Master Informatique **Compilation – J1IN7M13**
Université Bordeaux 1 **2012 – 2013**

Compilateur Léa – Projet à rendre le 21 décembre 2012

Le but de ce projet est de livrer à terme un compilateur pour un langage élémentaire algorithmique (Léa) que nous illustrons en Fig.2. Ce langage est destiné à enseigner la programmation dans les cursus de sciences humaines. Nous nous inspirerons des langages Pascal, Python, Ruby etc. avec cette exigence : les connaissances en programmation avancées ne sont pas exigées pour savoir programmer avec Léa, et les habitudes récentes de programmation objet seront conservées.

Le projet Léa est surtout un prétexte pour proposer un travail de compilation complet. Il ne vous sera d'aucune façon reproché dans ce travail

1. De ne pas avoir livré un compilateur exploitable dans l'état
2. D'avoir été associé à une équipe faible

Il est demandé à chacun de traiter les sections écrites en noir en s'aidant des explications données en TD. Il n'est pas exigé de traiter l'ensemble du projet, on pourra en faire une partie seulement.

Les parties en vert et en bleu sont de difficulté croissante, vous pouvez tenter de les faire.

1 Modalités pratiques

Ce projet est à réaliser en groupe de 4 personnes, pour le xx décembre 2012 au plus tard.
Chaque groupe est composé de :

1. Un(e) correspondant(e)
   C'est le porte-parole du projet, celui qui écrit sa documentation et assure sa cohérence.
2. Un(e) syntacticien(ne)
   Il écrit le code CUP/JFlex.
3. Un(e) sémanticien(ne)
   Il écrit le code intermédiaire
4. Un développeur
   Il écrit le code cible.

Chaque correspondant devra envoyer un courrier électronique au nom du groupe à (`frederique.carrere@labri.fr` ET `lionel.clement@labri.fr`)

- Dont l'objet est "DM Compilation"
- Signé de tous les prénoms et noms du groupe
- Contenant une archive `tar.bz2` avec les fichiers suivants :

1. `dm1.tar`, une archive tar, en excluant tout autre format d'archivage. Cette archive contiendra toutes les sources et toutes les données permettant la compilation et l'exécution du programme complet par simple appel à la commande `ant main`.
2. `Prénom-NomCorrespondant.pdf`, une explication de votre travail, sous la forme d'un texte en format .pdf en excluant tout autre format. Ce texte contiendra :
   (a) Un cahier des charges précisé
   (b) Un exposé des problèmes envisagés
   (c) Une explication des choix retenus

FIG. 1 – Exemple de code Léa

```
/* ********************************************
* Un exemple complet de code Léa
******************************************** */

// Définition des constantes
MAX = 500;

// Définition des types
fiche = struct {
        nom: string;
        age: int;
}

// Définition de la fonction main
function main(argn: int, argv: list of string): int
{
        // Définition des variables
        table: list of fiche;
        echange: bool;

        // Instructions
        for i in [0 .. MAX-1] {
                table[i].nom := "";
                table[i].age := random(100);
                }

        repeat {
                echange := false;
                for i in [0 .. MAX-2] {
                        if (T[j].age > T[j+1].age){
                                // Variable locale
                                aux: fiche;
                                aux := T[j];
                                T[j] := T[j+1];
                                T[j+1] := aux;
                                echange := true;
                        }
                }
        } while (echange);
return 0;
}
```

2 Léa

Le compilateur doit être écrit en CUP/JFlex/Java et doit produire un programme Java équivalent. La gestion de la mémoire doit être entièrement dynamique et les types complexes sont implémentés en utilisant les bibliothèques standards de Java.

3 Généralités sur le langage

Le langage est algorithmique procédural. Il utilise un typage statique explicite. C'est-à-dire que toutes les variables sont déclarées avec un type immuable durant l'exécution du code. La syntaxe objet est utilisée pour les fonctions membres de l'objet désigné. Par exemple on peut accéder à la fonction membre `length()` d'une variable `x` de type `string` ainsi : `x.length()`

3.1 Structure générale d'un programme Léa

1. Déclaration des modules requis
2. Déclaration des types et constantes
3. Déclaration des signatures des fonctions et procédures
4. Implémentation des fonctions et procédures dont la fonction **main**

3.2 Déclaration des constantes, des types et des fonctions

Une constante et un type n'ont qu'une seule déclaration possible pour l'ensemble du programme.

Les procédures et fonctions peuvent avoir des déclarations multiples en cas de surcharge.

Un programme doit contenir une déclaration de la fonction `main(int, list of string): int` qui correspond à l'implémentation de la procédure principale. La signature `procedure main();` est également acceptée (le programme retournera 0 et les arguments seront inaccessibles).

1. Déclaration de constante :

   $identificator\ =\ constant\,;$

   où *constant* est un littéral ou une constante complexe (expression de liste, d'ensemble, de tuple, etc constante)
2. Déclaration de type construit :

   $identificator\ =\ type\,;$

   où *type* peut faire référence à des types déjà déclarés
3. Déclaration de procédure :

   procedure $identificator\ \ (liste\ d'arguments)\ \ bloc$
4. Déclaration de fonction :

   function $identificator\ \ (liste\ d'arguments)\,:\ \ type\ \ bloc$
5. On peut déclarer la signature des fonctions et procédures avant leur implémentation comme on le fait en langage C :

   procedure $identificator\ \ (liste\ d'arguments)\ \ ;$
   function $identificator\ \ (liste\ d'arguments)\,:\ \ type\ \ ;$

3.3 Types

Les types suivants sont disponibles ($T, T_1, T_2, \ldots, T_k$ sont des types quelconques) :

- Types élémentaires `char`, `int`, `float`, `bool`, `string`
- Énumérés (vecteur de bits) `enum`
  On accède à l'un des bits par le nom d'un champs
- Listes (implémentation des suites mathématiques) `list of` T
- Tuples $(T_1, T_2, \ldots, T_k)$
- Structures `struct` $\{slot_1 : T_1; slot_2 : T_2, \ldots slot_k : T_k\}$
  On accède au champs $slot_i$ avec l'opérateur $.slot_i$
- Ensembles `set of` T
- Dictionnaires `map of` (T_1, T_2)
- Variables de type. Il conviendra de ne pas confondre avec les noms de type (un identificateur utilisé pour désigner un type), ni les types nommés (un identificateur utilisé comme type distingué pour nommer les slots des structures et des classes). Ces variables de type sont utiles pour écrire les fonctions polymorphes comme dans cet exemple-ci :

```
function plus_petit_element(l: list of 'x): 'x {
        min: 'x := l.first();
        for i in [0 .. l.len()] // [0 .. l.len()] est une
                                    liste d'entiers
                if (min > l[i])
                        min := l[i];
        return min;
}
```

- Classes
  Les classes contiennent
  - Les variables, procédures et fonctions de classe (notées *static*)
  - Les variables d'instance
  - Les constructeurs d'instance
  - Les opérateurs d'instance

Des opérateurs et fonctions membres sont accessibles pour chaque type. Nous en dressons la liste plus loin.

L'allocation mémoire est implicitement dynamique. C'est-à-dire que les variables sont dynamiquement allouées lors de l'exécution sans utilisation de `new`, `malloc` ou autre instruction d'allocation, de réallocation.

Dans le cas d'une structure réentrante `S`, l'allocation est `null` par défaut pour chaque champs de type `S`. D'une manière générale, la valeur d'une variable non déclarée est `null`, mais cela n'est jamais visible de l'utilisateur qui reçoit un message d'erreur bien avant le calcul du type. Une variable dont le type est marqué comme réentrant est de valeur `null`, sinon en langage cible, elle aura la valeur de l'objet correspondant.

Voici un exemple :

```
arbre = struct {
        left: arbre;
        right: arbre;
        etiq: int;
}

function parcours(root: arbre): string {
        s: string := "";
        if (root.left != null)
                s += parcours(root.left)
        if (root.right != null)
                s += parcours(root.right)
        s += root.etiq;
        return s;
        }

procedure main() {
        a: arbre; // allocation implicite de a
        a.etiq := 0;
        a.left.etiq := 1; // allocation implicite de a.left

        writeln (parcours(r));
}
```

3.4 Instructions et structures de contrôle

3.4.1 Affectation

Pour dissuader l'apprenant de confondre l'égalité avec l'affectation, nous utilisons le signe `:=`

Sont affectables, les variables, les éléments de type complexe (liste, ensemble, etc). Dans le cas de l'affectation d'un élément d'une suite mathématique, l'allocation dynamique (ou la réallocation) doit prévoir n^2 éléments disponibles où n est l'indice. En code cible java, on ne souciera pas de ce problème d'allocation, on utilisera par exemple immédiatement `ensureCapacity(int)` de `Arraylist` qui désaloue et réalloue dynamiquement.

AffectableExpr := *expr*;

On pourra associer l'affectation et la déclaration

variable: *type* := *expr*;

3.4.2 Appel de procédure

Si `foo` est une procédure préalablement déclarée, `foo(...)` est accessible comme instruction.

Remarque. Contrairement à de nombreux langages, une expression n'est pas une instruction ni l'inverse. Il n'est donc pas possible d'écrire `fact(7)` comme instruction où `fact` est une fonction ni une affectation comme expression `if (x := foo(i)) ...`

ProcedureName ($expr_1, expr_2, \ldots, expr_k$) ;

3.4.3 Entrées sorties

Les procédures `write(string)`, `writeln(string)` permettent un accès aux sorties standard. Ajoutons `read()` comme expression pour un accès à l'entrée standard (une chaîne de caractères terminée par `return`) Le type de ces fonctions et de leur argument est `string`.

3.4.4 Conditionnelles et choix multiples

Si ($I, I_1, I_2, \ldots I_k$ sont des instructions), les instructions conditionnelles suivantes sont admises :

if (*expr*) *I*
if (*expr*) I_1 **else** I_2

expr doit être du type `bool` contrairement à de nombreux langages

case *expr* **of**
$enum_slot_1 : I_1$
$enum_slot_2 : I_2$
...
$enum_slot_k : I_k$

expr doit être de type `enum`

3.4.5 Boucles

Si I est une instruction, les boucles suivantes sont admises :

1. boucle `while` et `repeat`

   while (*expr*) *I*
   repeat *I* **while** (*expr*) ;

   Comme pour les conditionnelles, *expr* doit être du type `bool`.

2. boucle `for`

   for *var* **in** *expr* *I*

   Expr est une expression de type
   – Liste
   – Ensemble
   – Dictionnaire
   – Intervalle
   – Chaîne de caractères
   – Énuméré
   La variable *var* est déclarée localement et son type se rapporte au type de *expr* (`char` si *expr* est un `string`, l'un des énumérés pour `enum`, `int` pour `interval`, élément pour la liste l'ensemble ou le dictionnaire).

exemples

```
{
k: map of (int, string);
s: string;
t: enum (ROUGE, BLEU, VERT);
for i in [1 .. 100] print i;
for i in [1, 2, 3] print i;
for i in {1, 2, 3} print i;
for i in k
        print(j.first().toString() + " => " + j.second();
for i in s
        print(l);
for i in t
        print(l.toString());
}
```

Techniquement, cette instruction `for` est équivalente à

```
{
uniqIterator: int := 0;
variable: typeofelement;
while (uniqIterator <= exp.size()) {
        variable := expr[uniqIterator];
        I
        uniqIterator++;
        }
}
```

où *exprIterator* est un itérateur de la suite *expr*

3.4.6 Blocs

Un bloc est un ensemble d'instructions précédé d'un ensemble de déclarations de variables. On ne mettra pas de déclaration de types ni de constantes dans les blocs.

3.5 Expressions

Les expressions manipulent les variables, les constantes élémentaires et complexes, les opérateurs, les fonctions et les fonctions membres .

On peut trouver des expressions qui ont des effets comme $x++$ ou $foo(y)$ où y est un argument passé en paramètre. Nous rappelons qu'une expression ne peut être assimilée à une instruction, donc nous écrirons $x := x + 1$; ou $x+ = 1$; au lieu de $x + +$;

Les opérateurs et fonctions membres seront repris du langage Java comme traditionnellement. On notera que le signe `=` est réservé pour l'égalité.

Nous dressons dans le tableau suivant les expressions du langage **Léa**

Expression	**Type**	**Valeur**
"..."	string	chaîne de caractères où sont échappés les caractères spéciaux `\t,\n,\r,\\,\"`
'a'	char	caractère où est échappé le caractère spécial `\'`
nombre entier	int	nombre entier décimal signé de Integer.MIN_VALUE à Integer.MAX_VALUE
nombre flottant	int	nombre flottant décimal signé de Float.MIN_VALUE à Float.MAX_VALUE
$True$, $False$	bool	
$E + E$	type des opérandes	addition, concaténation
E - E	type des opérandes	soustraction
E * E	type des opérandes	multiplication
E / E	type des opérandes	division
E % E	type des opérandes	modulo
- E	type de l'opérande	$* -1$
$E < E$	bool	inférieur
$E > E$	bool	supérieur
$E \leq E$	bool	inférieur ou égal
$E \geq E$	bool	supérieur ou égal
$E = E$	bool	égal
E != E	bool	différent
E && E	bool	et
E || E	bool	ou
!E	bool	non
(E)	type de E	identité
$(E_1, E_2, \ldots E_k)$	tuple $(type(E_1), type(E_2), \ldots, type(E_k))$	construction d'un tuple
$[E_1, E_2, \ldots E_k]$	list of $type(E_i)$	construction d'une liste
$[E_1 \cdot\cdot E_k]$	list of $type(E)$	construction d'une liste $[E_1, \ldots, E_k]$ où $i < j \Rightarrow E_i < E_j$ L'opérateur $<$ est donné pour les types élémentaires, il doit être défini pour les classes
$\{E_1, E_2, \ldots E_k\}$	set of $type(E_i)$	construction d'un ensemble. Les opérateurs $<$ et $=$ sont donnés pour les types élémentaires, ils doivent être définis pour les classes
$\{(key_1, val_1), \ldots (key_k, val_k)\}$ ou $\{key_1 => val_1, \ldots key_k => val_k\}$	map of $(type(key_i), type(val_i))$	construction d'une relation. Les opérateurs $<$ et $=$ sont donnés pour les types de key_i élémentaires, ils doivent être définis pour les classes
$foo(E_1, E_2, \ldots E_k)$	$'y$ où $type(foo) =' x \rightarrow' y$	appel d'une fonction foo où $'x$ s'unifie avec $type(E_1) \times type(E_2) \ldots type(E_k)$
$t[E]$	$'x$ où $type(t) = liste\ of\ 'x$	E^{ieme} élément de la liste t (E doit être de type int)
$t[E]$	$'y$ où $type(t) = map\ of\ ('x,' y)$	deuxième élément du E^{ieme} élément de la relation t (E doit être de type $('x,' y)$)
$E.identifier$	type du champs	champs d'une structure, fonction ou variable membre d'une instance de classe ou d'une classe

4 Travail des uns et des autres

1. Le(a) correspondant(e)
   C'est le porte-parole du projet, celui qui écrit sa documentation et assure sa cohérence.
   Il est aussi en charge de coordonner le projet et de résoudre les difficultés théoriques qui se trament au fil des ajouts. Il résoud les problèmes d'interface entre les différents éléments du projet.
   Il devra écrire le fichier `build.xml` et s'assurer de l'architecture complète du projet et de sa compilation. Il devra écrire des programmes tests en langage **Léa**. Il devra être initiateur d'un serveur `svn` pour ses camarades.
   Le fichier `build.xml` produira automatiquement des éléments de debogage, des graphiques, des documentations, des tests, etc. Ce fichier permettra de produire un archive automatiquement et permettra de forcer le nettoyage des fichiers produits par la compilation (`ant clean`)
   Il pourra proposer automatiquement la documentation sous forme d'un hypertexte à partir des commentaires de documentation en tête de chaque fonction et chaque classe. Les commentaires de documentation sont marqués avec `/**` et `**/`
2. Le(a) syntacticien(ne)
   Il écrit le code CUP/JFlex. Il doit intervenir sur la gestion des erreurs de syntaxe et permettre au compilateur une poursuite de calcul en cas d'erreur. Il doit s'efforcer à produire une grammaire LALR.
   On peut ajouter une syntaxe plus complexe pour les classes, l'héritage, l'encapsulation, etc.
   Il serait intéressant de pouvoir traiter des modules et donc de permettre la compilation d'un ensemble de fichiers. Le lexer doit pouvoir traiter un ensemble de flux grâce à des `include`.
3. Le(a) sémanticien(ne)
   Il écrit le code intermédiaire permettant au compilateur de gérer le typage des variables, la gestion du code intermédiaire et sa bonne formation.
   Il serait intéressant de surcharger les fonctions, et donc de ne pas effacer la définition du type d'une fonction par une nouvelle définition. Une solution possible est d'ajouter une table des symboles avec surcharge (qui revient à un ensemble de signatures différentes pour le même nom) concurrente de la table des symboles qui masque une variable d'un bloc à un autre. Il est nécessaire de pouvoir trouver le code de la fonction surchargée par unification entre l'appel et la fonction.
   Exemple :

```
function foo(i: int): int{
        code_1
}

function foo(i: int): bool{
        code_2
}

bool a := foo(6); // appel du code code_2
```

   On pourra ajouter le polymorphisme en utilisant les variables de type. La variable doit alors recevoir un type constant à la compilation (c'est le but du typage statique). Et chaque portion de code dépendant d'une variable de type doit être copié dans le code cible.
4. Un développeur
   Il écrit le code cible en Java en utilisant au maximum les bibliothèques proposées pour ce langage.
   Chaque variable du langage Léa correspond à un objet Java. Il convient d'allouer de la mémoire pour cet objet quand cela est pertinent.

Le type `enum` mérite un encodage avec des vecteurs de bits. Chaque élément d'un type énuméré correspond alors au poids d'un nombre binaire.

Les fonctions et opérateurs surchargés doivent être distingués par leur signature (le type de leurs paramètres et de la valeur rendue). Il convient de faire la gestion de ces noms pour le code cible.

Les paramètres des fonctions sont passés par valeur ou référence explicitement (comme en `C++`). En java, on a un passage par référence implicite si l'argument est un objet, sinon c'est un passage par valeur.

Pour écrire les fonctions, il est possible de produire le code correspondant en Java presque immédiatement. Mais dans le cas d'un passage par valeur pour une fonction récursive, cela pose problème. Il faut trouver une solution pour permettre une programmation récursive avec des arguments de type non élémentaires.

Les fonctions polymorphes sont recopiées pour chaque instance de type calculée en statique. Il serait intéressant de partager une portion constante de ces codes redondants (très difficile).

FIG. 2 – Exemple de code Léa

```
/* ******************************************
* Un exemple complet de code Léa étendu
****************************************** */

// Définition des types
fiche = class {
        static id: int;
        nom: string;
        age: int;


        // définition d'un constructeur
        fiche(){
                this.nom := "";
                this.age := 0;
        }

        // définition de l'opérateur >
        operator >(a: fiche, b: fiche){
                return (a.age > b.age);
        }

}


// Définition d'une fonction
function sort(T: list of 'x) {
        repeat {
                echange : bool := false;
                for i in [0 .. MAX-2] {
                        if (T[j] > T[j+1]){
                                // Variable locale
                                aux: 'x;
                                aux := T[j];
                                T[j] := T[j+1];
                                T[j+1] := aux;
                                echange := true;
                        }
                }
        } while (echange);
}

function main(argn:int, argv list of int): int{
        // Définition des variables
        table: list of fiche;

        // Instructions
        for i in [0 .. MAX-1] {
                table[i].nom := "";
                table[i].age := random(100);
                }

        sort(table);
        return 0;
}
```