UE Compilation - Projet

1. Définition du langage

Le langage **Zoot** est un langage de programmation typé rudimentaire incluant les variables entières et booléennes, les structures de contrôle élémentaires et les fonctions.

Grammaire

L'axiome de la grammaire est **PROGRAMME**. Les non terminaux sont écrits en majuscules ; les terminaux qui ne sont pas des symboles sont écrits en minuscules et soulignés. La notation $\{\alpha\}*$ signifie que la séquence α peut être répétée un nombre quelconque de fois, éventuellement nul ; la notation $\{\alpha\}+$ signifie que la séquence α peut être répétée un nombre quelconque de fois, jamais nul ; la notation $\{\alpha\}$ signifie que la séquence α est optionnelle.

En **Zoot**, les commentaires commencent par la séquence de caractères // et se terminent à la fin de la ligne.

Les terminaux génériques sont constitués de la façon suivante :

- **csteEntiere** est une suite non vide de chiffres décimaux ;
- **idf** est une suite non vide de lettres et de chiffres commençant par une lettre, sans limitation de taille ; la différence entre majuscules et minuscules est significative.

Les mots clés sont écrits en minuscules (exactement comme dans la grammaire) et sont réservés. Par exemple, un programme **Zoot** peut contenir un identificateur SI qui ne sera pas confondu avec le mot clé réservé **si**.

PROGRAMME	→	**variables** { VARIABLE } + **fonctions** { FONCTION } + **debut** { INSTRUCTION } + **fin**
TYPE	→	**entier** | **booleen**
VARIABLE	→	**TYPE** **idf** ;
FONCTION	→	**TYPE** **idf** PARAMETRES **variables** { VARIABLE } * **debut** { INSTRUCTION } + **fin**
PARAMETRES	→	**(** { VARIABLE } * **)**
INSTRUCTION	→	AFFECT | BOUCLE | CONDITION | ECRIRE | RETOURNE
AFFECT	→	**idf** = EXP ;
BOUCLE	→	**repeter** { INSTRUCTION } + **jusqua** EXP **finrepeter**
CONDITION	→	**si** EXP **alors** { INSTRUCTION } * { **sinon** { INSTRUCTION } + } **finsi**
RETOURNE	→	**retourne** EXP ;
ECRIRE	→	**ecrire** EXP ;
EXP	→	**idf** **(** PAR_EFF **)** | **csteEntiere** | **idf** | **vrai** | **faux**
		| **(** EXP **)** | **non** EXP | **-** EXP | EXP OPER EXP
OPER	→	**+** | ***** | **<** | **==** | **!=** | **et** | **ou**
PAR_EFF	→	{ EXP { **,** EXP } * }

Sémantique

1. Un programme est constitué de déclarations de variables, de fonctions et d'instructions. À l'exécution du programme, chacune de ces instructions est exécutée dans l'ordre d'écriture.

2. La portée d'une déclaration de variable ou de fonction est constituée de l'intégralité de la région qui la contient, moins les régions imbriquées où le même identificateur est déclaré dans le même espace des noms. Une région est délimitée par les mots clés **debut** et **fin**.

3. Les variables ne sont pas initialisées par défaut.

4. Les doubles déclarations de variables sont interdites. La surcharge des fonctions est autorisée, sous réserve de définir des profils différents, c'est-à-dire avec des nombres de paramètres différents. Une variable et une fonction ne peuvent pas porter le même nom.

5. Dans une affectation, les parties gauche et droite sont de même type.

6. Dans le corps d'une fonction, l'instruction **retourne** est obligatoire. Le type de l'expression est identique au type de retour de la fonction. L'exécution de cette instruction provoque l'arrêt de la fonction en retournant comme résultat la valeur de l'expression. L'instruction **retourne** ne peut pas se trouver en dehors du corps d'une fonction.

7. Dans une itération conditionnelle, l'expression est de type booléen ; elle est évaluée après exécution des instructions itérées.

8. Dans une instruction conditionnelle, l'expression est de type booléen.

9. La fonction prédéfinie **ecrire** écrit, soit un entier, soit un booléen, suivi d'un retour à la ligne. Les booléens sont écrits sous la forme **vrai** ou **faux**.

10. Les opérandes des opérateurs +, * et - unaire sont entiers, le résultat est entier. Les opérandes des opérateurs **et**, **ou** et **non** sont booléens. L'opérateur relationnel < est à opérandes entiers et résultat booléen, les opérateurs relationnels == et != sont à opérandes de même type et résultat booléen.

11. Les opérateurs binaires sont tous associatifs gauche-droite. Les priorités des opérateurs sont définies dans l'ordre strictement décroissant suivant : **( )** , **non** et - , * , + , < , == et != , **et** , **ou**.

12. Dans un appel de fonction, le compilateur choisit le profil de la fonction à partir du nombre de paramètres effectifs. Le passage des paramètres se fait par valeur.

2. Le compilateur Zoot

Le compilateur **Zoot** doit être développé en langage Java en utilisant les générateurs d'analyseurs **JFlex** et **JavaCup**. Il génère du code **MIPS**.

De sorte que les tests soient facilement automatisables, il est impératif de respecter les contraintes ci-dessous :

- le compilateur est livré dans l'archive **zoot.jar**. Son exécution nécessite exactement un argument : le nom du fichier contenant le texte du programme à compiler, suffixé **.zoot**. Elle provoque la compilation du texte et, si celle-ci se déroule sans erreur, crée un fichier de même préfixe, suffixé **.mips**, contenant le texte cible correspondant.
- l'exécution n'est pas conversationnelle ; en dehors du fichier **.mips** créé, elle doit laisser intact l'environnement de celui qui l'appelle.
- selon le cas, l'exécution produit sur la sortie standard l'un des résultats suivants (et rien d'autre …) :

  ERREUR LEXICALE : no ligne d'erreur : message d'erreur explicite
  ERREUR SYNTAXIQUE : no ligne d'erreur : message d'erreur explicite
  ERREUR SEMANTIQUE : no ligne d'erreur : message d'erreur explicite
  COMPILATION OK

- une erreur lexicale ou syntaxique stoppe la compilation du texte source ; toutes les erreurs sémantiques détectées doivent être signalées.

  Si une erreur se produit lors de l'exécution du code généré, l'exécution s'arrête immédiatement avec l'affichage du message ERREUR EXECUTION sur la sortie standard.

3. Déroulement du projet

Vous travaillerez en groupe de 2. Inscrivez rapidement vos noms dans la feuille prévue à cet effet sur Arche.

Chaque groupe aura un projet dédié sur le gitlab de l'UL. Doivent y être déposés les sources du projet, les sources **Zoot** utilisés pour faire les tests du compilateur ainsi que le diagramme de classes, au fur et à mesure de l'évolution des versions.

La réalisation de ce compilateur doit se faire par noyaux successifs du langage. Vous trouverez ci-dessous la composition des différents noyaux à compiler. Lorsque le compilateur d'un noyau est terminé et complètement testé, vous pouvez commencer le développement du noyau suivant, sans attendre. Et inversement, il est inutile de commencer un nouveau noyau tant que le précédent n'est pas totalement testé. Les dates limites indiquées dans le tableau ci-dessous sont les dates auxquelles les tests automatiques de chaque version seront effectués.

Grammaire Zoot0 - que des affichages

Le compilateur ne traite que des programmes avec des instructions d'écriture ; on se contente de l'extrait de grammaire ci-dessous.

PROGRAMME → **debut** { INSTRUCTION } + **fin**

INSTRUCTION → ECRIRE

ECRIRE → **ecrire** EXP **;**

EXP → **csteEntiere**

Noyaux du langage et dates limites des dépôts

Noyau	Constructions reconnues par le compilateur	Date limite de dépôt sur Arche de l'archive exécutable zoot.jar
Zoot0	Instruction d'écriture Commentaires	**vendredi 28 janvier - à la fin de la séance - 10h00**
Zoot1	Déclaration de variables (entières/booléennes) Affectation Expression réduite à une constante ou une variable	**Mardi 22 février - 20h**
Zoot2	Fonction sans paramètre ni variable locale Expression réduite à une constante ou une variable ou un appel de fonction	**Mardi 8 mars - 20h**
Zoot3	Fonction avec paramètres et variables locales entières et booléennes	**Mardi 22 mars - 20h**
Zoot4	Expressions quelconques Instruction conditionnelle Instruction itérative	**Mardi 5 avril - 20h**