Université de Lorraine
Licence Informatique - 2013/2014

UE Compilation - Projet

1. Définition du langage

Le langage **Plic** est un langage objet offrant les mécanismes d'héritage simple, de liaison dynamique, de polymorphisme, de surcharge et de redéfinition sans covariance.

Grammaire

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

En Plic, 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 :
- **cste ent** est une suite non vide de chiffre décimaux ;
- **cste chaîne** est une suite non vide de caractères délimitée par deux guillemets ; pour apparaître dans la chaîne, le guillemet doit être doublé ;
- **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 Plic peut contenir un identificateur CLASSE qui ne sera pas confondu avec le mot clé réservé **classe**.

SYSTEME	→	{ CLASSE }$^+$
CLASSE	→	**classe** **idf** { DECLARATION }* **fin** | **classe** **idf** **herite** **idf** { DECLARATION }* **fin**
DECLARATION	→	DECL_CHAMP | DECL_CONST | DECL_FONCTION
DECL_CHAMP	→	STATUT TYPE **idf** { , **idf** }* ;
TYPE	→	**entier** | **entier** [ ] | **idf** | **idf** [ ]
DECL_CONST	→	STATUT **idf** PARAMETRES **debut** { DECL_VAR }* { INSTRUCTION }$^+$ **fin**
DECL_VAR	→	TYPE **idf** { , **idf** }* ;
DECL_FONCTION	→	STATUT TYPE_RES **idf** PARAMETRES **debut** { DECL_VAR }* { INSTRUCTION }$^+$ **fin**
TYPE_RES	→	**entier** | **idf**
STATUT	→	**publique** | **privee**
PARAMETRES	→	({ TYPE **idf** { ; TYPE **idf** }* })
INSTRUCTION	→	SUPER | AFFECT | INSTANCIATION | BOUCLE | CONDITION | LIRE | ECRIRE | RETOU
SUPER	→	**super** (PAR_EFF) ;
PAR_EFF	→	{ EXP { , EXP }* }
AFFECT	→	ACCES = EXP ;
INSTANCIATION	→	ACCES = **nouveau** **idf** (PAR_EFF) ; | ACCES = **nouveau** **idf** [EXP] ;
	|	ACCES = **nouveau** **entier** [EXP] ;
ACCES	→	**idf** | **idf** [EXP]
BOUCLE	→	**pour** (**idf** = EXP ; EXP ; INSTRUCTION) **debut** { INSTRUCTION }$^+$ **fin** ;

	|	**tantque** EXP **repeter** { INSTRUCTION }$^+$ **fintantque** ;
CONDITION	→	**si** EXP **alors** { INSTRUCTION }* { **sinon** { INSTRUCTION }$^+$ } **fsi** ;
LIRE	→	**lire** **idf** ;
RETOURNE	→	**retourne** EXP ;
ECRIRE	→	**ecrire** EXP ; | **ecrire** **cste chaîne** ;
EXP	→	**idf**.**longueur** | **cste ent** | **null** | (EXP) | - EXP | EXP OPER EXP
	|	**idf** [EXP] | **idf** (PAR_EFF) { . **idf** { (PAR_EFF) } }* | **idf** { . **idf** { (PAR_EFF) } }*
OPER	→	+ | - | * | > | < | == | !=

Sémantique

1. Un système est un ensemble de classes, toutes contenues dans le même fichier. Lors de la commande de compilation, l'une de ces classes est désignée comme étant la racine de l'appliction ; elle doit obligatoirement contenir un constructeur public sans paramètre, considéré comme étant le point d'entrée de l'application.
2. La portée de la déclaration d'une classe est l'ensemble du fichier qui la contient. 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.
3. Une région est délimitée par les mots clés **classe** et **fin** ou **debut** et **fin**.
4. Dans une déclaration de types, la notation T [ ] désigne un type tableau à une dimension dont les éléments sont de type T. Les tableaux sont dynamiques. Leur taille est fixée lors de l'instanciation.
5. Les champs et méthodes privées ne sont pas utilisables par les clients de la classe. Les champs et méthodes publics sont utilisables par tous les clients.
6. Les doubles déclarations de classes sont interdites. Les doubles déclarations de variables sont interdites. Une classe et une variable peuvent porter le même nom. La surcharge des fonctions est autorisée, sous réserve de définir des profils différents (le profil est défini uniquement par le type des paramètres). Une variable et une fonction peuvent porter le même nom ; la différence est faite syntaxiquement : un appel de fonction est obligatoirement suivi d'une parenthèse ouvrante.
7. Dans une classe, les constructeurs déclarés par **DECL_CONST** portent obligatoirement le nom de la classe. Ils peuvent avoir un nombre quelconque de paramètres, éventuellement nul. La définition d'un constructeur n'est pas obligatoire ; par défaut, il existe toujours un constructeur sans paramètre dont l'effet est d'initialiser les champs avec les valeurs par défaut (0 pour les entiers et null pour les autres).
8. Une sous-classe hérite (mot clé **herite**) de tous les champs et de toutes les fonctions de sa super-classe, ainsi que du statut d'exportation de ceux-çi. La redéfinition d'une fonction héritée est possible, sans modification du type des paramètres, ni du résultat. La définition d'un champ de même nom qu'un champ de la super-classe masque le champ hérité.
9. Dans un constructeur d'une sous-classe, un appel à **super** permet d'appeler le constructeur de la super-classe dont le profil correspond à cette utilisation ; cet appel doit être la première instruction du constructeur.
10. Dans une affectation avec instanciation (mot clé **nouveau**, première alternative), l'identificateur en partie gauche est un identificateur de variable de type T. L'identificateur en partie droite est celui d'un constructeur du type T, dont le profil correspond à cette utilisation.
11. Dans une affectation avec instanciation (mot clé **nouveau**, deuxième alternative), l'identificateur en partie gauche est un identificateur de variable de type T [ ]. L'identificateur en partie droite est obligatoirement T. L'expression entre crochets est de type entier ; elle désigne le nombre d'éléments alloués.
12. Dans une affectation avec instanciation (mot clé **nouveau**, troisième alternative), l'identificateur en partie gauche est un identificateur de variable de type entier. L'expression entre crochets est de type entier ; elle désigne le nombre d'éléments alloués.
13. On interdit le passage de tableau en paramètre.

14. L'instruction **retourne** est obligatoire. Son exécution provoque l'arrêt de la fonction en retournant comme résultat la valeur de l'expression.
15. Deux types identiques concordent. Le type T [ ] ne concorde qu'avec lui-même (équivalence nominale). Le type ST concorde avec le type T si la classe ST hérite directement ou indirectement de la classe T.
16. Dans une affectation sans instanciation, le type de l'expression doit concorder avec celui de la notation d'accès en partie gauche.
17. Dans un accès utilisant la notation pointée, le compilateur choisit le profil de la fonction à partir du type des paramètres effectifs. Lors de l'exécution, la liaison dynamique s'applique pour définir l'opération effectivement appelée en fonction du type dynamique du receveur.
18. Dans un accès à un tableau (notation avec [ ]), l'expression est de type entier. Les éléments du tableau sont indicés à partir de 0.
19. Dans une itération simple (mot clé **pour**), les informations entre parenthèses définissent le contrôle de la boucle. L'identificateur sert de variable de boucle. Son initialisation sous-entend une déclaration de la variable de type entier, dans le bloc des instructions itérées. La seconde expression est la dernière valeur de la variable de boucle pour laquelle les instructions du bloc seront répétées. L'instruction est quelconque (elle devrait modifier la variable de la boucle, mais le compilateur n'est pas censé le vérifier, sauf s'il optimise !). Dans les instructions itérées, la modification de la variable de boucle, ainsi que des variables intervenant dans l'expression qui désigne la valeur finale, est tolérée mais n'influence pas le comportement de l'itération (toute affectation à cette variable doit être ignorée).
20. Dans une conditionnelle (mot clé **tantque**), l'expression est de type booléen ; elle est obligatoirement évaluée avant toute entrée dans la boucle.
21. Dans une instruction conditionnelle (mot clé **si**), l'expression est de type booléen.
22. La fonction prédéfinie **lire** lit un entier sur l'entrée standard.
23. La fonction prédéfinie **ecrire** écrit, soit un entier, soit une chaîne de caractères, sans aller à la ligne. Le programmeur Plic peut demander un retour explicite à la ligne en incluant le caractère \n dans la chaîne écrite.
24. Si les deux opérandes des opérateurs binaires +, - et * sont entiers, le résultat est entier ; s'ils sont booléens, le résultat est booléen ; ces opérateurs notent alors respectivement le **ou**, le **ou exclusif** et le **et** logiques. L'opérateur unaire - note l'opposé pour un entier et la négation pour un booléen. Les opérateurs relationnels <, >, == et != sont à opérandes entiers et résultat booléen.
25. Les opérateurs binaires sont tous associatifs gauche-droite ; les opérateurs relationnels ont la plus grande priorité, avant celle de l'opérateur *, avant celle des opérateurs + et -.
26. Lors de l'exécution, l'ordre d'évaluation des opérandes d'un opérateur n'est pas fixé ; il est choisi dans un souci d'optimisation.
27. Le passage des paramètres se fait par valeur ; il n'est autorisé qu'entre objets de types qui concordent.
28. La notation **idf.longueur** désigne le nombre d'éléments du tableau de nom **idf**. C'est une expression de type entier.
29. En mémoire, les entiers sont représentés directement par leur valeur ; les autres objets, y compris les tableaux, sont représentés par un pointeur. Aucune initialisation particulière n'est prévue par défaut. L'allocation de l'objet est réalisée par l'instruction **nouveau**.

2. Le Compilateur Plic

Le compilateur Plic 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 :

- la commande de compilation s'appelle **plic** ; elle attend exactement deux arguments : le nom du fichier contenant le texte des classes à compiler, suffixé `.plic` , suivi du nom de la classe racine du système. Elle provoque la compilation du texte et, si celle-ci se déroule sans erreur, crée un fichier de même préfixe, suffixé `.asm`, contenant le texte cible correspondant.
- la commande **plic** n'est pas conversationnelle et doit laisser intact l'environnement de celui qui l'appelle (en particulier, elle ne doit pas créer de fichiers temporaires ailleurs que sur /tmp).
- selon le cas, la commande **plic** doit produire sur la sortie standard l'un des résultats suivants :

  ERREUR LEXICALE : ligne d'erreur : message d'erreur explicite
  ERREUR SYNTAXIQUE : ligne d'erreur : message d'erreur explicite
  ERREUR SEMANTIQUE : ligne d'erreur : message d'erreur explicite
  COMPILATION OK *suivi du résultat d'exécution*
- une erreur lexicale ou syntaxique stoppe la compilation du texte source ; toutes les erreurs sémantiques détectées doivent être signalées ; une erreur lors de l'exécution stoppe l'exécution du code généré.

3. Déroulement du projet

Vous travaillerez en binôme. Pour vous rafraîchir la mémoire, vous trouverez sur Arche le document de l'an dernier relatif à l'utilisation des deux générateurs d'analyseurs.

La réalisation de ce compilateur doit se faire par noyaux successifs du langage. En annexe, vous trouverez la composition des différents noyaux à compiler. Dès que le compilateur d'un noyau est terminé (et complètement testé), vous pouvez, après accord de l'enseignant, commencer le développement du noyau suivant.

Voici, à titre indicatif, les dates limites auxquelles les différents noyaux doivent être terminés :

Plic0	16 avril 2014
Plic1	15 mai 2014
Plic2	30 mai 2014
Plic3	12 juin 2014

Grammaire Plic0 – expressions arithmétiques

EXP	→	**cste ent** | (EXP) | - EXP | EXP OPER EXP
OPER	→	+ | - | * | > | < | == | !=

Grammaire Plic1 – une seule classe avec déclarations d'attributs

SYSTEME	→	CLASSE
CLASSE	→	**classe** **idf** { DECLARATION }* **fin**
DECLARATION	→	DECL_CHAMP | DECL_CONST
DECL_CHAMP	→	STATUT TYPE **idf** { , **idf** }* ;
STATUT	→	**publique** | **privee**
TYPE	→	**entier**
DECL_CONST	→	{ INSTRUCTION }$^+$
INSTRUCTION	→	AFFECT | LIRE | ECRIRE
AFFECT	→	ACCES = EXP ;
ACCES	→	**idf**
LIRE	→	**lire** **idf** ;
ECRIRE	→	**ecrire** EXP ; | **ecrire** **cste chaîne** ;
EXP	→	**cste ent** | (EXP) | - EXP | EXP OPER EXP | **idf**
OPER	→	+ | - | * | > | < | == | !=

Exemple de programme Plic1

```
classe Essai
        // La seule et unique classe de Plic1
        b = a + 25 ;
        publique entier a, b ;
        a = 0 ;

        privee entier c1, d2, de ;
        privee entier A, unNomDidentificateurTresLongPourEmbeterLesCompilateursDuLangagePlic1 ;

        ecrire "programme de test\n" ;
        ecrire "\n\n" ;
        ecrire a ; ecrire "\n" ;

        // je calcule des // "expressions"

        A = 0012 ; A = + 0000012 + A ;

        c1 = (a+10) * 12 * (-1) ; f = 5 ;
        d2 = (a*3 - 3 + c1*f) ;
        unNomDidentificateurTresLongPourEmbeterLesCompilateursDuLangagePlic = c1 ;

        ecrire unNomDidentificateurTresLongPourEmbeterLesCompilateursDuLangagePlic ; ecrire "\n" ;
        ecrire "ecriture d'un mot entre guillemets ""bonjour""" ;
        ecrire "\n" ;
```

```
lire f ; lire Essai ;
publique entier f ;
Essai = 1 ;
publique entier Essai, essai ;
essai = 2*Essai ; fin
```

Grammaire Plic2 – les classes, les fonctions et les constructeurs

SYSTEME	→	{ CLASSE } *
CLASSE	→	**classe** **idf** { DECLARATION }* **fin**
DECLARATION	→	DECL_CHAMP | DECL_CONST | DECL_FONCTION
DECL_CHAMP	→	STATUT TYPE **idf** { , **idf** }* ;
DECL_CONST	→	STATUT **idf** PARAMETRES **debut** { DECL_VAR }* { INSTRUCTION }+ **fin**
DECL_FONCTION	→	STATUT TYPE_RES **idf** PARAMETRES **debut** { DECL_VAR }* { INSTRUCTION }+ **fin**
STATUT	→	**publique** | **privee**
TYPE	→	**entier**
TYPE_RES	→	**entier**
PARAMETRES	→	({ TYPE **idf** { ; TYPE **idf** }* })
PAR_EFF	→	{ EXP { , EXP }* }
DECL_VAR	→	TYPE **idf** { , **idf** }* ;
INSTRUCTION	→	AFFECT | LIRE | ECRIRE | RETOURNE
AFFECT	→	ACCES = EXP ;
ACCES	→	**idf**
LIRE	→	**lire** **idf** ;
ECRIRE	→	**ecrire** EXP ; | **ecrire** **cste chaîne** ;
RETOURNE	→	**retourne** EXP ;
EXP	→	**cste ent** | (EXP) | - EXP | EXP OPER EXP | **idf** | **idf** (PAR_EFF)
OPER	→	+ | - | * | > | < | == | !=

Exemple de programme Plic2

```
// Test de declarations
classe Essai1                                   // bloc 0
        // Declarations de champs
        publique entier x , y ;
        publique entier a ;
        // Declarations de constructeurs
        publique Essai1(entier x, entier y)     // bloc 1
                debut
                        x = a ;
                fin
        publique Essai1(entier x)               // bloc 2
                debut
                        y = x+a + f (x, a) ;
                fin
        // Declaration d'une fonction
        publique entier f (entier a, entier b)
                debut
```

```
            entier x ;
                x = a + b ;
                retourne(x) ;
            fin
fin
classe Test                                 // bloc 0
    publique entier a ;
    publique Test(entier y)                 // bloc 1
            debut
                x = a + y ;
            fin
    publique entier x , y ;
    publique Test()                         // bloc 2
            debut
                y = x+a ;
            fin
fin
```

Grammaire Plic3 – l'héritage

SYSTEME	→	{ CLASSE } *
CLASSE	→	**classe** **idf** { DECLARATION }* **fin** | **classe** **idf** **herite** idf { DECLARATION }* **fin**
DECLARATION	→	DECL_CHAMP | DECL_CONST | DECL_FONCTION
DECL_CHAMP	→	STATUT TYPE **idf** { , **idf** }* ;
DECL_CONST	→	STATUT **idf** PARAMETRES **debut** { DECL_VAR }* { INSTRUCTION }+ **fin**
DECL_FONCTION	→	STATUT TYPE_RES **idf** PARAMETRES **debut** { DECL_VAR }* { INSTRUCTION }+ **fin**
STATUT	→	**publique** | **privee**
TYPE	→	**entier**
TYPE_RES	→	**entier**
PARAMETRES	→	({ TYPE **idf** { ; TYPE **idf** }* })
PAR_EFF	→	{ EXP { , EXP }* }
DECL_VAR	→	TYPE **idf** { , **idf** }* ;
INSTRUCTION	→	AFFECT | LIRE | ECRIRE | RETOURNE
AFFECT	→	ACCES = EXP ;
ACCES	→	**idf**
LIRE	→	**lire** **idf** ;
ECRIRE	→	**ecrire** EXP ; | **ecrire** **cste chaîne** ;
RETOURNE	→	**retourne** EXP ;
EXP	→	**cste ent** | (EXP) | - EXP | EXP OPER EXP | **idf** | **idf** (PAR_EFF)
OPER	→	+ | - | * | > | < | == | !=