l'opération avant l'analyse de l'opérande gauche du terme (variable OP) :

```
procedure TERM ;
var OP : TOKENS ;
begin
  FACT ;
  while TOKEN in [MULT_TOKEN, DIV_TOKEN] do
    begin
      OP := TOKEN ; (* memorise l'operation *)
      NEXT_TOKEN ;
      FACT ;
      if OP = MUL_TOKEN
        then GENERER1 (MUL)
        else GENERER1 (DIV)
    end
end ;
```

Génération des expressions (EXPR)

L'analyse d'une expression laisse aussi une valeur sur la pile P-Code ; le code est similaire à celui de l'analyse des termes

Génération des conditions

La génération des conditions (procédure COND) est calquée sur celle des expressions

Génération des instructions simples

Il n'y pas de code à générer lors de l'analyse du bloc d'instructions (INSTS) ou d'une instruction (INST). On détaille ici la génération à réaliser lors de l'analyse d'une instruction d'affectation et des instructions d'entrée/sortie.

Génération d'une affectation

Une affectation A := expression est générée suivant le modèle

LDA <adresse de A>	empile l'adresse de A
<code>	empile la valeur de l'expression
STO	stocke la valeur de l'expression dans A

Le P-Code <code> dépose la valeur de expression sur la pile ; il est généré lors de l'analyse de expression par l'appel EXPR. On a donc :

```
procedure AFFEC ;
begin
   TESTE_ET_CHERCHE (ID_TOKEN, VARIABLE) ;
   GENERER2 (LDA, TABLESYM [PLACESYM]. ADRESSE) ;
   TESTE (AFFEC_TOKEN) ;
   EXPR ;
   GENERER1 (STO)
end ;
```

Génération des instructions d'entrée/sortie

Le code à générer pour une instruction d'écriture telle ECRIRE (e1, e2, e3) est le suivant :

<code1>	empile la valeur de l'expression e1
PRN	imprime cette valeur
<code2>	empile la valeur de l'expression e2
PRN	imprime cette valeur
<code3>	empile la valeur de l'expression e3
PRN	imprime cette valeur

Les P-Codes <code1>, <code2> et <code3> sont générés lors de l'analyse des expressions e1, e2 et e3 par des appels à EXPR. On modifiera la procédure ECRIRE en conséquence.

Le code à générer pour une instruction de lecture telle LIRE(v1, v2, v3) est le suivant :

LDA <adresse de v1>	empile l'adresse de la variable v1
INN	lit un entier, le stocke dans v1
LDA <adresse de v2>	empile l'adresse de la variable v2
INN	lit un entier, le stocke dans v2
LDA <adresse de v3>	empile l'adresse de la variable v3
INN	lit un entier, le stocke dans v3

Modifier la procédure LIRE en conséquence.

Procédures de génération de P-Code pour les instructions IF et WHILE

Suivre la méthodologie vue en cours afin d'utiliser les branchements conditionnels dus aux instructions de type IF et WHILE.

Procédures de génération de P-Code

Les fonctions de génération de code GENERER1 et GENERER2 s'écrivent alors simplement (voir cours)

Questions :

Vous devez implémenter le compilateur du langage L3Lang à l'aide du jeu d'instructions MACH.

a. Pour cela vous commencerez par l'étape N°1 : Réalisation de l'interpréteur de code lié au jeu d'instructions MACH
b. Implémentation des programmes liés à l'étape N°2 : Réalisation de l'analyseur lexical et syntaxique de langage L3Lang
c. Transformation du code que vous avez écrit lors de l'étape 2 afin de prendre en compte la gestion de la table des symboles et la gestion des erreurs (Etape 3).
d. Transformation du code écrit lors de l'étape 3 afin de générer le code (à l'aide du jeu d'instructions MACH) pour un programme écrit dans le langage L3Lang.

Vous veillerez à tester soigneusement le code que vous écrirez à chacune des 4 étapes.

Remarques :

1. Vous pouvez choisir le langage dans lequel vous aller écrire le compilateur
2. Vous devez vous appuyer sur les algorithmes vus en cours pour réaliser le compilateur.