TELECOM Nancy Apprentis
15 Janvier 2020

Projet de compilation.

L'objectif de ce projet est d'écrire un traducteur du langage `LEAC` *(Langage Élémentaire Algorithmique pour la Compilation), dont les constructions sont proches de celles du langage Pascal. Ce traducteur produira en sortie du code C, code qui sera ensuite compilé avec gcc (par exemple).*

1 Réalisation du projet.

Vous travaillerez par groupes de 3 élèves. Si vous êtes amenés à former un binôme ou un monôme, nous en tiendrons compte lors de l'évaluation de votre projet.

Vous utiliserez l'outil ANTLR, générateur d'analyseur lexical et syntaxique *descendant*, interfacé avec le langage Java (ou Python pour M. Bouillon) pour les étapes d'analyse lexicale et syntaxique.
Votre compilateur doit signaler les erreurs lexicales, syntaxiques et sémantiques rencontrées. Lorsqu'une de ces erreurs est rencontrée, elle doit être signalée par un message explicite comprenant un numéro de ligne. Votre compilateur doit également poursuivre l'analyse après avoir signalé une erreur sémantique.

Vous serez évalué en fin de projet lors d'une soutenance au cours de laquelle vous présenterez le fonctionnement de votre traducteur.

Déroulement et dates à retenir.

Définition complète de la grammaire du langage.

Vous définirez la grammaire du langage et la soumettrez à ANTLR afin qu'il génère l'analyseur syntaxique descendant. Bien sûr, l'étape d'analyse lexicale est réalisée parallèlement à l'analyse syntaxique. Vous aurez testé votre grammaire sur des exemples (au moins 15 exemples dans un fichier) variés de programmes écrits en `LEAC` (avec et sans erreur lexicales et syntaxiques).

Évaluation de la grammaire.

Vous ferez une démonstration de l'analyse lexicale et syntaxique sur votre grammaire.

Évaluation de l'AST et de la TDS.

Vous montrerez ces deux structures sur des exemples de programmes de test (une visualisation, même sommaire, de ces structures est indispensable).

Fin du projet.

Vous continuez votre projet en mettant en place la phase d'analyse sémantique (si ce n'est pas déjà fait) et la génération de code C. Pour cette dernière étape, vous veillerez à générer le code C de manière incrémentale, en commençant par les structures "simples" du langage.

La démonstration.

Il est impératif que vous ayez prévu des exemples de programmes permettant de tester votre projet et ses limites (ces exemples ne seront pas à écrire au moment de la démonstration...)

Pour finir...

Bien entendu, il est interdit de s'inspirer trop fortement du code d'un autre groupe ; vous pouvez discuter entre-vous sur les structures de données à mettre en place, sur certains points techniques à mettre en œuvre, etc... mais il est interdit de copier sur vos camarades.

La note finale (sur 20) de votre projet prend en compte les "notes" prises lors des différentes présentations intermédiaires.

La fin du projet est fixée au **20 Janvier 2020**, date à laquelle vous effectuerez une démonstration de votre projet.

2 Présentation du langage.

Aspects lexicaux.

Identificateurs : Un *identificateur* est composé des lettres de l'alphabet, majuscules ou minuscules, et des chiffres de 0 à 9, à l'exception de tout autre caractère.
Un identificateur commence obligatoirement par une lettre. Les majuscules et minuscules sont différenciées. Dans la suite du sujet, le symbole `IDF` sera utilisé comme synonyme d'identificateur.
Commentaires : Un commentaire peut apparaître n'importe où dans le texte source : les commentaires commencent par /* et se terminent par */. Ils ne sont pas imbriqués.

Le langage - aspects syntaxiques.

On donne ci-dessous la grammaire complète du langage `LEAC`. Dans cette grammaire, les non-terminaux sont en lettres minuscules et les terminaux en lettres majuscules. Les autres symboles, tels `( ) + -` etc sont aussi des symboles terminaux et sont écrits en caractères gras. Les mots-clés seront en minuscules et également en gras. Le terminal CSTE représente les constantes entières, booléennes (`true` ou `false`) ou chaînes de caractères (écrites entre guillemets). Le terminal IDF représente les identificateurs, cf. paragraphe précédent. Le type `void` sert uniquement à spécifier le type de la valeur retournée par une fonction dont le seul but est de causer un effet de bord (cette fonction est donc une procédure).

Le symbole | désigne l'alternative dans la grammaire et $\wedge$ le mot vide.

program	→	**program** IDF vardeclist fundeclist instr
vardeclist	→	$\wedge$ | varsuitdecl | varsuitdecl vardeclist
varsuitdecl	→	**var** identlist **:** typename **;**
identlist	→	IDF | IDF **,** identlist
typename	→	atomtype | arraytype
atomtype	→	**void** | **bool** | **int**
arraytype	→	**array [** rangelist **] of** atomtype
rangelist	→	CSTE **..** CSTE | CSTE **..** CSTE**,** rangelist
fundeclist	→	$\wedge$ | fundecl fundeclist
fundecl	→	**function** IDF **(** arglist **) :** atomtype vardeclist instr
arglist	→	$\wedge$ | arg | arg **,** arglist
arg	→	IDF **:** typename | **ref** IDF **:** typename
instr	→	**if** expr **then** instr | **if** expr **then** instr **else** instr
		| **while** expr **do** instr | lvalue = expr | **return** expr | **return**
		| IDF **(** exprlist **)** | IDF **( )** | **{** sequence **}** | **{ }**
		| **read** lvalue | **write** lvalue | **write** CSTE
sequence	→	instr **;** sequence | instr**;** | instr
lvalue	→	IDF | IDF **[** exprlist **]**
exprlist	→	expr | expr **,** exprlist
expr	→	CSTE | **(** expr **)** | expr opb expr | opun expr
		| IDF **(** exprlist **)** | IDF **( )** | IDF **[** exprlist **]** | IDF
opb	→	+ | - | * | / | ^ | < | <= | > | >= | == | != | and | or
opun	→	- | not

Le langage - aspects sémantiques.

Les fonctions seront systématiquement déclarées avant leur utilisation (pas de "prototype" comme en C).
Le mot-clé `ref` dans une fonction désigne un mode de passage des paramètres par adresse. En l'absence de ce mot-clé, le mode de passage des paramètres est le mode par valeur.

Portée des déclarations et visibilité des variables.
Des variables dites globales pourront être déclarées en début du programme : la portée de la déclaration de ces variables globales est tout le fichier et elles seront visibles dans toute la partie du fichier située après leur déclaration. Les variables définies dans un bloc sont visibles seulement dans ce bloc. Les paramètres d'une fonction sont visibles uniquement dans cette fonction.

Il n'y a qu'un seul espace des noms.

Entrées-Sorties.

Pour les entrées-sorties, on utilisera les opérations `read` et `write` qui réalisent respectivement la lecture à partir d'une entrée au clavier et l'écriture sur la sortie standard qu'est l'écran.

Exemples de programmes.

Le programme suivant est un exemple de programme simple écrit en `LEAC`.

```
/* Un  exemple de programme écrit en LEAC */
program essai
   var i, j, maximum: int;
   var Tval : array[-3..3, 0..5] of int;

   function maxTAB (t:  array[-3..3, 0..5] of int) : int
      var i, j, max: int;
      { i = -3;
        j = 0;
        max = t[-3, 0];
        while i <= 3 do
          {while j <= 5 do
             { if t[i, j] > max then max = t[i, j];
               j = j + 1
             }
             i = i + 1
          }
       return max
      }

    function theEnd () : void
       { write "that's all !" }

 { /* début du programme principal */
   i = -3;
   j = 0;
   maximum = t[-3, 0];
   while i <= 3 do
     {while j <= 5 do
        { read Tval[i, j];
          j = j + 1
        }
        i = i + 1
      }
   maximum = maxTAB(Tval);
   write maximum;
   theEnd()
}
```